National Technical University of Athens
School of Electrical & Computer Engineering
Division of Communication, Electronic and Information Engineering

Deep Learning techniques for system optimization in Cloud architectures

A thesis submitted for the degree of
Doctor of Philosophy

by

Dimosthenis G. Masouros

Athens, May 2023

National Technical University of Athens
School of Electrical & Computer Engineering
Division of Communication, Electronic and Information Engineering

Deep Learning techniques for system optimization in Cloud architectures

Ph.D. Thesis
of
Dimosthenis G. Masouros

Athens, May 2023

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών & Μηχανικών Υπολογιστών
Τομέας Επικοινωνιών, Ηλεκτρονικής και Συστημάτων Πληροφορικής

Τεχνικές Βαθιάς Μάθησης για Βελτιστοποίηση Συστημάτων Υπολογιστικού Νέφους

Διδακτορική Διατριβή
του
Δημοσθένη Γ. Μασούρου

Αθήνα, Μάιος 2023

National Technical University of Athens
School of Electrical & Computer Engineering
Division of Communication, Electronic and Information Engineering
Microprocessors and Digital Systems Lab

Deep Learning techniques for system optimization in Cloud architectures

Ph.D. Thesis
of
Dimosthenis G. Masouros

Supervising Committee: Dimitrios Soudris
Kiamal Pekmestzi
Michael Huebner

Approved by the advisory committee on May, 2023.

....	....	....
Dimitrios Soudris	Kiamal Pekmestzi	Michael Huebner
Professor N.T.U.A	Professor N.T.U.A	Professor B.T.U

....	....
Sotirios Xydis	Georgios Goumas
Assistant Professor N.T.U.A.	Associate Professor N.T.U.A
....	....
Cristina Silvano	Dionisios Pnevmatikatos
Professor POLI.MI.	Professor N.T.U.A

Athens, May 2023

....
Δημοσθένης Γ. Μασούρος
Διδάκτωρ Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Abstract

Nowadays, Cloud computing is becoming one of the most attractive solutions for application execution, due to the enhanced flexibility and efficiency it offers. Cloud computing refers to the on-demand provision of system resources, especially computing power and storage. These resources typically exist on individual servers found in datacenter environments and/or server farms around the world. Proper and efficient management of those resources becomes crucial, both from the providers' as well as end-users' point of view, since it can dramatically improve performance and reduce cost expenses for all the parties involved. However, orchestrating cloud computing resources is not a straightforward matter, due to i) the huge amount of available optimization knobs, ii) the different levels that these optimizations can be applied (server- to cluster- to application-level) and iii) the interrelationship between software and hardware optimizations combined with the extreme hardware heterogeneity found in today's data-centers environments. On top of that novel computing paradigms are emerging (e.g., hardware disaggregation), which unveil extra optimization knobs on the foreground, thus, further complicating the problem of efficient resource orchestration.

In this dissertation, we examine the applicability of deep learning techniques for system optimization in Cloud architectures. Given the huge amount and the multi-level nature of the available optimization knobs, which tend to be unmanageable using conventional, human-driven orchestration mechanisms, deep learning approaches appear to be unavoidable in order to exploit and leverage the full potential of modern Cloud systems. First, we investigate the employment of state-of-the-art neural networks in the field of system monitoring, which forms an integral part of modern Cloud infrastructures. Then, we examine the application of ML-driven orchestration on different levels of the hierarchy, i.e., application-driven automatic optimizations, cluster-level application deployment and orchestration and system-level control of running applications. To achieve the above, we propose three frameworks, i.e., Rusty, Adrias and Sparkle, which employ deep learning driven optimizations on different levels of the hierarchy.

Keywords: Cloud Computing, Deep Learning, Machine Learning, Resource Management, Cloud Monitoring

Περίληψη

Στις ημέρες μας, η εκτέλεση εφαρμογών μέσω χρήσης του "Υπολογιστικού Νέφους" αποτελεί μία από τις πιο ελκυστικές και ευρέως υιθετούμενες λύσεις, λόγω της μεγάλης ευελιξίας και αποτελεσματικότητας που προσφέρει. Το "Υπολογιστικό Νέφος" ουσιαστικά εξυπηρετεί στην κατ΄ απαίτηση παροχή υπολογιστικών πόρων σε τελικούς χρήστες και τις περισσότερες φορές αναφέρεται σε υπολογιστική ισχύ και χώρο αποθήκευσης δεδομένων. Αυτοί οι πόροι συνήθως είναι κομμάτι κάποιων μεμονωμένων διακομιστών, οι οποίοι βρίσκονται σε περιβάλλοντα κέντρων δεδομένων ή/και σε φάρμες διακομιστών ανά τον κόσμο. Η σωστή και αποδοτική διαχείριση αυτών των πόρων γίνεται κρίσιμη, τόσο από την πλευρά των παρόχων όσο και των τελικών χρηστών, καθώς μπορεί να βελτιώσει δραστικά την απόδοση και να μειώσει τα έξοδα κόστους τόσο για τους τελικούς χρήστες όσο και για τους παρόχους υπολογιστικών υπηρεσιών. Ωστόσο, η ενορχήστρωση των πόρων υπολογισμού στο Νέφος δεν είναι απλή υπόθεση, λόγω 1) της τεράστιας ποσότητας διαθέσιμων παραμέτρων προς βελτιστοποίηση, 2) των διαφορετικών επιπέδων στα οποία μπορούν να εφαρμοστούν αυτές οι βελτιστοποιήσεις (επίπεδο διακομιστή - επίπεδο συστάδας - επίπεδο εφαρμογής) και ιιι) της ισχυρής αλληλεπίδρασης μεταξύ λογισμικού και υλικού σε συνδυασμό με την έντονη ανομοιογένεια του υλικού που βρίσκεται στα σύγχρονα περιβάλλοντα κέντρων δεδομένων. Επιπλέον, νέα υπολογιστικά πρότυπα αναδύονται, τα οποία φέρνουν στο προσκήνιο επιπλέον σημεία βελτιστοποίησης, επομένως, περιπλέκουν περαιτέρω το πρόβλημα της αποτελεσματικής οργάνωσης των πόρων.

Σε αυτήν τη διατριβή, εξετάζουμε την εφαρμογή τεχνικών βαθιάς μάθησης για τη βελτιστοποίηση συστημάτων σε αρχιτεκτονικές "Υπολογιστικού Νέφους". Λαμβάνοντας υπόψη τον μεγάλο αριθμό και το πολυεπίπεδο χαρακτήρα των διαθέσιμων σημείων βελτιστοποίησης, τα οποία τείνουν να είναι δύσκολα διαχειρίσιμα με συμβατικούς, ανθρώπινα καθοδηγούμενους μηχανισμούς οργάνωσης, οι προσεγγίσεις βαθιάς μάθησης φαίνεται να είναι αναπόφευκτες για την αξιοποίηση και εκμετάλλευση του πλήρους δυναμικού των σύγχρονων συστημάτων Νέφους. Προς αυτήν την κατεύθυνση, σαν πρώτο αντικείμενο της διατριβής, ερευνούμε τη χρήση νευρωνικών δικτύων στον τομέα των συστημάτων παρακολούθησης στο Υπολογιστικό Νέφος, η οποία αποτελεί σημαντική συνιστώσα των σύγχρονων υποδομών Υπολογιστικού Νέφους. Στη συνέχεια, εξετάζουμε την αποτελεσματικότητα της τεχνητής νοημοσύνης για βελτιστοποίηση διαχείρισης πόρων σε διαφορετικά επίπεδα της ιεραρχίας, δηλαδή αυτόματες βελτιστοποιήσεις σε επίπεδο εφαρμογής, καθώς και την ανάπτυξη και οργάνωση σε επίπεδο συστάδας υπολογιστών. Για να επιτευχθούν το παραπάνω, προτείνουμε τρία εργαλεία, το Rusty, το Adrias και το Sparkle, καθένα από τα οποία χρησιμοποιεί βελτιστοποιήσεις καθοδηγούμενες από βαθιά νευρωνικά δίκτυα και στοχεύουν σε διαφορετικά επίπεδα της ιεραρχίας.

Λέξεις Κλειδιά: Υπολογιστικό Νέφος, Βαθιά Μάθηση, Μηχανική Μάθηση, Οργάνωση Υπολογιστικών Πόρων, Σύστημα Παρακολούθησης Υπολογιστικού Νέφους

Acknowledgements

While residing within the preamble of the thesis, this section represents the final piece of my PhD puzzle, and perhaps the most challenging one to articulate. Even though just a couple of pages long, it holds incredible importance, as it highlights the effort of many people who have done their bit for this moment to come. It is here that I attempt to express my deepest gratitude to all those who have played a significant role in shaping my doctoral journey.

First, I would like to express my gratitude to my supervisor, Prof. Dimitrios Soudris, who gave me the opportunity to pursue my PhD in the Microprocessors and Digital Systems laboratory. Prof. Soudris always guided and supported me throughout my PhD, providing continuous feedback and always pushing me to do and achieve more and more. But it is not only the professional aspect for which I would like to thank Prof. Soudris, but primarily the personal one. Prof. Soudris was always there whenever a problem arose (at any level), and he always tried to find a solution and help as much as possible, like a “second father”. Our numerous (and many hours-long discussion) throughout my doctoral studies helped me broaden and mature my way of thinking. As he often says, through MicroLab, I became a “different person”, and I believe a significant part of this “transformation” is owed to him. I sincerely hope that our relationship will continue to hold till (and beyond) the day that “he takes his revenge”.

I would also like to thank Prof. Sotirios Xydis. Probably one paragraph of ink is not enough to express my full gratitude for this person. I had the pleasure of collaborating with Sotiris on numerous projects, through which I evolved as an engineer. However, beyond the projects, Sotiris was the person who closely supervised the progress of my PhD. He was the one who gave shape and substance to the ideas of this dissertation and more. Always aiming for excellence and setting high goals, he led me to broaden my horizons and evolve as a researcher. Truly, I believe Sotiris is the best mentor I could have asked for, and he is the person who shaped me intellectually. For that, I am extremely thankful and I hope that we will continue to come up with new research ideas together in the future.

Of course, I could never forget to thank the entire MicroLab family. Firstly, I express my gratitude to Vasilis Tsoutsouras, who introduced me to MicroLab and encouraged Prof. Soudris to accept me as a candidate. Also, many thanks to my colleague and friend Georgios Zervakis, who, through his guidance (and many beers we had) helped me evolve as an engineer. Moreover, I would like to thank the individuals with whom we started and finished this journey together: Konstantina, Ioannis, Haris, and Vasilis. Each of them contributed in their own way to creating a pleasant atmosphere in the laboratory. As knowledge is not a “one-way game”, I would also like to thank all the students with whom I collaborated on thesis projects, as they helped me grow as a researcher. Special thanks to Achilleas, Manolis, Ioannis, and Aggelos, who were the first students I collaborated with and who, in turn, dared to explore the PhD path. Also, many thanks to Petros and Antreas for their great work, part of which is included in this thesis. Last but not least, special thanks go to the “heart” of MicroLab, Zefi. Her smile and pleasant presence always brightened our days in MicroLab.

Next, I would like to acknowledge the individuals with whom I closely collaborated during my Phd. Firstly, I would like to thank Prof. Kostas Siozios, who entrusted me with the opportunity to work together on the SERRANO project, but also for the enjoyable moments we had throughout these years. Also, I would like to express my gratitude to Christian Pinto and Michele Gazzetti, with whom I closely collaborated during my internship in IBM Research in Dublin. I hope they do not “hold a grudge” against me for choosing to follow the academia path!

Apparently, a challenging journey cannot come to an end without the existence of joyful moments with close friends to break the monotony and rejuvenate yourself. Firstly, I would like to thank Dr. George Retsinas and (hopefully Dr. soon) Athena Elafrou, two close friends who I met on my undergraduate studies and who always supported me both regarding research and personal matters. Next, thanks to two of my best friends, John and Dora, who have always been on my side. The “Tuesday’s movie nights” really recharged my batteries every time to keep up the work of my PhD. Also thanks to Babis, John V., John K., George, Marianna and Aristea for the many gaming and drinking nights we had.

Last, getting my PhD wasn’t just about the six years of hard work. It was a journey that started long before, thanks to the love and support of my family. They prepared me for this moment and helped me become the person who could take on the challenge. For this reason, I would like to thank my parents, George and Tota, my sister Vicky, her husband Dimos and my niece Alina. Also thanks to all the rest of my relatives and my godmother Athina, who were always there supporting my steps.

Εκτεταμένη Ελληνική Περίληψη

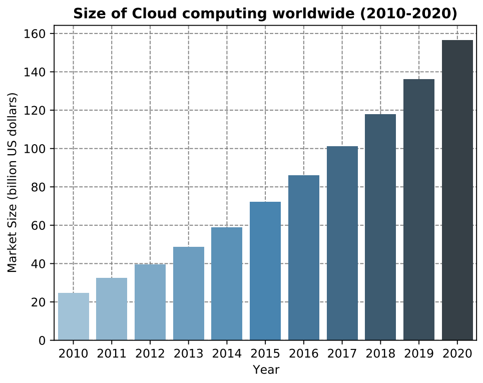
1. Εισαγωγή

Στις ημέρες μας, βιώνουμε την "Ψηφιακή Εποχή" ή "Ψηφιακή Επανάσταση", όπου ο ρυθμός με τον οποίο η τεχνολογία εξελίσσεται, αυξάνεται διαρκώς σε όλο τον κόσμο. Αυτό το ψηφιακό "τσουνάμι" είναι απόρροια ενός συνδυασμού τεχνολογικών αναπτύξεων αλλά και κοινωνικοοικονομικών παραγόντων και τάσεων, συμπεριλαμβανομένων μεταξύ άλλων της ταχείας προόδου στον τομέα των δικτύωσης και συνδεσιμότητας [9], της ραγδαίας αύξησης του αριθμού συνδεδεμένων συσκευών στο διαδίκτυο [10], τον τεράστιο όγκο δεδομένων που δημιουργούν αυτές οι συσκευές και τις γνώσεις που μπορούμε να αποκομίσουμε από αυτά [11], καθώς και την ανάγκη για γρήγορες και ευέλικτες αντιδράσεις σε φυσικές ή ανθρωπογενείς καταστροφές ή καταστάσεις κρίσης [12].

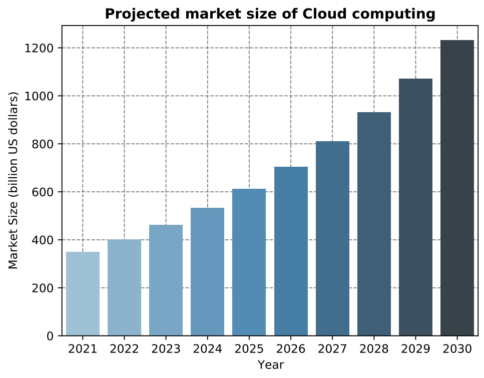
Η χρήση του "Υπολογιστικού Νέφους" αποτελεί έναν σημαντικό πυλώνα στο πλαίσιο αυτής της ψηφιακής εναλλαγής, καθώς επιτρέπει στους ενδιαφερόμενους να ψηφιοποιήσουν γρήγορα τις διάφορες υπηρεσίες της επιχείρησής τους, χωρίς να χρειάζεται να διαθέσουν υψηλό αρχικό κεφάλαιο για επένδυση σε υλικό καθώς και χρόνο για τη διαχείρισή του. Μάλιστα, η ταχεία υιοθέτηση αυτού του νέου προτύπου γίνεται ακόμα πιο εμφανής εάν κοιτάξουμε διάφορες πρόσφατες αγοραστικές αναφορές (Σχήμα 1), όπου φαίνεται πως τα τελευταία 10 χρόνια το μέγεθος της αγοράς υπηρεσιών "Υπολογιστικού Νέφους" αυξήθηκε πάνω από 6 φορές, ενώ προβλέπεται να αυξηθεί και περαιτέρω μέσα στα επόμενα χρόνια[1].

Τα Υπερκλιμακούμενα Κέντρα Δεδομένων (ΥΚΔ/ΚΔ) προσφέρουν όλη την απαραίτητη υπολογιστική δύναμη (π.χ., δίσκους για αποθήκευση δεδομένων, χιλιάδες διακομιστές για εκτέλεση εφαρμογών, κ.α.) για την εκτέλεση των διάφορων υπηρεσιών του "Υπολογιστικού Νέφους". Παρά την τεράστια διαθεσιμότητα υπολογιστικών πόρων, τα ΚΔ παρουσιάζουν αρκετές προκλήσεις, κυρίως αναφορικά με την αποτελεσματική διαχείριση των διαθέσιμων πόρων, προκειμένου αυτοί να μπορούν να εξυπηρετούν τον τεράστιο αριθμό πελατών, όσο πιο αποδοτικά γίνεται. Μία από τις κυριότερες προκλήσεις που εμφανίζονται στα ΚΔ είναι

[1] Size of the cloud computing and hosting market market worldwide from 2010 to 2020



(α΄) Μέγεθος της αγοράς του Υπολογιστικού Νέφους το διάστημα 2010 έως 2020



(β΄) Πρόβλεψη μεγέθους της αγοράς του Υπολογιστικού Νέφους από το 2021 έως το 2030

Σχήμα 1.. Μέγεθος της αγοράς του Υπολογιστικού Νέφους παγκοσμίως

η αποδοτική διαχείριση των διαθέσιμων υπολογιστικών πόρων. Δεδομένου ότι η αγορά και συντήρηση του εξοπλισμού των ΚΔ είναι μία αρκετά πολυδάπανη διαδικασία, οι πάροχοι υπηρεσιών Υπολογιστικού Νέφους στοχεύουν συνεχώς στο να βελτιστοποιήσουν την χρησιμοποίησή των διαθέσιμων υπολογιστικών πόρων, προκειμένου να μεγιστοποιήσουν την αποδοτικότητα τους και, συνεπώς, να μειώσουν τα έξοδα αναβάθμισης εξοπλισμού και να μεγιστοποιήσουν το κέρδος τους.

Ενώ η καλύτερη αποδοτικότητα των πόρων μπορεί να επιτευχθεί με την εφαρμογή βελτιστοποιήσεων σε διάφορα επίπεδα (από τον διακομιστή αυτόν καθαυτόν, στο επίπεδο συστοιχίας διακομιστών και στο επίπεδο της εφαρμογής), τα σύγχρονα πλαίσια διαχείρισης πόρων δεν είναι σε θέση να διαχειριστούν την πολυπλοκότητα των σύγχρονων υποδομών Υπολογιστικού Νέφους. Αυτό έχει ως συνέπεια τα κέντρα δεδομένων να υπολειτουργούν, καθώς οι πάροχοι υπηρεσιών Υπολογιστικού Νέφους τείνουν να θυσιάζουν την υπολογιστική ισχύ προκειμένου να πετύχουν καλύτερες επιδόσεις για τους τελικούς χρήστες [29, 30]. Ένας τρόπος με τον οποίο οι πάροχοι υπηρεσιών Υπολογιστικού Νέφους προσπαθούν να μειώσουν την υπολειτουργεία των διακομιστών εντός των ΥΚΔ είναι μέσω της συντοποθέτησης εφαρμογών σε κοινούς διακομιστές. Σε αυτό το μοντέλο, εφαρμογές από διαφορετικούς χρήστες τοποθετούνται σε κοινούς διακομιστές, επιτυγχάνοντας έτσι την αύξηση του ποσοστού χρησιμοποίησης των πόρων των μηχανημάτων. Ωστόσο, ταυτόχρονα με την αύξηση του ποσοστού χρησιμοποίησής των πόρων, η συντοποθέτηση εφαρμογών σε κοινούς διακομιστές οδηγεί ταυτόχρονα και σε παρεμβολές στους κοινούς πόρους των συστημάτων, το οποίο με τη σειρά του οδηγεί στη μείωση της απόδοσης των εφαρμογών. Στην πραγματικότητα, παρεμβολές μπορούν να υπάρξουν σε πολλά διαφορετικά επίπεδα του συστήματος, από το επίπεδο του επεξεργαστή, στις κρυφές μνήμες καθώς και στους διαύλους επικοινωνίας με την κύρια μνήμη.

Επίσης, με στόχο την αντιμετώπιση του προβλήματος της υποχρησιμοποίησης των πόρων (καθώς και του κατακερματισμού αυτών) οι πάροχοι υπηρεσιών Υπολογιστικού Νέφους τείνουν προς την υιοθέτηση καινοτόμων υπολογιστικών μοντέλων, όπως για παράδειγμα το παράδειγμα της "Αποσύνθεσης Υπολογιστικών Πόρων". Σε αυτό το μοντέλο, οι πόροι ενός ΚΔ οργανώνονται σε διακομιστές οι οποίοι περιλαμβάνουν αμιγώς συγκεκριμένου τύπου πόρους (π.χ. διακομιστές αποκλειστικά και μόνο με μνήμη).

Όπως είναι ξεκάθαρο από τα παραπάνω, η βελτιστοποίηση της χρησιμοποίησής των πόρων μέσα σε ένα ΚΔ αποτελεί ένα πολύ δύσκολο και πολύπλοκο πρόβλημα. Παρ'όλο που οι πάροχοι υπηρεσιών Υπολογιστικού Νέφους, διαθέτουν μηχανισμούς για την τοποθέτηση εφαρμογών στους διαθέσιμους υπολογιστικούς πόρους, οι μηχανισμοί αυτοί βασίζονται κατά κύριο λόγο σε απλοϊκούς αλγορίθμους οργάνωσης καθώς και μηχανισμούς δρομολόγησης των εφαρμογών, οι οποίοι δεν καθίστανται ικανοί να αντιμετωπίσουν την οργάνωση των πόρων με αποδοτικό τρόπο. Προκειμένου να υλοποιηθούν πιο "έξυπνοι" μηχανισμοί διαχείρισης πόρων, τελευταία παρατηρούμε να μελετάται εκτενώς η υιοθέτηση της Τεχνητής Νοημοσύνης (ΤΝ) και της Μηχανικής Μάθησης (ΜΜ) στον τομέα του Υπολογιστικού Νέφους. Συγκεκριμένα, με την εφαρμογή τεχνικών ΜΜ, οι πάροχοι υπηρεσιών Υπολογιστικού Νέφους είναι σε θέση να επεξεργαστούν, να αναλύσουν και να αναγνωρίσουν συσχετίσεις μεταξύ τεράστιων όγκων δεδομένων, πράγμα που θα ήταν αδύνατο με χρήση των παλαιότερων μηχανισμών διαχείρισης των υπολογιστικών πόρων. Πράγματι, για τους παρόχους δημόσιου νέφους, η τεχνητή νοημοσύνη (AI) και η ΜΛ αποτελούν ήδη μέρος της υποδομής και των λειτουργιών των κέντρων δεδομένων τους[2].

Αν και η ΤΝ και η ΜΜ αποτελούν κυρίαρχες λύσεις για πιο αποδοτικά ΚΔ, είναι ακόμα ασαφές ποιος είναι ο καλύτερος τρόπος να ενσωματωθούν σε μηχανισμούς για τη διαχείριση πόρων στο Υπολογιστικό Νέφος. Στην πραγματικότητα, διάφορες προηγούμενες προσπάθειες ενσωμάτωσης ποικίλλουν κατά πολύ στον τρόπο προσέγγισης του προβλήματος. Για παράδειγμα, όπως αναφέρθηκε προηγουμένως, σε ορισμένες περιπτώσεις, η ΜΜ συνδέεται στενά με τη διαχείριση των πόρων, αλλά είναι εντελώς διακριτή σε άλλες περιπτώσεις [28]. Προς αυτήν την κατεύθυνση, απαιτείται περαιτέρω έρευνα για να κατανοήσουμε πώς, πού και πότε να χρησιμοποιήσουμε τεχνικές ΜΛ για τη βελτίωση της αποδοτικότητας των πόρων στα συστήματα νέφους.

2. Στόχος Διδακτορικής Διατριβής & Συνεισφορές αυτής

Ενώ η αποδοτικότητα των πόρων των κέντρων δεδομένων μπορεί να βελτιωθεί μέσω διαφόρων στοιχείων της υποδομής υλικού και λογισμικού, στην παρούσα διατριβή δίνουμε ι-

[2]Google uses DeepMind AI to cut data center PUE by 15%

διαίτερη έμφαση στη βελτιστοποίηση της διαχείρισης πόρων και του χρονοπρογραμματισμού εφαρμογών σε συστήματα Νέφους. Δεδομένης της ισχυρής σχέσης επιδόσεων μεταξύ υλικού και λογισμικού, βασίζουμε τα μοντέλα βαθιάς μάθησης σε μετρητές επιδόσεων προσανατολισμένους στο υλικό, στοχεύοντας έτσι στη προβολή γεγονότων χαμηλού επιπέδου σε μετρήσεις υψηλότερου επιπέδου που μας ενδιαφέρουν. Συγκεκριμένα, οι βασικές αρχές που καθοδήγησαν τις ερευνητικές μας δραστηριότητες είναι οι ακόλουθες:

1. Δεδομένης της στενής σχέσης μεταξύ υλικού και λογισμικού, τα σύγχρονα συστήματα Υπολογιστικού Νέφους θα πρέπει να παίρνουν αποφάσεις ενορχήστρωσης πόρων λαμβάνοντας υπόψη τη δυναμική τόσο των εφαρμογών αυτών καθαυτών όσο και του υποκείμενου συστήματος. Προς αυτή την κατεύθυνση, πιστεύουμε ότι οι μετρητές επιδόσεων χαμηλού επιπέδου μπορούν να παρέχουν εξαιρετικά χρήσιμες πληροφορίες σχετικά με τα σημεία συμφόρησης/θορύβου των συστημάτων Υπολογιστικού Νέφους.

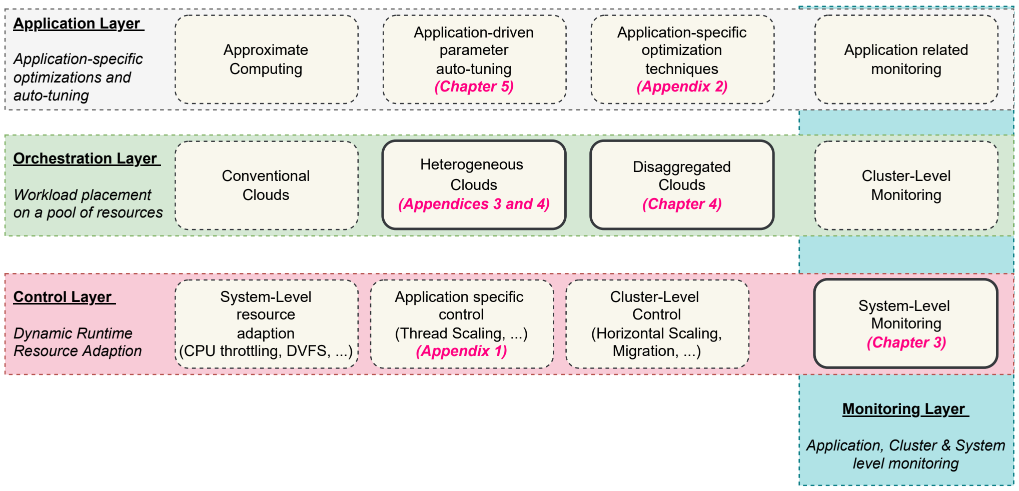
2. Υποστηρίζουμε ότι η πολυπλοκότητα των σύγχρονων συστημάτων Υπολογιστικού Νέφους είναι τεράστια, διαμορφώνοντας έτσι αφελείς λύσεις χρονοπρογραμματισμού/παρακολούθησης ανίκανες να χειριστούν αποτελεσματικά αυτόν τον κατακλυσμό διαθέσιμων δεδομένων και κουμπιών βελτιστοποίησης. Προς αυτή την κατεύθυνση, εξετάζουμε την αποτελεσματικότητα της μηχανικής μάθησης και της τεχνητής νοημοσύνης σε διάφορες πτυχές και επίπεδα βελτιστοποίησης του τομέα του νέφους.

Εξετάζουμε και τις δύο αυτές αρχές σε διάφορα προβλήματα στον τομέα του υπολογιστικού νέφους, που κυμαίνονται από βελτιστοποιήσεις συγκεκριμένων εφαρμογών έως βελτιστοποιήσεις σε επίπεδο συστήματος. Το Σχήμα 2 παρουσιάζει, με αφηρημένο τρόπο, μια υψηλού επιπέδου επισκόπηση της τοποθέτησης των συνεισφορών της παρούσας διατριβής σε σχέση με τα διάφορα επίπεδα βελτιστοποίησης.

Συνοπτικά, οι κύριες συνεισφορές της παρούσας διατριβής είναι τρεις:

- **Rusty:** Μια νέα εξελιγμένη λύση παρακολούθησης για υποδομές Υπολογιστικού Νέφους. Το Rusty αξιοποιεί τα δίκτυα μακράς βραχυπρόθεσμης μνήμης (LSTM) για να επιτρέψει γρήγορες και ακριβείς προβλέψεις κατανάλωσης πόρων και ενέργειας των συστημάτων υπό την παρουσία παρεμβολών λόγω θορύβου μεταξύ εφαρμογών. Μέσω του Rusty, η φιλοδοξία μας είναι να καθιερώσουμε την προγνωστική παρακολούθηση ως την μελλοντική λύση για την παρακολούθηση του νέφους, με στόχο την υιοθέτησή της για την καθοδήγηση προορατικών μηχανισμών κατανάλωσης πόρων.

- **Adrias:** Ένα πλαίσιο παρακολούθησης και ενορχήστρωσης για συστήματα με διαχωρισμένη μνήμη. Το Adrias παρακολουθεί συνεχώς το υποκείμενο σύστημα και συγκεντρώνει συμβάντα απόδοσης σε επίπεδο εφαρμογής και συστήματος. Αξιοποιώντας προσεγγίσεις βαθιάς μηχανικής μάθησης, το Adrias χρησιμοποιεί τις πληρο-


Application Layer
Application-specific optimizations and auto-tuning
Approximate Computing
Application-driven parameter auto-tuning (Chapter 5)
Application-specific optimization techniques (Appendix 2)
Application related monitoring
Orchestration Layer
Workload placement on a pool of resources
Conventional Clouds
Heterogeneous Clouds (Appendices 3 and 4)
Disaggregated Clouds (Chapter 4)
Cluster-Level Monitoring
Control Layer
Dynamic Runtime Resource Adaption
System-Level resource adaption (CPU throttling, DVFS, ...)
Application specific control (Thread Scaling, ...) (Appendix 1)
Cluster-Level Control (Horizontal Scaling, Migration, ...)
System-Level Monitoring (Chapter 3)
Monitoring Layer
Application, Cluster & System level monitoring

Σχήμα 2.. Σχηματική αναπαράσταση των συνεισφορών της παρούσας διδακτορικής διατριβής

φορίες παρακολούθησης για τη δυναμική τοποθέτηση νέων εφαρμογών σε συστήματα Υπολογιστικού Νέφους με αποσυντεθειμένη μνήμη.

- **Sparkle:** Ένα πλαίσιο αυτόματης ρύθμισης παραμέτρων με βάση τη βαθιά μάθηση για εφαρμογές Spark. Το Sparkle αξιοποιεί μια υβριδική αρχιτεκτονική βαθιού νευρωνικού δικτύου μαζί με γεγονότα παρακολούθησης επιδόσεων χαμηλού επιπέδου και επεκτείνεται σε ολόκληρο το χώρο διαμόρφωσης παραμέτρων του Spark, εξαλείφοντας έτσι πλήρως την ανάγκη για ανθρώπινη διαχείριση ή/και στατιστική ανάλυση προκειμένου να προσδιορίσουμε τις βέλτιστες παραμέτρους για εφαρμογές Spark. Χρησιμοποιώντας μια προσέγγιση γενετικής βελτιστοποίησης, το Σπαρκλε διατρέχει γρήγορα το χώρο σχεδιασμού παραμέτρων και εντοπίζει βελτιστοποιημένες διαμορφώσεις Spark.

3. Πλαίσιο Rusty

3.1. Εισαγωγή

Τα τελευταία χρόνια, ο αριθμός των εφαρμογών που εκτελούνται στο Υπολογιστικό Νέφος αυξήθηκε ραγδαία και αναμένεται να αυξηθεί περισσότερο στο μέλλον [272]. Η άνοδος των πλατφορμών για εκτέλεση εφαρμογών στο Νέφος (π.χ., Kubernetes [175]) που διευκολύνουν την εκτέλεση εφαρμογών σε κιβώτια και επεκτείνουν την ικανότητά τους να κλιμακώνουν δυναμικά τους πόρους, αυξάνει περαιτέρω την πυκνότητα των σύγχρονων συστημάτων νέφους. Επιπλέον, οι τρέχουσες λύσεις Υπολογιστικού Νέφους, όπως το Amazon AWS [72], το Google Cloud [178], το Microsoft Azure [177] και άλλα, παρέχουν στους χρήστες ελαστικότητα και δυνατότητα αλλαγής μεγέθους της υπολογιστικής τους χωρητικότητας, οδηγώντας σε δυναμική παροχή πόρων. Αυτή η αύξηση της πυκνότητας και της δυναμικής των φόρτων εργασίας στο νέφος συνεπάγεται ότι οι πάροχοι υπηρεσιών Υπολογιστικού Νέφους θα πρέπει να εφαρμόζουν προηγμένες τεχνικές κατανομής πόρων, ώστε να παρέχουν καλύτερη ποιότητα υπηρεσιών στους χρήστες τους, καθώς και να μεγιστοποιούν το κέρδος τους. Ωστόσο, αυτός ο στόχος βελτιστοποίησης δύο παραγόντων είναι εν γένει δύσκολα επιτεύξιμος, δεδομένου ότι η μεγιστοποίηση της απόδοσης απαιτεί την εκτέλεση των εφαρμογών σε απομονωμένα περιβάλλοντα, ενώ η αύξηση του κέρδους επιτυγχάνεται μέσω της συντοποθέτησης εφαρμογών σε κοινούς υπολογιστικούς πόρους, η οποία όμως οδηγεί σε συμφόρηση και θόρυβο στους διαμοιρασμένους υπολογιστικούς πόρους ενός συστήματος.

Για να μπορέσουμε να κατανοήσουμε καλύτερα τα πραγματικά σημεία συμφόρησης ενός συστήματος και να προσδιορίσουμε τη βασική αιτία της υποβάθμισης των επιδόσεων, θα πρέπει να εξετάσουμε πιο προσεκτικά αρχιτεκτονικά γεγονότα χαμηλότερου επιπέδου και να μπορούμε να αναλύσουμε συμβάντα σε επίπεδο συστήματος [274]. Για την επίτευξη των παραπάνω στόχων, η παρακολούθηση και η ανάλυση των σημάτων του συστήματος από το εσωτερικό του κέντρου δεδομένων έχει αποδειχθεί ότι είναι πραγματικά επωφελής και διορατική. Για παράδειγμα, η Alibaba και η Google παρέχουν πραγματικά ίχνη από αρχιτεκτονικά γεγονότα χαμηλού επιπέδου [29] από τα συστήματα συστάδων τους και ενθαρρύνουν τους ερευνητές και τους επαγγελματίες να τα επεξεργαστούν και να τα αναλύσουν. Επιπλέον, η Google έχει εντοπίσει ότι η παρακολούθηση των μετρητών επιδόσεων χαμηλού επιπέδου μπορεί να οδηγήσει σε καλύτερες αποφάσεις διαχείρισης πόρων και χρονοπρογραμματισμού [74].

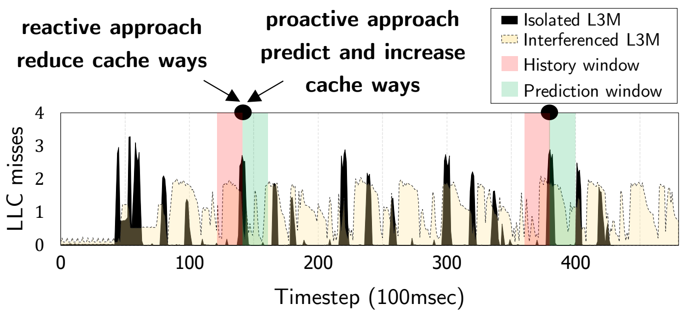
Προκειμένου να αξιοποιήσουμε αυτά τα χαρακτηριστικά χαμηλού επιπέδου, απαιτείται η παρακολούθηση εφαρμογών κατά την διάρκεια εκτέλεσής τους. Έτσι, η πληροφορία αυτή μπορεί να χρησιμοποιηθεί προκειμένου να τροφοδοτεί και καθοδηγεί αλγορίθμους διαχείρισης πόρων ή/και χρονοπρογραμματισμού, οι οποίοι πρέπει να είναι σε θέση να προβλέπουν δυναμικά τις ανάγκες σε πόρους ανά εφαρμογή, ώστε να επιβάλλουν βελτιστοποιημένες αποφάσεις σε σε πραγματικό χρόνο. Παλαιότερες ερευνητικές δραστηριότητες έχουν υποδείξει ότι οι εφαρμογές βιώνουν διαφορετικές φάσεις κατά τη διάρκεια της εκτέλεσής τους [161, 162, 208], οι οποίες οδηγούν σε "μη κανονικές" συμπεριφορές όσον αφορά τα πρότυπα πρόσβασης στη μνήμη και τη χρήση της CPU. Οι συμπεριφορές αυτές γίνονται ακόμη πιο πολύπλοκες αν λάβουμε υπόψη τις παρεμβολές που προκαλούνται λόγω συντοποθέτησης εφαρμογών σε κοινούς διακομιστές. Ως εκ τούτου, η λεπτομερής και ακριβής προβλεψιμότητα κατά τη διάρκεια εκτέλεσης των εφαρμογών είναι ζωτικής σημασίας για την λήψη αποφάσεων διαχείρισης πόρων σε πραγματικό χρόνο.

Από τα παραπάνω, είναι προφανές πως απαιτούνται νέες εξελιγμένες λύσεις παρακολούθησης για τον αποτελεσματικό χειρισμό του αναδυόμενου πεδίου των εφαρμογών σε συστήματα Υπολογιστικού Νέφους. Αξιοποιώντας τις δυνατότητες της υποδομής υλικού, η παρακολούθηση εφαρμογών και υποδομών θα πρέπει να στραφεί προς την παροχή ι) ταχύτερης παρατηρησιμότητας, ώστε να γίνεται αντιληπτή η ακραία ποικιλομορφία και ο δυναμισμός στα μεταβλητά φορτία εργασίας, και ιι) συνεχούς χρόνου εκτέλεσης και προβλεψιμότητας με επίγνωση των παρεμβολών, προκειμένου να λαμβάνονται αποφάσεις κατανομής πόρων με πιο τεκμηριωμένο τρόπο.

Στοχεύοντας στο να λύσουμε τα παραπάνω προβλήματα, σε αυτό το κεφάλαιο, παρουσιάζουμε το Rusty, ένα πλαίσιο παρακολούθησης που αξιοποιεί τα δίκτυα μακράς βραχυπρόθεσμης μνήμης (LSTM), επιτρέποντας έτσι γρήγορες και ακριβείς προβλέψεις κατανάλωσης πόρων και ενέργειας ενός συστήματος υπό την επήρεια παρεμβολών. Το Rusty αποτελεί το πρώτο σύστημα *μη διακοπτόμενης προγνωστικής παρακολούθησης* που είναι σε θέση να προβλέπει μετρητές επιδόσεων χαμηλού επιπέδου ενός συστήματος υπό παρεμβολή. Συγκεκριμένα, μέσω του Rusty, είμαστε σε θέση να προβλέψουμε τις εκτελούμενες εντολές ανά κύκλο (IPC) καθώς και τις αστοχίες σε δεδομένα στην Κρυφή Μνήμη Τελευταίου Επιπέδου (Last-Level Cache misses) των εφαρμογών που εκτελούνται ταυτόχρονα σε ένα πολυπύρηνο σύστημα και, επίσης, την κατανάλωση ενέργειας αυτού. Σε αντίθεση με τις προηγούμενες προσεγγίσεις που βασίζονται σε συστηματική αξιολόγηση του συστήματος ανά τακτά χρονικά διαστήματα σε πραγματικό χρόνο για τη μοντελοποίηση παρεμβολών [41, 62, 208], το Rusty χρησιμοποιεί την ικανότητα πρόβλεψης των LSTM και επιτρέπει συνεχείς προβλέψεις των μετρικών απόδοσης με επίγνωση της παρεμβολής.

3.2. Σημαντικότητα Προγνωστικής Παρακολούθησης

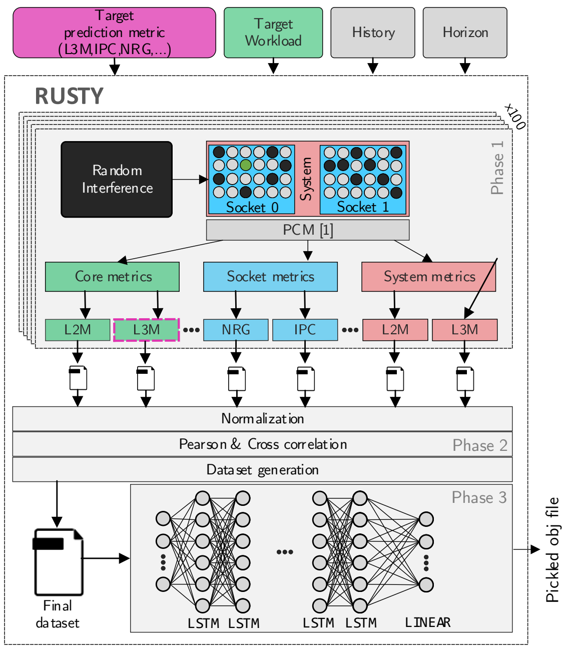
Όπως αναφέραμε προηγουμένως, το πλαίσιο Rusty αποτελεί ένα σύστημα *προγνωστικής παρακολούθησης*, δηλαδή χρησιμοποιείται προκειμένου να μπορούμε να προβλέπουμε μετρικές



Σχήμα 3.. Πραγματικό αποτύπωμα των αστοχιών για πρόσβαση σε δεδομένα της κρυφής μνήμης για μία εφαρμογή τεχνητής νοημοσύνης.

απόδοσης του συστήματος, υπό την παρουσία θορύβου, σε πραγματικό χρόνο. Γιατί όμως είναι σημαντική μία τέτοια λειτουργία για έναν ενορχηστρωτή πόρων σε συστήματα Υπολογιστικού Νέφους; Προκειμένου να απαντήσουμε σε αυτήν την ερώτηση, εκτελέσαμε μία τυπική εφαρμογή τεχνητής νοημοσύνης και κατά τη διάρκεια εκτέλεσής της παρακολουθήσαμε και συλλέξαμε τις μετρικές απόδοσης του συστήματός μας.

Στο Σχήμα 3 μπορούμε να δούμε το αποτύπωμα της εφαρμογής όσον αφορά τις αστοχίες για πρόσβαση σε δεδομένα στην κρυφή μνήμη του συστήματος. Όπως φαίνεται και από το σχήμα, η εφαρμογή περνάει από διαφορετικές φάσεις κατά τη διάρκεια εκτέλεσής της. Συγκεκριμένα, παρατηρούμε φάσεις τις εφαρμογές όπου οι αστοχίες της εφαρμογής κινούνται σε πολύ υψηλά επίπεδα (“κορυφές”) και άλλες όπου ο αριθμός αστοχιών είναι σχεδόν μηδενικός (“κοιλάδες”). Ένας ενορχηστρωτής πόρων θα πρέπει να είναι σε θέση να μπορεί να προβλέψει αυτές τις εναλλαγές μεταξύ “κορυφών” και “κοιλάδων” και να τροποποιεί/κατανέμει τους πόρους του συστήματος αναλόγως. Ας πάρουμε ως παράδειγμα το σημείο της εκτέλεσης που έχουμε επισημάνει με την μαύρη κουκκίδα και έστω πως θέλουμε να ρυθμίσουμε το μέγεθος της κρυφής μνήμης το οποίο διαθέτουμε για την συγκεκριμένη εφαρμογή. Ένας ενορχηστρωτής πόρων ο οποίος δεν λειτουργεί με προβλεπτικό τρόπο (αντιδραστικός) θα διάβαζε τις μετρικές συστήματος αφότου είχε επέλθει η “κορυφή” των αστοχιών και έτσι θα αύξανε (λανθασμένα) το μέγεθος της κρυφής μνήμης που θα διέθετε για την εφαρμογή, ενώ αυτή θα είχε μπει σε φάση με χαμηλές αστοχίες. Αντιθέτως, ένας ενορχηστρωτής ο οποίος λειτουργεί με προβλεπτικό τρόπο, θα μπορούσε να εκτιμήσει την αύξηση των αστοχιών πριν αυτές προκύψουν και έτσι να αυξήσει προληπτικά το μέγεθος κρυφής μνήμης που θα διαθέσει στην εφαρμογή, πιθανότατα αυξάνοντας έτσι την συνολική απόδοσή της.



Σχήμα 4.. Επισκόπηση Αρχιτεκτονικής του πλαισίου Rusty

3.3. Περιγραφή Αρχιτεκτονικής του Rusty

Η σχεδίαση της αρχιτεκτονικής του ενγΡυστψ αποτελείται από δύο φάσεις, την φάση συλλογής δεδομένων για την εκπαίδευση των νευρωνικών δικτύων καθώς και την εκπαίδευση αυτών, και τη φάση όπου τα δίκτυα αυτά χρησιμοποιούνται κατά την εκτέλεση των εφαρμογών προκειμένου να πραγματοποιήσουν προβλέψεις των μετρικών του συστήματος σε πραγματικό χρόνο. Παρακάτω αναλύουμε εν συντομία τις εργασίες που πραγματοποιούνται σε κάθε μία από αυτές τις φάσεις.

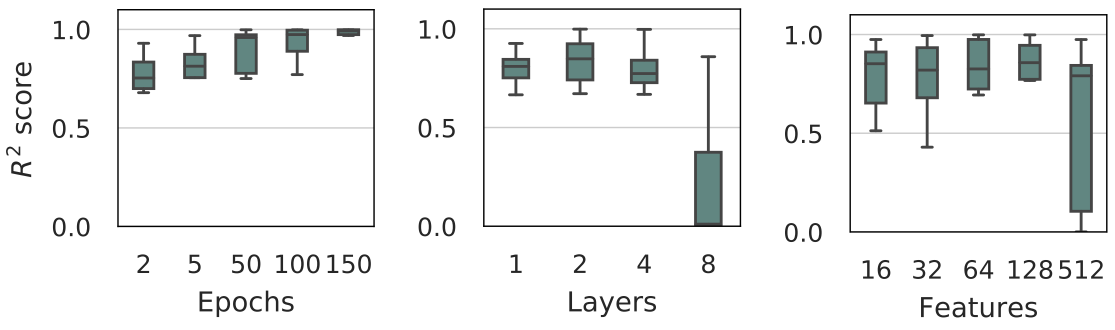
Φάση Συλλογής Δεδομένων και Εκπαίδευσης Μοντέλων

Στο Σχήμα 4 φαίνεται η αρχιτεκτονική του πλαισίου Rusty. Όπως φαίνεται και από το σχήμα, το πλαίσιο λαμβάνει ως είσοδο τέσσερις παραμέτρους: i) την εφαρμογή για την οποία θέλουμε να προβλέψουμε μετρικές απόδοσης χαμηλού επιπέδου, ii) την μετρική απόδοσης την οποία θέλουμε να προβλέψουμε, η οποία μπορεί να είναι οποιαδήποτε μετρική από το εργαλείο PCM [285] (για την τρέχουσα εργασία εστιάζουμε στις IPC, L3M και NRG), iii) μία μεταβλητή την οποία καλούμε "History" και iv) μία μεταβλητή την οποία καλούμε

"Horizon". Η μεταβλητή *History* αναφέρεται στον αριθμό των δειγμάτων που προέρχονται από το εργαλείο PCM και χρησιμοποιούνται ως ακολουθία εισόδου στο μοντέλο LSTM, ενώ η μεταβλητή *horizon* αναφέρεται στον αριθμό των χαρακτηριστικών εξόδου/μελλοντικών τιμών που θα προβλέψει το μοντέλο LSTM.

Συλλογή δεδομένων υπό την επήρεια παρεμβολών: Σαν πρώτο βήμα, το Rusty συλλέγει δεδομένα από τις εφαρμογές υπό την ύπαρξη θορύβου και παρεμβολών στο σύστημα. Για να επιτευχθεί αυτό, εκτελούμε την εφαρμογή στο σύστημά μας 100 φορές, κάθε φορά με διαφορετικό φορτίο παρεμβολής, το οποίο μπορεί να είναι είτε πραγματικό είτε συνθετικό (π.χ. εφαρμογές οι οποίες προκαλούν θόρυβο στο σύστημα μας). Στα πλαίσια αυτής της εργασίας, προκαλούμε παρεμβολές στο σύστημά μας χρησιμοποιώντας τη σουίτα iBench [3], η οποία παρέχει διάφορες εφαρμογές, όπου η κάθε μία καθένα στοχεύει στο να προκαλεί θόρυβο σε διαφορετικούς πόρους σε πολυπύρηνο συστήματα (π.χ. CPU, κρυφή και κύρια μνήμη κ.ά.). Συγκεκριμένα, σηκώνουμε στο σύστημά μας τυχαίο αριθμό από διεργασίες από τη σουίτα iBench, όπου η κάθε διεργασία ανατίθεται σε έναν τυχαία επιλεγμένο πυρήνα του συστήματος. Κατά τη διάρκεια της εκτέλεσης, το πλαίσιο Rusty συλλέγει και αποθηκεύει όλες τις πληροφορίες σχετικά με τις μετρικές απόδοσης του συστήματος, μέσω του εργαλείου PCM. Ακολουθώντας την παραπάνω διαδικασία, το iBench παράγει στατικές παρεμβολές για όλη τη διάρκεια ζωής του φόρτου εργασίας. Ωστόσο, εξετάζοντας πολλαπλά σενάρια συν-εκτέλεσης των εφαρμογών iBench με την εφαρμογή την οποία εξετάζουμε, παρατηρούμε ότι τα αποτελέσματα των παρεμβολών ανά σενάριο μπορούν να αλλάξουν δραματικά, παράγοντας έτσι ανόμοιες επιπτώσεις στην απόδοση της εφαρμογής που εξετάζουμε.

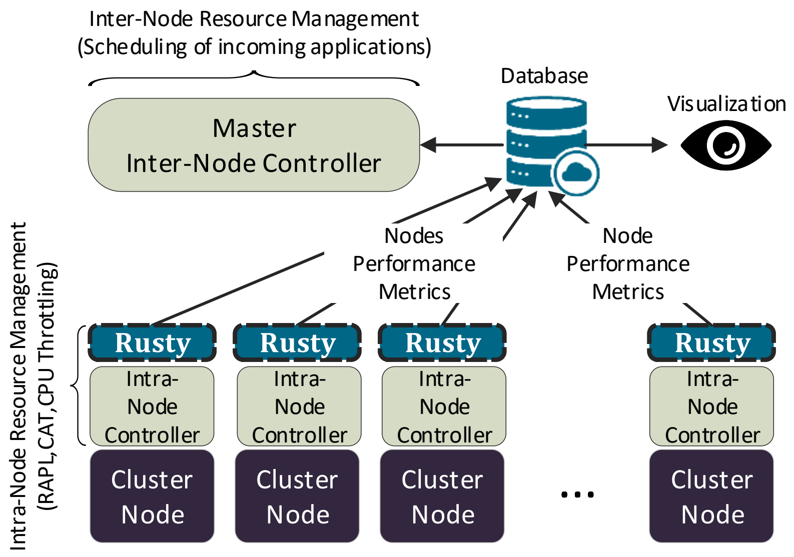
Προεπεξεργασία και δημιουργία συνόλου δεδομένων για εκπαίδευση των μοντέλων: Το αποτέλεσμα της φάσης 1 είναι ένα σύνολο από χρονοσειρές των μετρικών του συστήματος για τα 100 διαφορετικά σενάρια εκτέλεσης υπό διαφορετικά επίπεδα παρεμβολών. Στην δεύτερη φάση, το Rusty πραγματοποιεί μία προεπεξεργασία των δεδομένων αυτών προκειμένου αυτά να τα φέρουμε σε μορφή που να μπορούν να δοθούν ως είσοδο για να εκπαιδεύσουμε το νευρωνικό μας δίκτυο. Συγκεκριμένα, πρώτον, κανονικοποιούμε τις τιμές των μετρικών τις οποίες συλλέγουμε προκειμένου αυτές να κυμαίνονται στο εύρος [0,1]. Αυτή η διαδικασία είναι απαραίτητη, καθώς οι τιμές μεταξύ διαφορετικών μετρικών μπορεί να έχουν διαφορετική κλίμακα μεγέθους και βοηθά στην μετέπειτα εκπαίδευση των νευρωνικών δικτύων. Επίσης, προκειμένου να ελαχιστοποιήσουμε τις μετρικές που δίνουμε ως είσοδο στο μοντέλο (και έτσι την πολυπλοκότητά του), υπολογίζουμε δύο επιπλέον μετρικές, συγκεκριμένα την "συσχέτιση κατά Pearson" καθώς και την "Διασυσχέτιση". Η "συσχέτιση κατά Pearson" μας δείχνει κατά πόσο η μετρική την οποία θέλουμε να προβλέψουμε σχετίζεται με όλες τις υπόλοιπες μετρικές τις οποίες συλλέγουμε από το σύστημά μας, ενώ η "Διασυσχέτιση" μεταξύ 2 μετρικών μας δείχνει κατά πόσο οι χρονοσειρές των διαφορετικών μετρικών σχετίζονται κατά την πάροδο του χρόνου. Τέλος, προκειμένου να δημιουργήσουμε το σύνολο δεδομένων για την εκπαίδευση των μοντέλων μας, σπάμε τις



Σχήμα 5.. Διερεύνηση χώρου παραμέτρων των δικτύων LSTM

χρονοσειρές των μετρικών συστήματος τις οποίες συλλέξαμε προηγουμένως σε μικρότερα κομμάτια με τη χρήση ενός συρόμενου παραθύρου. Το παράθυρο αυτό έχει μήκος ίσο με το άθροισμα των τιμών των μεταβλητών "History" και "Horizon" που αναφέραμε παραπάνω και ουσιαστικά αντιστοιχεί στην περίοδο με την οποία θα πραγματοποιούσαμε προβλέψεις στο σύστημά μας σε ένα σενάριο πραγματικού χρόνου.

Διερεύνηση χώρου παραμέτρων LSTM και εκπαίδευση των μοντέλων Ως τελευταίο βήμα, το πλαίσιο Rusty πραγματοποιεί μία διερεύνηση σε διαφορετικές παραμέτρους η οποίες επηρεάζουν την απόδοση των μοντέλων LSTM. Αυτό το στάδιο έχεις ως στόχο να μειώσουμε όσο το δυνατόν περισσότερο την πολυπλοκότητα του μοντέλου μας, προκειμένου αυτό να είναι ελαφρύ και έτσι να μπορεί να πραγματοποιεί προβλέψεις γρήγορα σε πραγματικό χρόνο καθώς και να μην επιβαρύνει το σύστημα μας με περαιτέρω παρεμβολές. Συγκεκριμένα, εξετάζουμε τρεις διαφορετικές παραμέτρους, i) Τον αριθμό των εποχών για τις οποίες εκπαιδεύουμε το δίκτυό μας, ii) Τον αριθμό των επιπέδων του δικτύου μας και iii) Τον αριθμό των χαρακτηριστικών παραμέτρων σε κάθε επίπεδο. Στο Σχήμα 5 βλέπουμε την επίδραση που έχει η αλλαγή κάθε μίας από τις παραπάνω παραμέτρους στην απόδοση του μοντέλου μας. Όπως φαίνεται και από το σχήμα, όσο αυξάνουμε τον αριθμό των επιπέδων καθώς και τις παραμέτρους ανά επίπεδο, τόσο αυξάνεται και η απόδοση του μοντέλου μας, μέχρι ένα συγκεκριμένο πλατό (4 επίπεδα και 128 παράμετροι ανά επίπεδο). Επίσης, παρατηρούμε πως αυξάνοντας τις εποχές για τις οποίες εκπαιδεύουμε το μοντέλο μας, οδηγούμαστε σε καλύτερες τιμές απόδοσης, με τις 150 εποχές να παράγουν τα πιο υψηλά επίπεδα ακρίβειας.


Inter-Node Resource Management
(Scheduling of incoming applications)
Master
Inter-Node Controller
Database
Visualization
Nodes Performance Metrics
Node Performance Metrics
Intra-Node Resource Management
(RAPL,CAT,CPU Throttling)
Rusty
Intra-Node Controller
Cluster Node
...

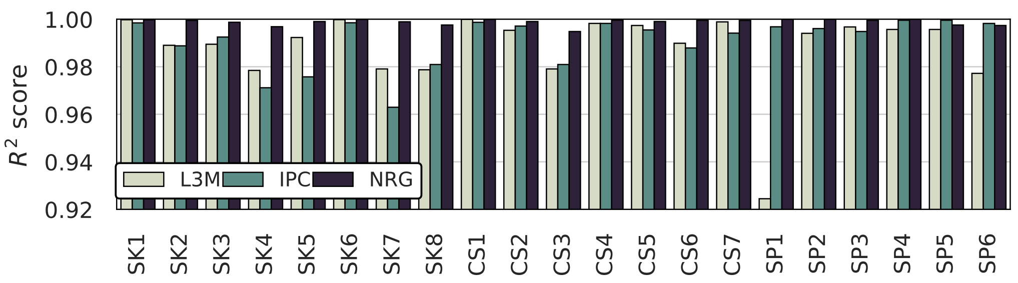
Σχήμα 6.. Παράδειγμα ανάπτυξης και λειτουργίας του πλαισίου Rusty σε ένα σύστημα αποτελούμενο από συστάδες διακομιστών.

Φάση Λειτουργίας και Προβλέψεων σε Πραγματικό Χρόνο

Στο Σχήμα 6 φαίνεται πως το πλαίσιο Rusty μπορεί να χρησιμοποιηθεί σε ένα σύστημα που αποτελείται από συστάδες διακομιστών, όπου το σύστημα αυτό βρίσκεται υπό την διαχείριση ενός κεντρικού ενορχηστρωτή πόρων (π.χ., Kubernetes). Σε μία τέτοια εγκατάσταση, θα υπήρχε ένα στιγμιότυπο του Rusty σε κάθε έναν από τους διακομιστές της συστάδας, το οποίο θα ήταν υπεύθυνο να παρακολουθεί καθώς και να προβλέπει τις μετρικές απόδοσης χαμηλού επιπέδου του συστήματος. Αυτές οι μετρικές θα μπορούσαν να χρησιμοποιηθούν τόσο από κάποιον ενορχηστρωτή πόρων εντός του ίδιου διακομιστή, προκειμένου να μπορεί να ρυθμίζει και να κατανέμει τους διαθέσιμους υπολογιστικούς πόρους μεταξύ των εφαρμογών βασισμένος στις προβλέψεις του Rusty, είτε από κάποιον κεντρικό διαχειριστή, ο οποίος έχει ως σκοπό να πραγματοποιήσει την αρχική τοποθέτηση και να δρομολογήσει τις εφαρμογές στους διαφορετικούς διακομιστές της συστάδας.

3.4. Πειραματική Αξιολόγηση

Σε αυτήν την ενότητα δείχνουμε κάποια ενδεικτικά αποτελέσματα σχετικά με την αποδοτικότητα και αποτελεσματικότητα του πλαισίου Rusty. Συγκεκριμένα αυτό που εξετάζουμε είναι το κατά πόσο μπορεί το πλαίσιο Rusty να πραγματοποιήσει ακριβείς προβλέψεις σχετι-



Σχήμα 7.. R^2 σκορ που πέτυχε το πλαίσιο Rusty για πρόβλεψη των μετρικών L3M χρησιμοποιώντας αρχιτεκτονική με 4 επίπεδα και 128 παραμέτρους ανά επίπεδο, για παράμετρο *history=50* και *horizon=1.*

κά με τις 3 διαφορετικές μετρικές συστήματος που εξετάζουμε, τόσο για γνωστές όσο και για άγνωστες εφαρμογές αλλά και για άγνωστους τύπους διακομιστών. Προκειμένου να αξιολογήσουμε το προτεινόμενο πλαίσιο, χρησιμοποιήσαμε εφαρμογές από τρεις διαφορετικές σουίτες [83, 283, 284], ενώ τα πειράματά μας διεξήχθησαν σε ένα ισχυρό διακομιστή που συναντάμε συχνά σε περιβάλλοντα Υπολογιστικού Νέφους (Intel©Xeon©E5-2658A v3).

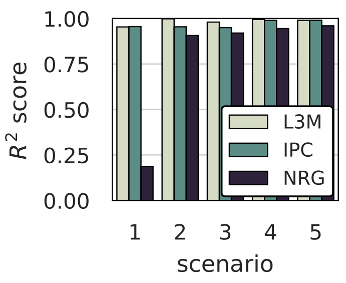
Ακρίβεια προβλέψεων για γνωστές εφαρμογές

Αρχικά, αξιολογούμε την ακρίβεια του προτεινόμενου πλαισίου όσον αφορά την πρόβλεψη μετρικών απόδοσης για γνωστές εφαρμογές, δηλαδή εφαρμογές τις οποίες τις χρησιμοποιήσαμε και κατά τη φάση συλλογής δεδομένων και εκπαίδευσης των μοντέλων μας. Το Σχήμα 7 δείχνει τα αποτελέσματα που πετύχαμε για κάθε εφαρμογή και κάθε μία από τις 3 μετρικές απόδοσης, όπου ως μέτρο αξιολόγησης χρησιμοποιήσαμε το σκορ R^2. Παρατηρούμε πως το πλαίσιο Rusty επιτυγχάνει πολύ υψηλές τιμές R^2, οι οποίες κυμαίνονται από 0.92 έως 0.99 σκορ, με κατά μέσο όρο 0.991 0.988 και 0.994 σκορ για πρόβλεψη των μετρικών L3M, IPC και NRG αντίστοιχα.

Ακρίβεια προβλέψεων για άγνωστες εφαρμογές

Στη συνέχεια, αξιολογούμε την ακρίβεια των προβλεπτικών μοντέλων του Rusty όσον αφορά άγνωστες εφαρμογές. Για το σκοπό αυτό, χρησιμοποιούμε διαφορετικές εφαρμογές προκειμένου να εκπαιδεύσουμε τα μοντέλα πρόβλεψης των μετρικών απόδοσης και διαφορετικές εφαρμογές προκειμένου να αξιολογήσουμε την επίδοσή τους.

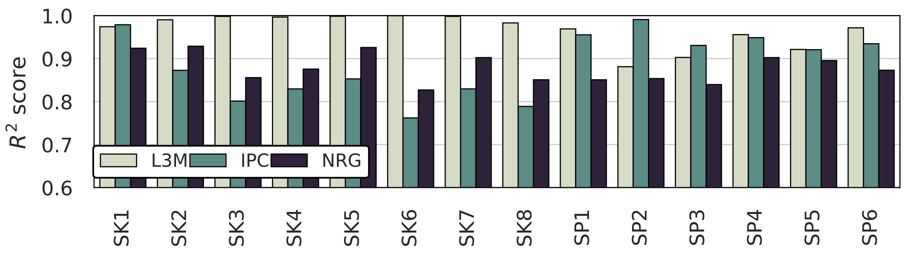
Στο Σχήμα 8 παρατηρούμε τα σχετικά αποτελέσματα, όπου ο άξονας y αναφέρεται και πάλι στην μετρική απόδοσης R^2 ενώ ο άξονας x δείχνει 4 διαφορετικά σενάρια, τα οποία αντιστοιχούν σε διαφορετικούς συνδυασμούς εφαρμογών τις οποίες χρησιμοποιούμε κατά τη διάρκεια εκπαίδευσης των νευρωνικών δικτύων μας και κατά τη διάρκεια αξιολόγησής τους. Όπως φαίνεται και από το σχήμα, στην πλειοψηφία των περιπτώσεων το πλαίσιο Rusty επιτυγχάνει πολύ υψηλά σκορ, με τις τιμές να κυμαίνονται μεταξύ 0.965 και 0.988 R^2. Όσον αφορά τη μετρική κατανάλωσης ενέργειας (NRG) παρατηρούμε πως ενώ για τα σενάρια 2-5, τα σκορ πρόβλεψης είναι αρκετά υψηλά, στο σενάριο 1 κυμαίνεται σε χαμηλές τιμές. Αυτό συμβαίνει διότι οι διακυμάνσεις καθώς και τα πρότυπα κατανάλωσης ενέργειας των εφαρμογών οι οποίες χρησιμοποιήθηκαν για την εκπαίδευση των μοντέλων δεν είναι αντιπροσωπευτικά των εφαρμογών οι οποίες χρησιμοποιούνται για την αξιολόγησή τους και, άρα, τα μοντέλα δεν είναι σε θέση να προβλέψουν επιτυχώς πρότυπα κατανάλωσης πάνω στα οποία δεν έχουν εκπαιδευτεί.



Σχήμα 8.. Απόδοση προβλέψεων για άγνωστες εφαρμογές

Ακρίβεια προβλέψεων για άγνωστους διακομιστές

Τέλος, στα πλαίσια αξιολόγησης, εξετάσαμε την συμπεριφορά του Rusty όταν αυτό λειτουργεί σε κάποιον άγνωστο διακομιστή. Για το σκοπό αυτό νοικιάσαμε έναν διακομιστή από έναν πραγματικό πάροχο υπηρεσιών Υπολογιστικού Νέφους και συγκεκριμένα από την Amazon AWS (`m5.metal server`), πάνω στον οποίο τρέξαμε 100 διαφορετικά σενάρια συντοποθέτησης εφαρμογών. Στη συνέχεια, εκπαιδεύσαμε τα μοντέλα μας χρησιμοποιώντας μετρικές απόδοσης από το αρχικό μας (τοπικό) σύστημα και τα αξιολογήσαμε χρησιμοποιώντας μετρικές απόδοσης από το σύστημα της Amazon.



Σχήμα 9.. Απόδοση προβλέψεων για άγνωστους διακομιστές

Στο Σχήμα 9 βλέπουμε τα αντίστοιχα αποτελέσματα. Όπως φαίνεται και από το σχήμα, το πλαίσιο Rusty και πάλι πετυχαίνει αρκετά υψηλά σκορ (αν και κάπως χαμηλότερα από προηγουμένως), με τις τιμές του R^2 να κυμαίνονται μεταξύ 0.76 έως 0.99. Συγκριτικά με τις αντίστοιχες τιμές για γνωστούς διακομιστές, παρατηρούμε μία μείωση της τάξης του 0.01, 0.10 και 0.12 για τις μετρικές L3M, IPC αι NRG αντίστοιχα. Αυτό το πείραμα αποκαλύπτει ότι, παρόλο που η υποκείμενη αρχιτεκτονική μπορεί να αλλάζει, υπάρχουν επαναλαμβανόμενα μοτίβα στα πρότυπα χρονοσειρών των εφαρμογών, τα οποία το πλαίσιο Rusty είναι σε θέση να μοντελοποιήσει, λόγω της κανονικοποίησης που εκτελείται κατά τη φάση προεπεξεργασίας δεδομένων.

3.5. Επίλογος

Η προβλεψιμότητα υπολογιστικών συστημάτων σε πραγματικό χρόνο υπό την ύπαρξη παρεμβολών είναι απαραίτητη για την καλύτερη διαχείριση των πόρων σε σύγχρονα ΚΔ μεγάλης κλίμακας. Σε αυτό το κεφάλαιο προτείναμε το Rusty, ένα ελαφρύ πλαίσιο προγνωστικής παρακολούθησης που βασίζεται σε δίκτυα LSTM, ικανό να παρέχει γρήγορες, ακριβείς, μη διακοπτόμενες προβλέψεις μετρήσεων συστήματος χαμηλού επιπέδου υπό την ύπαρξη θορύβου και παρεμβολών στο σύστημα. Αναλύσαμε και διερευνήσαμε πολλά διαφορετικά σχήματα αρχιτεκτονικές του δικτύου LSTM και καταλήξαμε σε μια γενική αποδοτική αρχιτεκτονική όσον αφορά την ακρίβεια, την απόκριση στους περιορισμούς χρόνου εκτέλεσης και το υπολογιστικό κόστος. Δείξαμε ότι το Rusty δημιουργεί μια ρεαλιστική λύση επιτυγχάνοντας εξαιρετικά υψηλή ακρίβεια πρόβλεψης R^2 0,98 κατά μέσο όρο κάτω από ρεαλιστικά σενάρια συντοποθέτησης εφαρμογών σε κοινούς διακομιστές.

4. Πλαίσιο Adrias

4.1. Εισαγωγή

Τα τελευταία χρόνια, ο υπολογισμός στο Υπολογιστικό Νέφος έχει καθιερωθεί ως το νέο πρότυπο για την ανάπτυξη εφαρμογών και αναμένεται να αναπτυχθεί ακόμη περισσότερο στο εγγύς μέλλον, λόγω της ευελιξίας και της οικονομικής αποδοτικότητας που προσφέρει [310]. Από την σκοπιά των παρόχων, η μεγιστοποίηση της απόδοσης που παρέχεται στους πελάτες με παράλληλη ελαχιστοποίηση του συνολικού κόστους ιδιοκτησίας αποτελεί τον πιο σημαντικό στόχο κατά τον σχεδιασμό των υποδομών τους. Ωστόσο, αυτές οι δύο αντικρουόμενες απαιτήσεις είναι δύσκολο να επιτευχθούν, καθώς η μεγιστοποίηση της απόδοσης απαιτεί μεμονωμένη εκτέλεση των εφαρμογών, η οποία, ωστόσο, οδηγεί σε υψηλή υποχρησιμοποίηση των υπολογιστικών πόρων στα ΚΔ [29, 30, 311].

Προκειμένου να αντιμετωπίσουν αυτό το πρόβλημα, οι πάροχοι υπηρεσιών Υπολογιστικού Νέφους έχουν υιοθετήσει στρατηγικές πολλαπλής μίσθωσης των διακομιστών τους σε διαφορετικούς χρήστες. Ενώ αυτή η προσέγγιση βελτιώνει τη συνολική χρήση πόρων των ΚΔ, οδηγεί επίσης σε παρεμβολές και θόρυβο στους κοινόχρηστους υπολογιστικούς πόρους, οι οποίοι με τη σειρά τους προκαλούν μεταβλητότητα και υποβάθμιση στην απόδοση των εφαρμογών [41, 88, 116, 313]. Για να περιοριστεί το παραπάνω πρόβλημα, τα τελευταία χρόνια έχουν αναπτυχθεί αρκετά πλαίσια τα οποία στοχεύουν στην έξυπνη ενορχήστρωση και διαχείριση υπολογιστικών πόρων τόσο από τον ακαδημαϊκό χώρο [41, 63, 64] όσο και από τη βιομηχανία [117], όπου σύνθετοι μηχανισμοί λογισμικού ελέγχουν τον τρόπο με τον οποίο οι διαθέσιμοι υπολογιστικοί πόροι κατανέμονται στις εφαρμογές που εκτελούνται στο σύστημα.

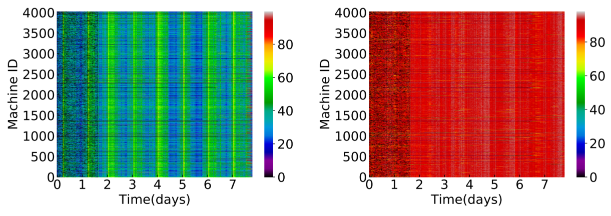
Παρά την πολυπλοκότητά τους, οι μηχανισμοί οι οποίοι στηρίζονται καθαρά σε υλοποιήσεις σε επίπεδο λογισμικού αποδεικνύονται μη ικανοί να επιλύσουν πλήρως το πρόβλημα υποχρησιμοποίησης πόρων σε συστήματα Νέφους, η οποία ουσιαστικά προκύπτει ως κατάληξη του συνδυασμού δύο διαφορετικών παραγόντων i) των διαφορετικών υπολογιστικών απαιτήσεων μεταξύ διαφορετικών εφαρμογών και ii) της σταθερής χωρητικότητας των σύγχρονων διακομιστών αναφορικά με τους διαθέσιμους υπολογιστικούς πόρους. Κατά συνέπεια, είναι σύνηθες στα σύγχρονα κέντρα δεδομένων να παρατηρείται ένας κατακερματισμός πόρων που είναι διαθέσιμοι αλλά δεν μπορούν να χρησιμοποιηθούν από καμία εφαρμογή [317, 318].

Προκειμένου να αντιμετωπιστεί το πρόβλημα του κατακερματισμού των υπολογιστικών πόρων, τελευταία έχει προταθεί ένα νέο πρότυπο σχεδίασης συστημάτων Υπολογιστικού Νέφους, γνωστό και ως "Αποσύνθεση Υπολογιστικών Πόρων". Στο πρότυπο αυτό, οι διαθέσιμοι υπολογιστικοί πόροι οργανώνονται σε ομογενείς ομάδες (π.χ., διακομιστές μόνο με κύρια μνήμη, διακομιστές μόνο με επιταχυντές, κτλ.), οι οποίοι μπορούν να συντεθούν δυναμικά, ανάλογα με τις υπολογιστικές απαιτήσεις των εφαρμογών οι οποίες τρέχουν στις υποδομές του Νέφους. Ως αποτέλεσμα, παρόλο που η "Αποσύνθεση Υπολογιστικών Πόρων" προσφέρει πιο λεπτομερή οργάνωση των υπολογιστικών πόρων, εισάγει επίσης και νέα σημεία τα οποία χρήζουν βελτιστοποίησης (π.χ. επιλογή μεταξύ τοπικής και απομακρυσμένης κατανομής μνήμης), τα οποία πρέπει να αντιμετωπίζονται σωστά για να παρέχουν αυξημένη αποδοτικότητα των υπολογιστικών πόρων. Ειδικά σε συστήματα νέφους με Αποσύνθεση Μνήμης, απαιτείται ενορχηστρωμένη πρόσβαση στους πόρους μνήμης, δεδομένου ότι η απόδοση των εφαρμογών μπορεί να επηρεαστεί σοβαρά λόγω της αυξημένης καθυστέρησης στην πρόσβαση στην απομακρυσμένη μνήμη [337], ενώ τα μοτίβα πρόσβασης στη μνήμη συχνά αποκαλύπτουν απρόβλεπτες διακυμάνσεις κατά τη διάρκεια της εκτέλεσης [162, 208]. Επιπλέον, πρόσφατες έρευνες έχουν δείξει πως το αποτύπωμα δυαδικού κώδικα των εφαρμογών που εκτελούνται στο Νέφος είναι μία έως δύο τάξεις μεγέθους μεγαλύτερο από τη κρυφή μνήμη εντολών L1 και μπορεί να οδηγήσει σε επαναλαμβανόμενες αστοχίες κατά την πρόσβαση σε δεδομένα, βλάπτοντας την απόδοση των εκτελούμενων εφαρμογών [288, 289], ενώ τελευταίες αναφορές από ΚΔ μεγάλης κλίμακας δείχνουν ότι η μνήμη αποτελεί το μεγαλύτερο σημείο συμφόρησης [29].

Στο κεφάλαιο αυτό, παρουσιάζουμε το πλαίσιο Adrias, έναν δρομολογητή εφαρμογών για συστήματα με αποσυντεθειμένους υπολογιστικούς πόρους μνήμης, τα οποία λειτουργούν κάτω από την παρουσία θορύβου λόγω συντοποθέτησης εφαρμογών. Τα βασικά χαρακτηριστικά του πλαισίου Adrias καθώς και οι συνεισφορές του μπορούν να συνοψιστούν στα παρακάτω σημεία: i) Την ικανότητά του να πραγματοποιεί προβλέψεις σχετικά με μετρικές απόδοσης χαμηλού επιπέδου, οδηγώντας έτσι προορατικές επιλογές αναφορικά με ενορχήστρωση πόρων σε συστήματα αποσυντεθειμένης μνήμης, ii) την ικανότητά του να μπορεί να προβλέψει την απόδοση εφαρμογών, πρωτού εκείνες εκτελεστούν, είτε στην τοπική είτε στην απομακρυσμένη μνήμη και υπό την επήρεια θορύβου και iii) την ικανότητά του να χρησιμοποιεί την απομακρυσμένη αποσυντεθειμένη μνήμη με ελάχιστη (ή και καθόλου) επιρροή στην απόδοση των εφαρμογών οι οποίες εκτελούνται επάνω στο σύστημα. Το πλαίσιο Adrias χρησιμοποιεί μετρικές απόδοσης χαμηλού επιπέδου τις οποίες τροφοδοτεί σε έναν ενορχηστρωτή πόρων ο οποίος βασίζεται σε βαθιά νευρωνικά δίκτυα πετυχαίνοντας έτσι να δρομολογεί εφαρμογές σε συστήματα αποσυντεθειμένης μνήμης με ελάχιστη επίπτωση στην απόδοσή τους.

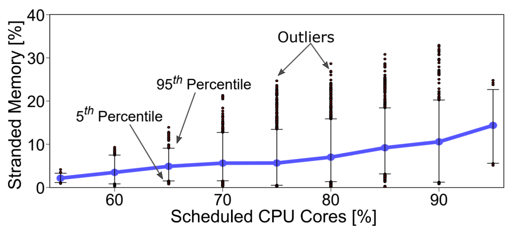
Σημαντικότητα πλαισίου Adrias

Όπως αναφέραμε και προηγουμένως, τα συστήματα αποσυντεθειμένων πόρων γίνονται ολοένα και πιο δημοφιλή στις υποδομές Υπολογιστικού Νέφους [233–235]. Πέραν των άλλων,



(a) CPU usage.　(b) Memory usage.

(α′) Alibaba [29]

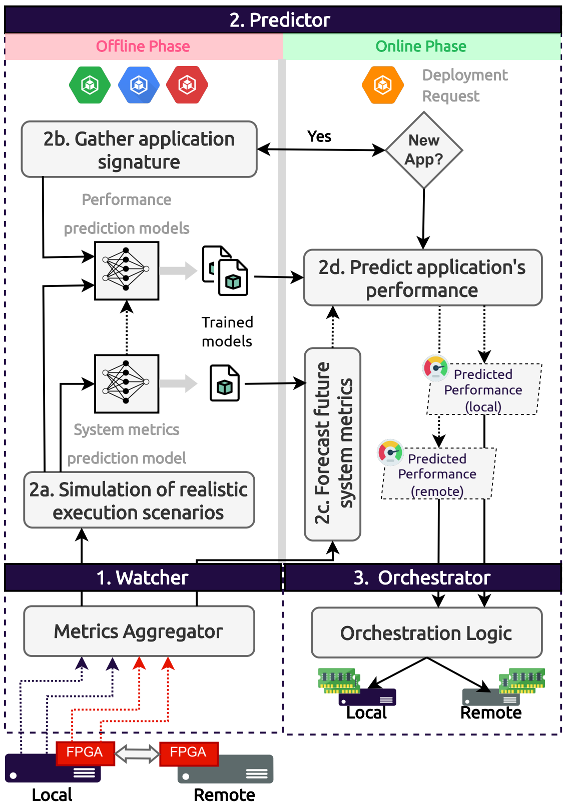


(β′) Azure [235]

Σχήμα 10.. Ίχνη σχετικά με την χρησιμοποίηση της CPU και της μνήμης στις υποδομές δημοφιλούς παρόχων Υπηρεσιών Νέφους

αυτό συμβαίνει λόγω του κατακερματισμού των πόρων, γεγονός το οποίο είναι συνέπεια τόσο των εξαιρετικά διαφορετικών απαιτήσεων των διαφορετικών εφαρμογών όσο και της μη βέλτιστης τοποθέτησης των εφαρμογών στους διαθέσιμους υπολογιστικούς πόρους. Ως χαρακτηριστικό παράδειγμα, στο Σχήμα 10 βλέπουμε τη χρησιμοποίηση της CPU καθώς και της μνήμης εντός των υποδομών δύο γνωστών παρόχων Υπηρεσιών Νέφους, ονομαστικά, της Alibaba [29] στα αριστερά και της Azure [235] στα δεξιά. Αν κοιτάξουμε την περίπτωση της Alibaba, παρατηρούμε πως ενώ η χρησιμοποίησή της CPU κυμαίνεται σε χαμηλά επίπεδα στη διάρκεια της ημέρας (μεταξύ 20%-60%) η χρησιμοποίησή της μνήμης κυμαίνεται σε πολύ υψηλά επίπεδα (άνω του 80%). Αυτή η περίπτωση μας οδηγεί σε κατακερματισμό των πόρων της CPU, οι οποίοι βρίσκονται εκεί, αλλά δεν μπορούν να χρησιμοποιηθούν από καμία εφαρμογή. Αντιθέτως, στην περίπτωση της Azure, παρατηρούμε το ακριβώς αντίθετο μοτίβο. Συγκεκριμένα, βλέπουμε πως όσο αυξάνεται ο αριθμός των πυρήνων οι οποίοι δεσμεύονται από εικονικές μηχανές, τόσο αυξάνεται και το μέγεθος της κατασπαταλημένης μνήμης, η οποία μπορεί να φτάσει και σε επίπεδα της τάξης του 10%. Αυτό οδηγεί σε κατακερματισμό της διαθέσιμης μνήμης, η οποία βρίσκεται εκεί, αλλά και πάλι δεν μπορεί να χρησιμοποιηθεί από καμία εφαρμογή.

Από τα παραπάνω, φαίνεται ξεκάθαρα πως ο κατακερματισμός πόρων αποτελεί σημαντικό πρόβλημα στα σύγχρονα συστήματα υποδομών Υπολογιστικού Νέφους. Για το λόγο αυτό, η "Αποσύνθεση Υπολογιστικών Πόρων" αποτελεί το επόμενο βήμα σε τέτοιες υποδομές, προκειμένου να μπορέσουν οι πάροχοι Υπηρεσιών Υπολογιστικού Νέφους να μπορούν να κάνουν καλύτερη διαχείριση του διαθέσιμου υλικού. Όπως είναι λογικό, σε τέτοια συστήματα, υπάρχει η ανάγκη για υλοποίηση και ανάπτυξη καινοτόμων ενορχηστρωτών, οι οποίοι λαμβάνουν υπόψη τις ιδιαιτερότητας και τα χαρακτηριστικά αυτών των νέων συστημάτων, έχοντας ως στόχο την καλύτερη χρησιμοποίηση των διαθέσιμων υπολογιστικών πόρων χωρίς να μειώνεται η απόδοση των εφαρμογών που εκτελούνται σε αυτά. Προς αυτήν την κατεύθυνση, το πλαίσιο Adrias αποτελεί τον πρώτη δρομολογητή εφαρμογών για συστήματα αποσυντεθειμένης μνήμης, το οποίο έχει ως στόχο να αποφασίζει αν κάποια εφαρμογή


2. Predictor
Offline Phase
Online Phase
Deployment Request
2b. Gather application signature
Yes
New App?
Performance prediction models
Trained models
2d. Predict application's performance
2c. Forecast future system metrics
Predicted Performance (local)
Predicted Performance (remote)
System metrics prediction model
2a. Simulation of realistic execution scenarios
1. Watcher
3. Orchestrator
Metrics Aggregator
Orchestration Logic
Local
Remote
FPGA
FPGA
Local
Remote

Σχήμα 11.. Επισκόπηση Αρχιτεκτονικής του πλαισίου Adrias

θα δεσμεύσει μνήμη είτε "παραδοσιακά", εντός του διακομιστή στον οποίο εκτελείται, είτε 'άποσυντεθειμένα", από κάποιον απομακρυσμένο διακομιστή.

4.2. Περιγραφή Αρχιτεκτονικής του Adrias

Στο Σχήμα 11 βλέπουμε μία σχηματική αναπαράσταση της αρχιτεκτονικής του πλαισίου Adrias. Το πλαίσιο αποτελείται από τρεις κύριες συνιστώσες, ονομαστικά, i) τον "Watcher", στόχος του οποίου είναι να παρακολουθεί και να συλλέγει τις μετρικές απόδοσης χαμηλού επιπέδου του συστήματος, ii) τον "Predictor", στόχος του οποίου είναι να πραγματοποιεί προβλέψεις τόσο όσο αφορά το πως θα κυμανθούν οι τιμές των μετρικών απόδοσης στο μέλλον, όσο και αναφορικά με την απόδοση των εφαρμογών οι οποίες πρόκειται να εκτελεστούν στο σύστημα, σε περίπτωση που δεσμεύσουν μνήμη είτε τοπικά είτε απομακρυσμένα, και iii) τον "Orchestrator, στόχος του οποίου είναι να αποφασίζει από ποιο κομμάτι θα δεσμεύσουν μνήμη οι εφαρμογές, λαμβάνοντας υπόψη τις προβλέψεις τις οποίες πραγματοποίησε ο Predictor. Στη συνέχεια, αναλύουμε πιο λεπτομερώς κάθε ένα από αυτά τα μέρη του πλαισίου μας.

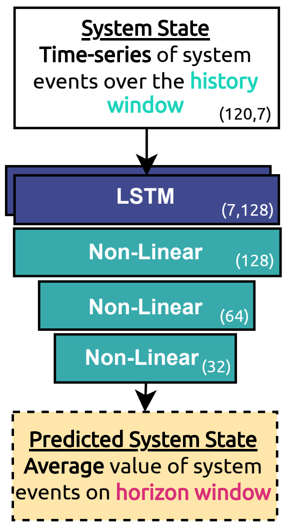
Watcher

Όπως αναφέραμε και προηγουμένως στόχος του Watcher είναι να παρακολουθεί το σύστημα και να συλλέγει μετρικές απόδοσης χαμηλού επιπέδου. Συγκεκριμένα, ο Watcher συλλέγει μετρικές οι οποίες σχετίζονται με τη χρησιμοποίηση καθώς και με τις προσβάσεις στην μνήμη, όπως για παράδειγμα τον αριθμό των αστοχιών για προσβάσεις σε δεδομένη στις κρυφές μνήμες, τον αριθμό των προσβάσεων στην τοπική κύρια μνήμη καθώς και την καθυστέρηση αλλά και το ρυθμό αποστολής και λήψης δεδομένων προς και από την απομακρυσμένη μνήμη. Μέσω αυτών των μετρικών, μπορούμε να έχουμε μία γενική εικόνα για τα διαφορετικά επίπεδα μνημών, καθώς και για τη ροή δεδομένων μεταξύ των διαφορετικών επιπέδων μνήμης στην πάροδο του χρόνου.

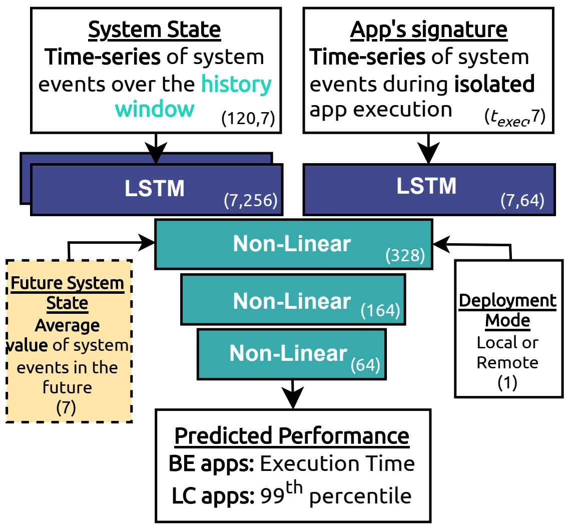
Predictor

Σκοπός του Predictor είναι να μπορεί να προβλέπει την κατάσταση του συστήματος στο μέλλον (προβλέποντας τις μετρικές απόδοσης χαμηλού επιπέδου), καθώς και να μπορεί να προβλέψει την απόδοση των εφαρμογών οι οποίες πρόκειται να εκτελεστούν στο σύστημά μας, ανάλογα με το εάν αυτές θα δεσμεύσουν τη μνήμη τους από το τοπικό ή από κάποιον απομακρυσμένο διακομιστή. Προκειμένου να πετύχει τα παραπάνω, ο Predictor χρησιμοποιεί μία αρχιτεκτονική βαθιών νευρωνικών δικτύων, η οποία αποτελείται από δύο επίπεδα πρόβλεψης όπου οι προβλέψεις του πρώτου τροφοδοτούνται στο δεύτερο και βασίζεται στα δίκτυα LSTM. Στο Σχήμα 12 βλέπουμε μία σχηματική αναπαράσταση της αρχιτεκτονικής των αυτών δικτύων.

Συγκεκριμένα, για το δίκτυο πρόβλεψης της κατάστασης του συστήματος, αυτό λαμβάνει ως είσοδο την τρέχουσα κατάσταση, η οποία αναπαριστάται μέσω ενός συνόλου διανυσμάτων χρονοσειρών, όπου κάθε διάνυσμα αντιστοιχεί σε μία διαφορετική μετρική απόδοσης από αυτές που παρακολουθεί ο Watcher. Το μέγεθος της χρονοσειράς είναι παραμετροποιήσιμο και εξαρτάται από το πόση παρελθοντική πληροφορία θέλουμε να τροφοδοτήσουμε στο σύστημά μας. Ως έξοδο, το δίκτυο προβλέπει την μέλλουσα κατάσταση του συστήματος, την οποία αναπαριστούμε μέσω ενός διανύσματος, όπου κάθε τιμή του αντιστοιχεί στον μέσο όρο της τιμής των μετρικών απόδοσης στο μέλλον. Αντίστοιχα, το μέγεθος του παραθύρου στο οποίο αντιστοιχεί αυτή η μέση τιμή είναι επίσης παραμετροποιήσιμο και εξαρ-


System State
Time-series of system events over the history window (120,7)
LSTM (7,128)
Non-Linear (128)
Non-Linear (64)
Non-Linear (32)
Predicted System State
Average value of system events on horizon window

(α′) Αρχιτεκτονική μοντέλου για πρόβλεψη της κατάστασης του συστήματος

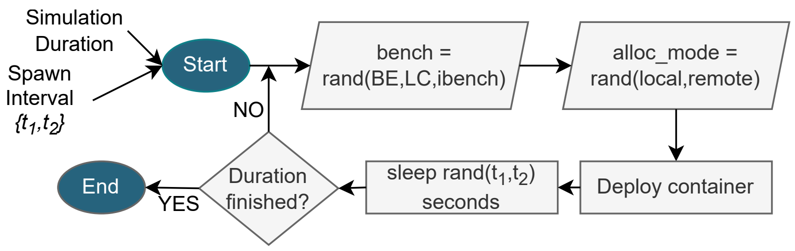


(β′) Αρχιτεκτονική μοντέλου για πρόβλεψη της απόδοσης των εφαρμογών

Σχήμα 12.. Αρχιτεκτονική Δικτύων Πρόβλεψης (α′) της κατάστασης του συστήματος και (β′) της απόδοσης των εφαρμογών. Οι παρενθέσεις υποδεικνύουν τον αριθμό των επιπέδων και των παραμέτρων ανά επίπεδο.

τάται από το για πόσο μακριά στο μέλλον θέλουμε να κάνουμε προβλέψεις για το σύστημά μας.

Αναφορικά με το δίκτυο πρόβλεψης της απόδοσης των εφαρμογών, αυτό λαμβάνει ως είσοδο τέσσερις παραμέτρους. Όμοια με το προηγούμενο δίκτυο, δέχεται ως είσοδο την τρέχουσα κατάσταση του υποκείμενου συστήματος, καθώς και την μέλλουσα κατάσταση, η οποία ουσιαστικά είναι η πρόβλεψη που πραγματοποίησε το δίκτυο πρόβλεψης της κατάστασης του συστήματος και τροφοδοτείται απευθείας από αυτό. Επιπλέον, το δίκτυο λαμβάνει ως είσοδο μία δυαδική παράμετρο, η οποία υποδηλώνει για ποιο τρόπο δέσμευσης μνήμης θέλουμε να πραγματοποιήσουμε την πρόβλεψη, όπου το 0 αντιστοιχεί σε δέσμευση μνήμης από τον τοπικό διακομιστή και το 1 σε δέσμευση μνήμης από τον απομακρυσμένο. Τέλος, το δίκτυο επίσης λαμβάνει ως είσοδο την "Υπογραφή της εφαρμογής". Ουσιαστικά, αυτή η είσοδος πρόκειται για ένα σύνολο διανυσμάτων χρονοσειρών, οι οποίες αντιστοιχούν σε μετρικές απόδοσης χαμηλού επιπέδου της εφαρμογής όταν αυτή εκτελείται κατάποκλειστικότητα στο σύστημα, δεσμεύοντας μνήμη από τον απομακρυσμένο διακομιστή. Εν γένει, το σύνολο αυτό των διανυσμάτων περιγράφει την συμπεριφορά εκτέλεσης της εφαρμογής και περιλαμβάνει πληροφορία σχετικά με τα χαρακτηριστικά της εφαρμογής όσον αφορά τις προσβάσεις της στη μνήμη, τις αστοχίες της σε πρόσβαση σε δεδομένα, κτλ. Μέσω της παραπάνω μο-

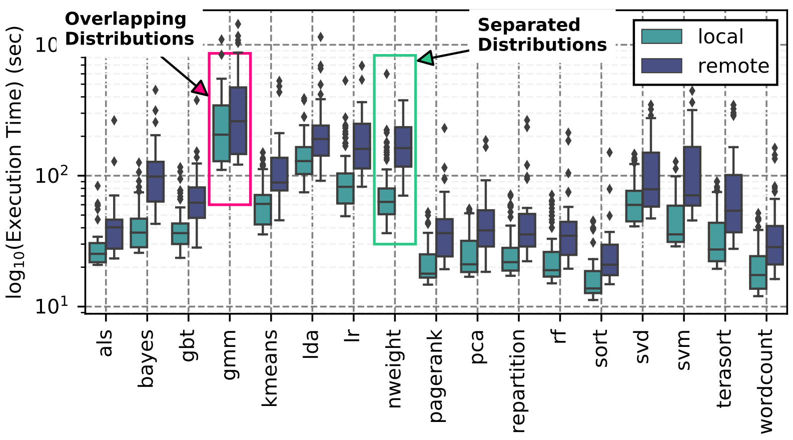


Σχήμα 13.. Διάγραμμα ροής για την παραγωγή και εκτέλεση διαφορετικών σεναρίων, για τη συλλογή δεδομένων.

ντελοποίησης και της τροφοδότησης στο δίκτυο της κατάστασης του συστήματος καθώς και της ‘ὑπογραφής” της εκάστοτε εφαρμογής, το μοντέλο πρόβλεψης απόδοσης είναι σε θέση να συσχετίσει χαρακτηριστικά του υποκείμενου συστήματος με χαρακτηριστικά των εφαρμογών που πρόκειται να εκτελεστούν στο σύστημα.

Συλλογή Δεδομένων και Εκπαίδευση Μοντέλων: Για να εκπαιδεύσουμε τα παραπάνω μοντέλα, απαιτείται η συλλογή των απαραίτητων δεδομένων τα οποία να είναι αντιπροσωπευτικά σεναρίων εκτέλεσης του πραγματικού κόσμου. Προκειμένου να το πετύχουμε αυτό, παράγουμε και εκτελούμε πολλαπλά διαφορετικά σενάρια κατά τα οποία οι εφαρμογές εισέρχονται στο σύστημα μας με τυχαίο τρόπο. Στο Σχήμα 13 φαίνεται σχηματικά η διαδικασία με την οποία παράγουμε αυτά τα σενάρια. Συγκεκριμένα, για κάθε σενάριο εκτελούμε μια τυχαία εφαρμογή στο σύστημά μας, η οποία δεσμεύει τυχαία μνήμη είτε από το τοπικό είτε από το απομακρυσμένο διακομιστή. Στη συνέχεια, περιμένουμε για κάποιο τυχαίο χρονικό διάστημα, προτού εκτελέσουμε την επόμενη εφαρμογή στο σύστημα μας, η οποία καθορίζεται ακριβώς με τον ίδιο τρόπο με προηγουμένως. Η διαδικασία αυτή επαναλαμβάνεται έως ότου υπερβούμε το συνολικό χρόνο διάρκειας του εκάστοτε σενάριο (ο οποίος έχει οριστεί κατά την αρχικοποίηση του σεναρίου).

Ανάλυση εκτέλεσης σεναρίων: Συνολικά, προκειμένου να συλλέξουμε τα απαραίτητα δεδομένα για την εκπαίδευση των μοντέλων μας, παρήχθησαν και εκτελέστηκαν 72 διαφορετικά σενάρια διάρκειας 1 ώρας το καθένα. Στο Σχήμα 14 βλέπουμε την κατανομή των χρόνων εκτέλεσης ενός υποσυνόλου των εφαρμογών που χρησιμοποιήσαμε κατά τη διαδικασία συλλογής δεδομένων, από όλα τα 72 σενάρια εκτέλεσης. Γενικά, παρατηρούμε πως η εκτέλεση των εφαρμογών δεσμεύοντας μνήμη από τον απομακρυσμένο διακομιστή οδηγεί σε αισθητή μείωση της απόδοσής τους, καθώς οι κατανομές των χρόνων εκτέλεσης για την απομακρυσμένη μνήμη διακυμαίνονται σε υψηλότερες τιμές σε όλες τις εφαρμογές. Αυτό που παρουσιάζει ιδιαίτερο ενδιαφέρον, είναι η συσχέτιση μεταξύ των κατανομών στον τοπικό και στον απομακρυσμένο διακομιστή, όσον αφορά μεμονωμένες εφαρμογές. Συγκεκριμένα, παρατηρούμε πως σε κάποιες εφαρμογές (π.χ., gmm), οι δύο κατανομές επικαλύπτονται μεταξύ τους. Για αυτές τις εφαρμογές αναμένουμε η χρήση της απομακρυσμένης μνήμης να



Σχήμα 14.. Κατανομή του χρόνου εκτέλεσης για ένα υποσύνολο εφαρμογών για όλα τα σενάρια εκτέλεσης

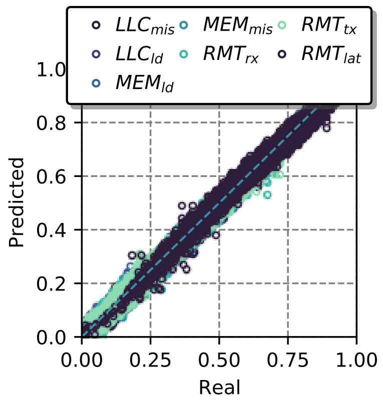
μπορεί να οδηγήσει σε καλύτερη (ή παρόμοια) απόδοση με την τοπική, σε σενάρια όπου το σύστημά μας βρίσκεται υπό την επήρεια συγκεκριμένου θορύβου. Αντιθέτως, σε άλλες εφαρμογές (π.χ., `nweight`), οι δύο κατανομές παρουσιάζουν ένα ελάχιστο υποσύνολο τομής μεταξύ των τιμών τους. Για αυτές τις εφαρμογές αναμένουμε η χρήση της απομακρυσμένης μνήμης να είναι απαγορευτική, καθώς θα οδηγήσει σε τεράστια μείωση της απόδοσης.

Orchestrator

Τέλος, σκοπός του Orchestrator είναι να αποφασίζει εάν μία εφαρμογή που εισέρχεται στο σύστημά μας θα δεσμεύσει μνήμη από τον τοπικό ή από τον απομακρυσμένο κόμβο. Συγκεκριμένα, όταν μία εφαρμογή πρόκειται να εκτελεστεί στο σύστημα, ο Orchestrator αρχικά εξετάζει εάν διαθέτει κάποια προγενέστερη πληροφορία σχετικά με την 'υπογραφή" της. Εάν όχι, τότε η εφαρμογή εκτελείται απομονωμένα, προκειμένου να συλλέξουμε τις απαραίτητες μετρικές εκτέλεσης, που να χαρακτηρίζουν την εφαρμογή. Σε αντίθετη περίπτωση, ο Orchestrator επικοινωνεί με τον Predictor προκειμένου να λάβει την πρόβλεψη των εκτιμώμενων χρόνων εκτέλεσης της συγκεκριμένης εφαρμογής, εάν αυτή εκτελεστεί δεσμεύοντας μνήμη από τον τοπικό ή από κάποιον απομακρυσμένο κόμβο. Με βάση αυτές τις προβλέψεις, ο Orchestrator εφαρμόζει μία απλή λογική για να αποφασίσει τον τελικό τρόπο εκτέλεσης, η οποία διαφοροποιείται ανάλογα με τη φύση της εκάστοτε εφαρμογής, στοχεύοντας σε κάθε περίπτωση στην ελαχιστοποίηση της μείωσης της απόδοσης των εφαρμογών, χρησιμοποιώντας παράλληλα όσο περισσότερο γίνεται την απομακρυσμένη μνήμη.

Event	$\mathbf{R^2}$
LLC_{mis}	0.9969
LLC_{ld}	0.9995
MEM_{ld}	0.9641
MEM_{st}	0.9983
RMT_{lat}	0.9977
RMT_{rx}	0.9871
RMT_{tx}	0.9876
Avg.	0.9932

Πίνακας 1.. Σκορ R^2 ανά μετρική απόδοσης



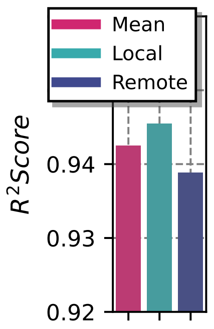
Σχήμα 15.. Προβλέψεις τιμών έναντι πραγματικών σχετικά με μετρικές απόδοσης του συστήματος στο μέλλον

4.3. Πειραματική Αξιολόγηση

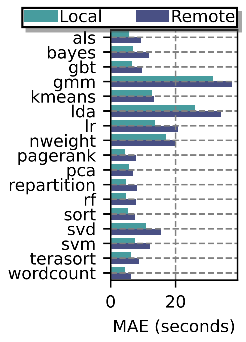
Σε αυτήν την ενότητα δείχνουμε κάποια ενδεικτικά αποτελέσματα σχετικά με την αποδοτικότητα και αποτελεσματικότητα του πλαισίου Adrias. Συγκεκριμένα οι άξονες που εξετάζουμε είναι το κατά πόσο μπορεί το πλαίσιο Adrias να i) πραγματοποιήσει ακριβείς προβλέψεις σχετικά με τον εκτιμώμενο χρόνο εκτέλεσης για γνωστές εφαρμογές και ii) χρησιμοποιήσει την απομακρυσμένη μνήμη αποδοτικά, χωρίς να επηρεάζει την απόδοση των εφαρμογών που εκτελούνται στο σύστημά μας. Προκειμένου να αξιολογήσουμε το προτεινόμενο πλαίσιο, χρησιμοποιήσαμε εφαρμογές από τρεις διαφορετικές σουίτες [2, 104, 105], ενώ τα πειράματά μας διεξήχθησαν σε ένα πραγματικό σύστημα αποσυντεθειμένης μνήμης [1].

Ακρίβεια προβλέψεων για γνωστές εφαρμογές

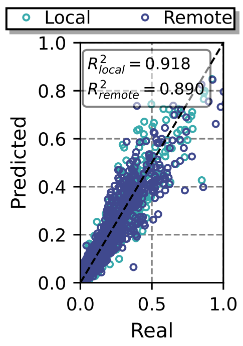
Δίκτυο πρόβλεψης της κατάστασης του συστήματος: Αρχικά, εξετάζουμε το κατά πόσο το δίκτυο της κατάστασης του συστήματος παράγει ακριβείς προβλέψεις σχετικά με τις μελλοντικές τιμές των μετρικών απόδοσης χαμηλού επιπέδου. Προκειμένου να αξιολογήσουμε το μοντέλο μας, χωρίζουμε το σύνολο δεδομένων μας σε δύο υποσύνολα, όπου το πρώτο από αυτά (60% του συνολικού) χρησιμοποιείται προκειμένου να εκπαιδεύουμε το μοντέλο και το δεύτερο (40%) προκειμένου να μετρήσουμε την ακρίβειά του. Το Σχήμα 15 δείχνει τα αντίστοιχα αποτελέσματα, όπου στον πίνακα φαίνονται οι μέσοι όροι ακρίβειας (χρησιμοποιώντας ως μετρική απόδοσης το σκορ R^2), ενώ στο αντίστοιχο διάγραμμα οι



(α΄) Συνολική ακρίβεια



(β΄) MAE ανά εφαρμογή

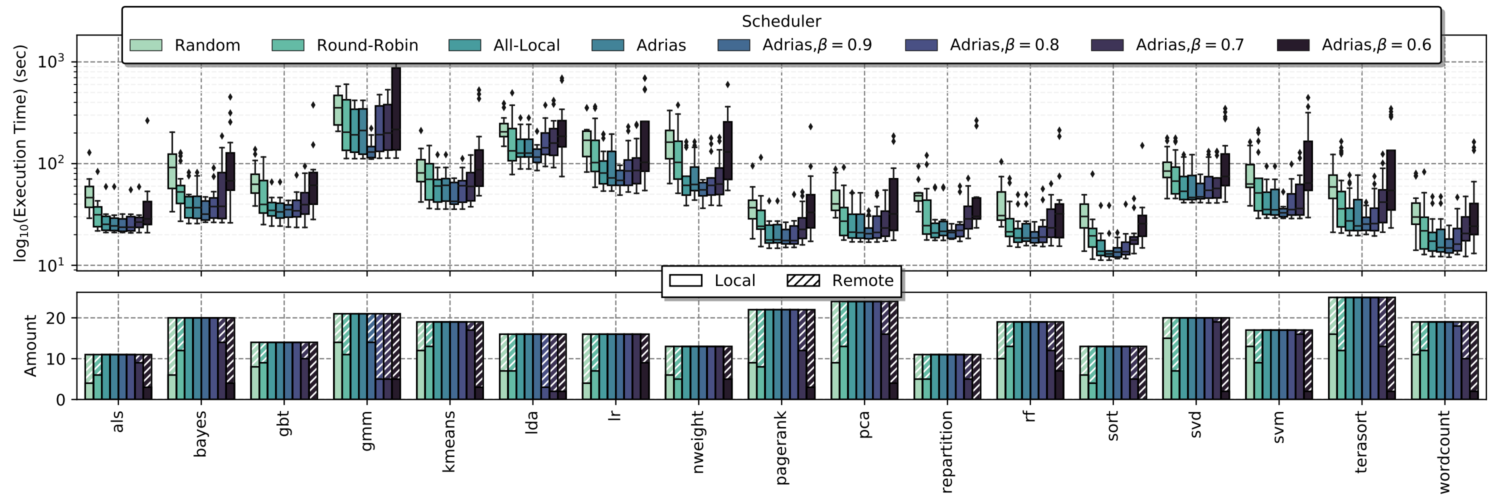


(γ΄) Προβλέψεις τιμών έναντι πραγματικών

Σχήμα 16.. Αξιολόγηση του μοντέλου πρόβλεψης απόδοσης εφαρμογών

πραγματικές τιμές έναντι των προβλεπόμενων. Παρατηρούμε πως το δίκτυο πρόβλεψης της κατάστασης του συστήματος παράγει πολύ ακριβείς προβλέψεις, με το σκορ R^2 να κυμαίνεται μεταξύ 0.964 και 0.993 συνολικά, και 0.99 κατά μέσο όρο.

Δίκτυο πρόβλεψης της απόδοσης των εφαρμογών: Στη συνέχεια αξιολογούμε την αποδοτικότητα του δικτύου πρόβλεψης της απόδοσης των εφαρμογών. Όμοια με το προηγούμενο πείραμα, χωρίζουμε το αρχικό μας σύνολο δεδομένων σε δύο υποσύνολα (60% και 40%), χρησιμοποιώντας το κάθε υποσύνολο για την εκπαίδευση και αξιολόγηση του μοντέλου. Στο Σχήμα 16 βλέπουμε τα αντίστοιχα αποτελέσματα. Παρατηρούμε πως το δίκτυο πρόβλεψης της απόδοσης των εφαρμογών παρουσιάζει αρκετά υψηλή ακρίβεια (Σχήμα 16α΄), πετυχαίνοντας κατά μέσο όρο σκορ 0.942 R^2, με λίγο υψηλότερη απόδοση αναφορικά με τις προβλέψεις για την τοπική μνήμη ($R^2 = 0.945$) έναντι της απομακρυσμένης ($R^2 = 0.939$). Επίσης, στο Σχήμα 16β΄ παραθέτουμε και την ακρίβεια των προβλέψεών μας για κάθε εφαρμογή ξεχωριστά, χρησιμοποιώντας ως μετρική απόδοσης το μέσο απόλυτο σφάλμα (MAE), ενώ στο Σχήμα 16γ΄ βλέπουμε και τις αντίστοιχες προβλεπόμενες τιμές έναντι των πραγματικών. Παρατηρούμε πως οι η απόδοση των προβλέψεων του μοντέλου μας διαφοροποιείται ανάλογα με την εφαρμογή για την οποία πραγματοποιεί τις προβλέψεις. Για παράδειγμα, βλέπουμε πως σε περιπτώσεις όπως π.χ. αυτήν της εφαρμογής rf το σφάλμα του μοντέλου μας κυμαίνεται σε χαμηλά επίπεδα, ενώ σε άλλες (π.χ., gmm) το σφάλμα αυξάνεται αισθητά. Παρ΄όλα αυτά, ακόμα και σε αυτές τις περιπτώσεις, το συνολικό σφάλμα αντιστοιχεί μόνο σε ένα 10% του μέσου χρόνου εκτέλεσης, αποδεικνύοντας την αποτελεσματικότητα του πλαισίου μας να πραγματοποιεί ακριβείς προβλέψεις.



Σχήμα 17.. Κατανομή των χρόνων εκτέλεσης των εφαρμογών (επάνω) και συνολικός αριθμός όπου η εφαρμογή δέσμευσε μνήμη από το τοπικό και το απομακρυσμένο διακομιστή (κάτω) για διαφορετικές λογικές ενορχήστρωσης των εφαρμογών

Αξιολόγηση του ενορχηστρωτή εφαρμογών: Στο επάνω μέρος του Σχήματος 17 παρουσιάζεται η κατανομή του χρόνου εκτέλεσης για ένα υποσύνολο των υπό εξέταση εφαρμογών ενώ στο κάτω, ο αριθμός των φορών που κάθε εφαρμογή δέσμευσε μνήμη τοπικά και απομακρυσμένα μνήμη όταν χρησιμοποιούνται διαφορετικές λογικές δρομολόγησης των εφαρμογών. Συγκεκριμένα, η λογική All-Local δεσμεύει για όλες τις εφαρμογές μνήμη από τον τοπικό διακομιστή, η λογική Random δεσμεύει μνήμη τυχαία μεταξύ του τοπικού και του απομακρυσμένου και η λογική Round-Robin δεσμεύει μνήμη εναλλάξ μεταξύ των δύο. Τα αποτελέσματα αποκαλύπτουν πως για την πλειονότητα των εφαρμογών, οι λογικές Random και Round-Robin παρέχουν τις χειρότερες κατανομές επιδόσεων, επιβεβαιώνοντας την ανάγκη για έξυπνες λογικές ενορχήστρωσης της μνήμης. Παρατηρούμε πως για μεγαλύτερες τιμές της παραμέτρου β[3] ο ενορχηστρωτής του πλαισίου Adrias παρέχει πανομοιότυπες αποφάσεις με τη λογική All-Local σχεδυλινγ, λόγω της εν γένει χειρότερης επίδοσης της απομακρυσμένης μνήμης συγκριτικά με την τοπική σε συνδυασμό με το σφάλμα των προβλέψεων που προκύπτει από τα μοντέλα μας. Για τιμές β ίσες με 0.8 και 0.7 το πλαίσιο Adrias καταφέρνει να αξιοποιήσει αποτελεσματικά την απομακρυσμένη μνήμη, δρομολογώντας περίπου το 10% και το 35% των εφαρμογών με μια πτώση της τάξης του 0.5% και 15% στη μέση απόδοση για όλες τις εφαρμογές αντίστοιχα. Ενώ οι τιμές $\beta = 0.8$ και $\beta = 0.7$ θα υποδήλωναν ισοδύναμη υποβάθμιση της απόδοσης των εφαρμογών, παρατηρούμε ότι αυτό δεν συμβαίνει, γεγονός που αποδίδεται στο σφάλμα ακρίβειας του μοντέλου πρόβλεψης της απόδοσης. Επιπλέον, η λογική δρομολόγησης του πλαισίου Adrias ευνοεί την εκφόρτωση ορισμένων εφαρμογών στην απομακρυσμένη μνήμη (π.χ. `gmm`, `lda`), οι οποίες, όπως αποδείχθηκε στην ενότητα 4.2 παρουσιάζουν 'επικαλυπτόμενες'' κατανομές απόδοσης μεταξύ τοπικών και απομακρυσμένων τρόπων λειτουργίας, ενώ αποφεύγει την εκφόρτωση αυτών που παρουσιάζουν ''μη επικαλυπτόμενες'' κατανομές (π.χ. texttnweight). Η παρατήρηση αυτή επαληθεύει ότι το Adrias είναι σε θέση να μοντελοποιήσει και να ερμηνεύσει σωστά τα εγγενή χαρακτηριστικά των εξεταζόμενων εφαρμογών. Τέλος, για χαμηλότερες τιμές σλαςκ (π.χ. $\beta = 0.6$) το Adrias μεταφορτώνει την πλειονότητα των εφαρμογών στην απομακρυσμένη μνήμη, γεγονός που, ωστόσο, προκαλεί σημαντική υποβάθμιση των επιδόσεων.

4.4. Επίλογος

Η ''Αποσύνθεση Υπολογιστικών Πόρων'' αποτελεί το επόμενο μεγάλο βήμα για την αποτελεσματική και λεπτομερή διαχείριση των υποδομών Υπολογιστικού Νέφους. Σε αυτό το κεφάλαιο παρουσιάσαμε το Adrias, ένα πλαίσιο παρακολούθησης και ενορχήστρωσης για συστήματα νέφους με αποσυντεθειμένη μνήμη. Με γνώμονα μία εκτενή ανάλυση αρκετών εφαρμογών, σχεδιάσαμε το Adrias, ένα πλαίσιο που αξιοποιεί τεχνικές βαθιάς μάθησης για να

[3]Η παράμετρος β υποδηλώνει ένα άνω όριο για το μέγιστο ποσοστό απόδοσης που δεχόμαστε να θυσιάσουμε προκειμένου να χρησιμοποιήσουμε την απομακρυσμένη μνήμη.

αποφασίζει τον τρόπο λειτουργίας της μνήμης των εφαρμογών υπό εκτέλεση και δείξαμε ότι μπορεί να αξιοποιήσει αποτελεσματικά την απομακρυσμένη μνήμη με ελάχιστη επιβαρύνση στην απόδοσή τους.

5. Πλαίσιο Sparkle

5.1. Εισαγωγή

Σήμερα, ο όγκος δεδομένων που παράγεται καθημερινά είναι πραγματικά συντριπτικός, με πάνω από πάνω από 2,5 πεντακισεκατομμύρια bytes δεδομένων να παράγονται κάθε μέρα [360]. Για την επεξεργασία και την αξιοποίηση αυτών των δεδομένων έχουν εμφανιστεί καινούρια πλαίσια [92, 141, 245], τα οποία επιτρέπουν την παράλληλη επεξεργασία τεράστιων όγκων δεδομένων με κατανεμημένο τρόπο. Το πλαίσιο Apache Spark [245] αποτελεί ένα μία από τις πιο γνωστές πλατφόρμες επεξεργασίας δεδομένων μεγάλου όγκου και πλέον υποστηρίζεται ως υπηρεσία σε σύγχρονες υποδομές Υπολογιστικού Νέφους [361].

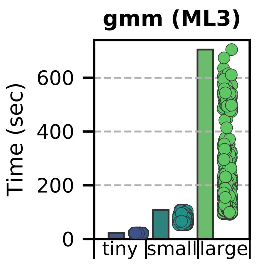
Παρόλο που η αρχιτεκτονική του πλαισίου Spark παρέχει πολύ υψηλές επιδόσεις, προσφέρει επίσης μια μεγάλη ποικιλία παραμέτρων, οι οποίες μπορούν να ρυθμιστούν ώστε να αλλάξουν διάφορες πτυχές της εσωτερικής διαρύθμισής του, προσφέροντας έτσι περαιτέρω αύξηση της απόδοσης. Πιο συγκεκριμένα, οι τελευταίες εκδόσεις του Spark διαθέτουν περισσότερες από 150 παραμέτρους διαμόρφωσης [362]. Παρά τη συμαντικότητα αυτών των παραμέτρων, οι επίσημες οδηγίες ρύθμισης της απόδοσης του Spark περιγράφουν μόνο ένα πολύ περιορισμένο υποσύνολο παραμέτρων του [247], αφήνοντας έτσι το βάρος της ανάλυσης των επιπτώσεών όλων των υπολοίπων αποκλειστικά στους τελικούς χρήστες/προγραμματιστές. Όμως, η ανάλυση και διερεύνηση της επίδρασης των διαφόρων παραμέτρων στην απόδοση των εφαρμογών, καθώς και η εξέταση της συσχέτισης μεταξύ των διαφόρων παραμέτρων είναι μια επίπονη διαδικασία, λόγω του *ι)* πολυδιάστατου χώρου διαθέσιμων παραμέτρων και τιμών ανά παράμετρο, *ιι)* του τεράστιου, απαιτούμενου χρόνου εκτέλεσης προκειμένου να διερευνηθούν τα παραπάνω, καθώς και *ιιι)* του χρόνου που απαιτείται για την κατανόηση σε βάθος του σκοπού της κάθε παραμέτρου. Επιπλέον, η αλληλοσυσχέτιση μεταξύ των διαφόρων παραμέτρων εισάγει ένα επιπλέον επίπεδο πολυπλοκότητας. Κατά συνέπεια, οι προγραμματιστές εφαρμογών τείνουν να ρυθμίζουν εμπειρικά μόνο τις πιο προφανείς παραμέτρους που σχετίζονται με την απόδοση, όπως ο αριθμός των executors ή η μνήμη RAM ανά executor.

Αν και το ίδιο το Spark προσφέρει ένα εγχειρίδιο για τη ρύθμιση των παραμέτρων του [247], επικεντρώνεται κυρίως στη σειριοποίηση δεδομένων και τον συντονισμό της μνήμης. Επίσης, η ο τρόπος με τον οποίο προτείνεται να γίνεται η ρύθμιση των παραμέτρων βασίζεται σε μία "trial and error" λογική, η οποία πρέπει να επαναλαμβάνεται για κάθε διαφορετική εφαρμογή και διαφορετικό μέγεθος δεδομένων, με αποτέλεσμα να απέχει πολύ από την παροχή βέλτιστων αποτελεσμάτων. Τέλος, η υψηλής διάστασης φύση του χώρου παραμέτρων που εκτίθεται από τη μηχανή Spark αποτελεί πρόκληση τόσο για τις στρατηγικές μοντελοποίησης και βελτιστοποίησης. Από την παραπάνω συζήτηση, είναι προφανές ότι υπάρχει ανάγκη για αυτοματοποιημένα πλαίσια συντονισμού, τα οποία να μπορούν αυτοματοποιημένα να ρυθμίζουν όλες τις διαφορετικές παραμέτρους του Spark, έτσι ώστε να διευκολύνεται η εξερεύνηση αυτού του χώρου αναζήτησης μεγάλων διαστάσεων.

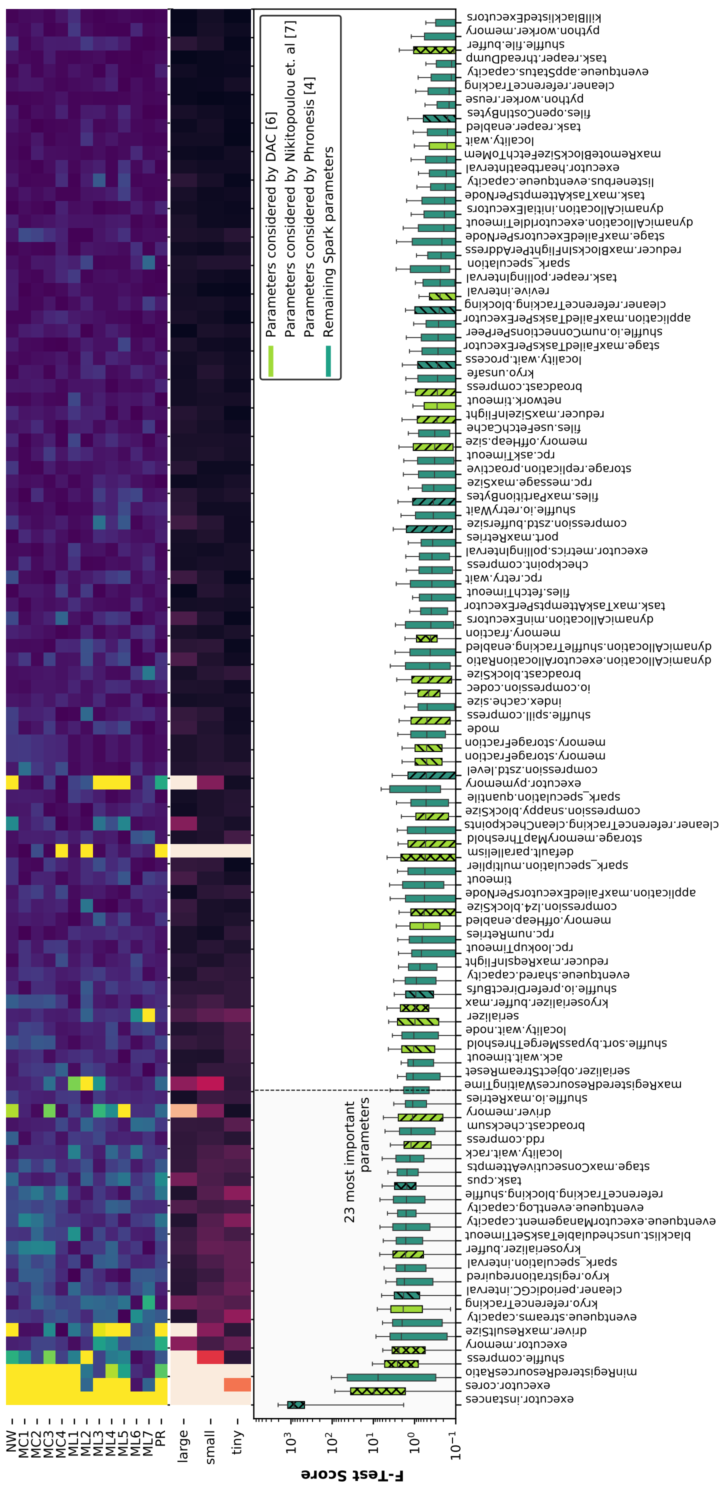
Για την αντιμετώπιση των προαναφερθέντων προκλήσεων, σε αυτό το κεφάλαιο προτείνουμε το *Sparkle*, ένα πλαίσιο αυτόματης ρύθμισης των παραμέτρων του πλαισίου Spark. Το *Sparkle* βασίζεται σε τεχνικές βαθιάς μάθησης και σε μετρικές επιδόσεων χαμηλού επιπέδου για τη μοντελοποίηση της απόδοσης τόσο γνωστών, όσο και άγνωστων, εφαρμογών Spark. Χρησιμοποιώντας αλγορίθμους γενετικής βελτιστοποίησης, το πλαίσιο διασχίζει αποτελεσματικά τον χώρο αναζήτησης σε πραγματικό χρόνο και βελτιστοποιεί την τιμή των παραμέτρων του πλαισίου Spark. Το *Sparkle* επεκτείνει τα υπάρχοντα πλαίσια αυτόματης ρύθμισης παραμέτρων Spark, παρέχοντας μια καθολική λύση για μοντελοποίηση της απόδοσης των εφαρμογών, σε αντίθεση με λύσεις οι οποίες εξετάζουν κάθε εφαρμογή (ή/και σύνολο δεδομένων) χωριστά, ενώ παράλληλα ρυθμίζει και όλες τις παραμέτρους που παρέχει το πλαίσιο Spark, εξαλείφοντας έτσι πλήρως την ανάγκη για τον προσδιορισμό των πιο σημαντικών παραμέτρων με εμπειρικό τρόπο από κάποιον 'ειδικό'', ή μέσω στατιστικών αναλύσεων.

5.2. Σημαντικότητα πλαισίου Sparkle

Εν γένει, η προσεκτική και σωστή ρύθμιση των παραμέτρων του Spark μπορεί να προσφέρει τεράστια κέρδη όσον αφορά την απόδοση των εφαρμογών οι οποίες εκτελούνται μέσω αυτού. Ενδεικτικά, στο Σχήμα 18 βλέπουμε το εύρος επιτάχυνσης που μπορούμε να επιτύχουμε ρυθμίζοντας κάποιες από τις παραμέτρους που προσφέρει το πλαίσιο Spark για μία ενδεικτική εφαρμογή, όπου ο άξονας x υποδηλώνει διαφορετικά μεγέθη συνόλου δεδομένων εισόδου. Παρατηρούμε πως για μεγάλα σύνολα δεδομένων, μπορούμε να πετύχουμε έως και ×7 επιτάχυνση, ενώ βλέπουμε επίσης πως και η επιτάχυνση μεταβάλλεται ανάλογα με το μέγεθος του συνόλου δεδομένων εισόδου. Παρά τα τεράστια κέρδη που μπορεί να αποφέρει η ρύθμιση των παραμέτρων του Spark, όπως αναφέραμε και στο εισαγωγικό κεφάλαιο, πρόκειται για μια επίπονη διαδικασία η οποία χρειάζεται μεγάλη εμπειρία σχετικά με την αρχιτεκτονική καθώς και τον τρόπο με τον οποίο λειτουργεί η μηχανή του Spark εσωτερικά, όπως επίσης η βελτιστοποίηση είναι και συνάρτηση της εφαρμογής αυτής καθαυτής, με διαφορετικές εφαρμογές να έχουν διαφορετική συμπεριφορά για ίδια ρύθμιση των παραμέτρων.



Σχήμα 18.. Επιτευχθείσα επιτάχυνση για διαφορετικές ρυθμίσεις παραμέτρων



Σχήμα 19.. Σημαντικότητα όλων των παραμέτρων του πλαισίου Spark όπως προέκυψε από το στατιστικό έλεγχο ANOVA F-Test. Επάνω: Σημαντικότητα ανά εφαρμογή. Μέση: Σημαντικότητα ανά μέγεθος συνόλου δεδομένων εισόδου. Κάτω: Συνολικά αποτελέσματα και παράμετροι τις οποίες εξέτασαν προηγούμενες ερευνητικές εργασίες.

Για τον λόγο αυτό, στο παρελθόν έχουν προταθεί και ερευνηθεί αρκετά πλαίσια τα οποία προσπαθούν να βελτιστοποιήσουν την ρύθμιση των παραμέτρων του πλαισίου Spark με αυτοματοποιημένο τρόπο. Παρ᾽όλα αυτά, τα προτεινόμενα αυτά πλαίσια υστερούν σε δύο βασικά σημεία. Πρώτον, όλες οι προτεινόμενες λύσεις προσπαθούν να ρυθμίσουν ένα υποσύνολο από τις διαθέσιμες παραμέτρους που προσφέρει η πλατφόρμα του Spark, τις οποίες επιλέγουν είτε με εμπειρικό τρόπο (π.χ., οι ερευνητές που πραγματοποίησαν την ανάπτυξη του εκάστοτε πλαισίου θεώρησαν 10 σημαντικές παραμέτρους λόγω πρόταιρης γνώσης) είτε μέσω στατιστικών μελετών (π.χ., υπολογίζοντας τη σημαντικότητα της κάθε παραμέτρου όσον αφορά την επίδρασή της στην απόδοση των εφαρμογών).

Γιατί όμως είναι απαραίτητο να ρυθμίσουμε και να βελτιστοποιήσουμε όλες τις διαθέσιμες παραμέτρους και όχι ένα υποσύνολο αυτών;

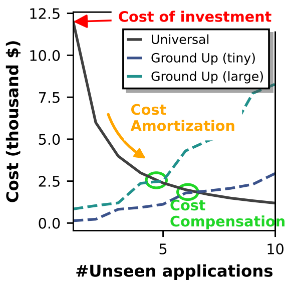
Προκειμένου να απαντήσουμε σε αυτήν την ερώτηση, αξιολογήσαμε τη σημαντικότητα όλως των παραμέτρων Spark μέσω ενός στατιστικού ελέγχου, και συγκεκριμένα του ANOVA F-Test [367]. Συγκεκριμένα, εκτελέσαμε διαφορετικές εφαρμογές Spark για διαφορετικές τιμές παραμέτρων εισόδου και διαφορετικά μεγέθη συνόλου δεδομένων εισόδου και αξιολογήσαμε τη σημαντικότητα ανά παράμετρο Spark σε όλα τα εκτελούμενα σενάρια.

Το boxplot (κάτω μέρος) του Σχήματος 19 δείχνει τα αντίστοιχα αποτελέσματα, όπου ο άξονας y υποδηλώνει τη σημαντικότητα της αντίστοιχης παραμέτρου που προέκυψε από το τεστ και ο άξονας x δείχνει όλες τις εξεταζόμενες παραμέτρους Spark ταξινομημένες σε φθίνουσα σειρά με βάση τη σημαντικότητα τους. Όπως αναμενόταν, φαίνεται ότι ο αριθμός των executors (`executor-instances`) αποτελεί με διαφορά τη σημαντικότερη παράμετρο, γεγονός που αποτελεί και τον λόγο που οι προηγούμενες προσπάθειες [146, 148] επικεντρώνονται αποκλειστικά στη βελτιστοποίηση αυτής της παραμέτρου. Επίσης, βλέπουμε πως η παράμετρος `scheduler.minRegisteredResourcesRatio` αποτελεί την τρίτη πιο σημαντική παράμετρο στην περίπτωσή μας, ωστόσο, προηγούμενες επιστημονικές προσπάθειες δεν έχουν λάβει καθόλου υπόψη τους αυτήν την παράμετρο. Τέλος, σχετικά με την σημαντικότητα για διαφορετικές εφαρμογές και διαφορετικά σύνολα δεδομένων εισόδου, βλέπουμε πως η σημαντικότητα των παραμέτρων μεταβάλλεται κατά περίπτωση, με κάποιες παραμέτρους να παρουσιάζουν μεγαλύτερη ευαισθησία σε συγκεκριμένα σενάρια και άλλες σε άλλα. Συνολικά, παρατηρούμε μια διαφορετική συμπεριφορά όσον αφορά τη σημασία των πρώτων 33 παραμέτρων, ενώ για τις υπόλοιπες η επίδραση φαίνεται να είναι μικρότερη.

Από την παραπάνω συζήτηση, είναι εμφανές πως ο τρόπος με τον οποία θα πρέπει να γίνεται η επιλογή των σημαντικότερων παραμέτρων του πλαισίου Spark δεν είναι προφανής και δεν υπάρχει "χρυσός κανόνας" για τον προσδιορισμό της ιδανικής διαμόρφωσης, καθώς η διαδικασία αυτή εξαρτάται, μεταξύ άλλων, από τη φύση της εφαρμογής που πρόκεται να εκτελεστεί καθώς και του συνόλου δεδομένων εισόδου της.

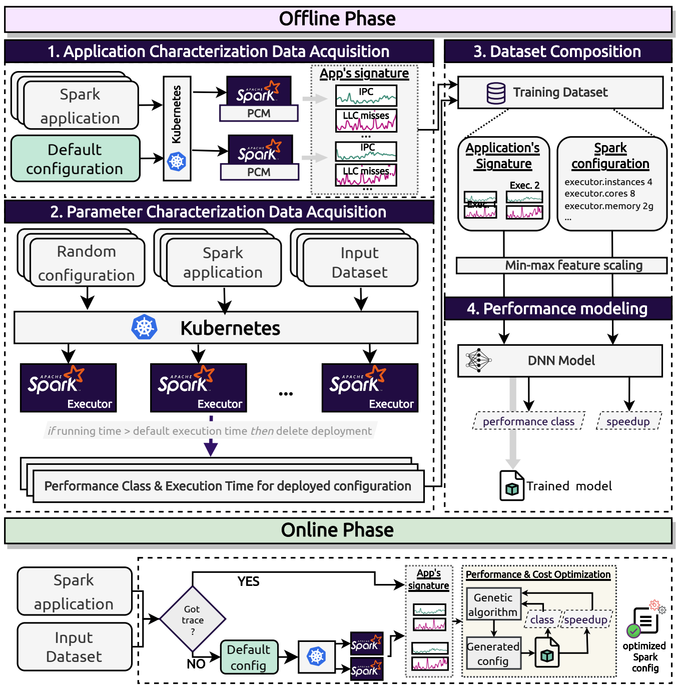
Επίσης, γιατί η καθολική μοντελοποίηση είναι σημαντική;

Το Spark είναι ένα πλαίσιο ανάλυσης δεδομένων γενικού σκοπού που επιτρέπει την ανάπτυξη νέων (ενδεχομένως άγνωστων) εφαρμογών. Τα πλαίσια τα οποία έχουν προταθεί στο παρελθόν προκειμένου να ρυθμίζουν αυτόματα τις παραμέτρους του πλαισίου Spark χρησιμοποιούν τεχνικές μοντελοποίησης επιδόσεων σε επίπεδο εφαρμογής [6, 146, 148] ή συστάδας εφαρμογών [7]. Ενώ αυτές οι τεχνικές έχουν δείξει ότι βελτιώνουν την απόδοση μεμονωμένων εφαρμογών, είναι ανεπαρκείς για γενίκευση σε νέες/άγνωστες εφαρμογές. Έτσι, τα συμβατικά πλαίσια αυτόματης ρύθμισης, όταν βελτιστοποιούν για μια νέα εφαρμογή, απαιτούν την επανάληψη ολόκληρου του κύκλου ζωής της ρύθμισης, ο οποίος περιλαμβάνει τη συλλογή δεδομένων απόδοσης για διαφορετικές διαμορφώσεις του Spark, την εκπαίδευση ενός νέου μοντέλου μηχανικής μάθησης από το μηδέν και τη ρύθμιση των υπερπαραμέτρων του για την αύξηση της ακρίβειας πρόβλεψής του. Με έναν αυξανόμενο αριθμό νέων εφαρμογών, το κόστος που σχετίζεται με αυτή τη διαδικασία γίνεται απαγορευτικά υψηλό λόγω της συλλογής νέων δεδομένων, ιδίως για εφαρμογές με μεγάλα σύνολα δεδομένων εισόδου, η εκτέλεση των οποίων μπορεί να διαρκέσει αρκετές ημέρες [368].



Σχήμα 20.. Κόστος ενός καθολικού μοντέλου πρόβλεψης έναντι πολλαπλών ξεχωριστών, ένα ανά εφαρμογή

Για την αντιμετώπιση αυτής της πρόκλησης, στην προσέγγισή μας μέσω του *Sparkle* υιοθετούμε ένα καθολικό τρόπο μοντελοποίησης που μετριάζει το συνολικό κόστος του συντονισμού άγνωστων εφαρμογών κατανέμοντας τα έξοδα της συλλογής δεδομένων και της εκπαίδευσης του μοντέλου μεταξύ διαφορετικών εφαρμογών. Στο Σχήμα 20 παρουσιάζεται το κόστος συλλογής δεδομένων και εκπαίδευσης ενός μοντέλου ΜΜ για έναν καθολικό τρόπο μοντελοποίησης (χωρίς απώλεια γενικότητας, θεωρούμε εκπαίδευση με 20 διαφορετικά ζεύγη εφαρμογών/συνόλων δεδομένων) έναντι ενός μοντέλου ΜΜ από την αρχή για κάθε νέα εφαρμογή. Λαμβάνοντας υπόψη το μοντέλο χρέωσης της AWS, ο άξονας y αντιπροσωπεύει το κόστος της ανάπτυξης μιας συστάδας 10 κόμβων `c4.8xlarge`, ενώ ο άξονας x απεικονίζει τη μεταβολή του κόστους καθώς προστίθενται περισσότερες άγνωστες εφαρμογές. Στην προσέγγιση ενός μοντέλου ΜΜ ανά εφαρμογή, οι χρήστες πρέπει να πληρώνουν για κάθε νέα εφαρμογή, με αποτέλεσμα το κόστος να αυξάνεται συνεχώς ανάλογα με τον αριθμό των εφαρμογών που εκτελούνται στο σύστημα. Αντιθέτως, στον καθολικό τρόπο μοντελοποίησης της απόδοσης των εφαρμογών, έχουμε ένα αρχικό κόστος επένδυσης, το οποίο χρειάζεται για τη συλλογή του αρχικού συνόλου δεδομένων εκπαίδευσης και τη δημιουργία του καθολικού μοντέλου ΜΜ. Καθώς όμως εκτελούνται όλο και περισσότερες άγνωστες εφαρμογές, το κόστος αυτό αποσβένεται, οδηγώντας σε ένα σημείο αντιστάθμισης όπου το αρχικό κόστος επένδυσης επιστρέφεται από την εξοικονόμηση που οφείλεται στην ικανότητα του καθολικού αυτόματου συντονισμού να βελτιστοποιεί προηγουμένως αθέατες εφαρμογές.



Σχήμα 21.. Επισκόπηση Αρχιτεκτονικής του πλαισίου *Sparkle*

5.3. Περιγραφή Αρχιτεκτονικής του *Sparkle*

Το *Sparkle* αποτελεί ένα ολοκληρωμένο πλαίσιο που ρυθμίζει αυτόματα τις παραμέτρους διαμόρφωσης του Spark, προκειμένου να βελτιστοποίησει την απόδοση και το κόστος των εφαρμογών προς εκτέλεση. Στο Σχήμα 21 βλέπουμε μία επισκόπηση της αρχιτεκτονικής του πλαισίου ενγ*Σπαρκλε*. Το *Sparkle* αποτελείται δύο φάσεις· μία κατά την οποία συλλέγονται τα απαραίτητα δεδομένα και εκπαιδεύεται το καθολικό μοντέλο ΜΜ και μια κατά την οποία το πλαίσιο ρυθμίζει και αποφασίζει αυτόματα τις βέλτιστες παραμέτρους για εφαρμογές οι οποίες είναι προς εκτέλεση.

Φάση συλλογής δεδομένων και εκπαίδευσης μοντέλου

Όπως αναφέραμε και προηγουμένως, το πλαίσιο Spark προσφέρει παραπάνω από 150 διαφορετικές παραμέτρους. Ως πρώτο βημα της φάσης συλλογής δεδομένων, επιλέγουμε μόνο το υποσύνολο των παραμέτρων οι οποίες αφορούν και επηρεάζουν την απόδοση των εφαρμογών. Για παράδειγμα η παράμετρος `spark.app.name` η οποία ρυθμίζει το όνομα της εφαρμογής παραλείπεται. Αυτό μας οδηγεί σε ένα υποσύνολο του αρχικού, το οποίο αποτελείται από 101 παραμέτρους. Για αυτές τις παραμέτρους εξετάζουμε ένα διακριτό σύνολο τιμών, με τρεις τιμές ανά παράμετρο, οι οποίες αντιστοιχούν σε χαμηλές, μεσαίες και υψηλές τιμές ρύθμισης της αντίστοιχης παραμέτρου.

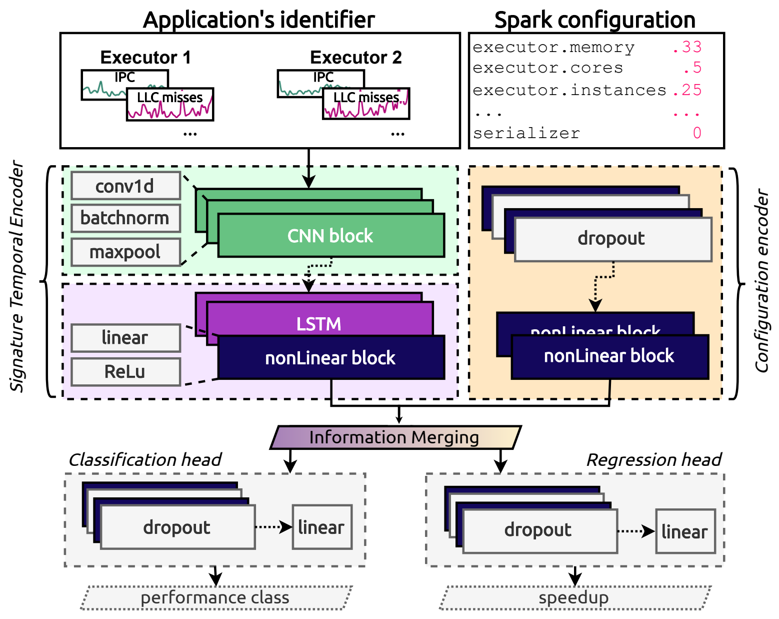
Συλλογή δεδομένων για χαρακτηρισμό των εφαρμογών: Ως πρώτο βήμα, το *Sparkle* συλλέγει δεδομένα προκειμένου να χαρακτηρίσει τις εφαρμογές οι οποίες αποτελούν το σύνολο εκπαίδευσής για το μοντέλο μας. Προκειμένου να το πετύχει αυτό, εκτελεί κάθε εφαρμογή χρησιμοποιώντας τις προκαθορισμένες παραμέτρους του Spark και κατά τη διάρκεια εκτέλεσης συλλέγει από το σύστημα μετρικές απόδοσης χαμηλού επιπέδου. Οι μετρικές αυτές απόδοσης λειτουργούν ως μία ‘ὑπογραφή” ανά εφαρμογή και δείχνουν πως κάθε εφαρμογή συμπεριφέρεται κατά την εκτέλεσή της στην πάροδο του χρόνου. Πιο συγκεκριμένα, η ‘ὑπογραφή” αυτή αποτελείται από 35 διαφορετικά σήματα τα οποία αντιστοιχούν σε διαφορετικές χαμηλού επιπέδου, όπως για παράδειγμα εντολές ανά κύκλο ρολογιού, αστοχίες σε προσβάσεις στην κρυφή μνήμη κ.ά.

Συλλογή δεδομένων για χαρακτηρισμό των παραμέτρων του Spark: Ως δεύτερο βήμα, το *Sparkle* συλλέγει δεδομένα προκειμένου να προσδιορίσει και να χαρακτηρίσει τη συμπεριφορά των εξεταζόμενων εφαρμογών για διαφορετικές ρυθμίσεις των παραμέτρων του Spark. Για το σκοπό αυτό, το πλαίσιό μας παράγει τυχαίους συνδυασμούς παραμέτρων, για όλες τις 101 διαφορετικές παραμέτρους. Συγκεκριμένα, για κάθε παράμετρο επιλέγεται τυχαία μία τιμή από το αντίστοιχο διακριτό πεδίο ορισμού της. Έπειτα, κάθε εφαρμογή από τις εξεταζόμενες εκτελείται στο σύστημα για όλα τα διαφορετικά σύνολα παραμέτρων και για όλα τα διαφορετικά μεγέθη συνόλου δεδομένων εισόδου. Το βήμα αυτό έχει ως αποτέλεσμα ένα σύνολο τιμών απόδοσης ανά εφαρμογή, όπου κάθε τιμή του συνόλου αντιστοιχεί σε διαφορετικές ρυθμίσεις των παραμέτρων του Spark.

Ανάπτυξη και εκπαίδευση μοντέλων βαθιάς μάθησης για πρόβλεψη της απόδοσης των εφαρμογών

Το πλαίσιο *Sparkle* ακολουθεί μια διπλή προσέγγιση για μοντελοποίηση της απόδοσης των εφαρμογών. Με δεδομένη την “ υπογραφή” μιας εφαρμογής καθώς και μία δεδομένη ρύθμιση των παραμέτρων του Spark, προβλέπει αν αναμένεται επιτάχυνση με μία δυαδική λογική ταξινόμησης (ναι/όχι), και σε περίπτωση που αναμένεται επιτάχυνση, εκτιμά την έκτασή της (παλινδρόμηση). Το *Sparkle* παρέχει ένα μοντέλο βαθιάς μάθησης, το οποίο είναι σε θέση να μοντελοποιεί και τα δύο παραπάνω προβλήματα (ταξινόμηση και παλινδρόμηση), η αρχιτεκτονική του οποίου φαίνεται στο Σχήμα 22. Συγκεκριμένα:

- *Signature Temporal Encoder:* Αυτό το κομμάτι αναλύει την ‘ὑπογραφή” των εφαρμογών



Σχήμα 22.. Αρχιτεκτονική του νευρωνικού δικτύου για καθολική μοντελοποίηση της απόδοσης των εφαρμογών Spark

και την μετασχηματίζει σε ένα διάνυσμα συγκεκριμένου και σταθερού μήκους.

- *Configuration Encoder:* Αυτό το κομμάτι του δικτύου έχει σκοπό να αναλύει τις παραμέτρους του πλαισίου Spark και να προσδιορίζει ποιες παράμετροι είναι πιο σημαντικές και επηρεάζουν την απόδοση των εφαρμογών.

- *Classification & Regressions Heads:* Τέλος, τα δύο αυτά κομμάτια έχουν την ίδια δομή και έχουν σκοπό να πραγματοποιήσουν την τελική πρόβλεψη σχετικά με την ταξινόμηση και παλινδρόμηση αντίστοιχα.

Γενικά, η προταθείσα αρχιτεκτονική εκπαιδεύεται συνδυάζοντας πληροφορία σχετικά τόσο με τη συμπεριφορά εκτέλεσης των εφαρμογών όσο και με τη ρύθμιση των παραμέτρων. Επίσης, χαρακτηρίζοντας κάθε εφαρμογή με βάση τις μετρικές χαμηλού επιπέδου, είμαστε σε θέση να μπορούμε δυνητικά να μοντελοποιούμε και άγνωστες εφαρμογές, καθώς πολλές από αυτές θα παρουσιάζουν παρόμοια συμπεριφορά με τις εφαρμογές που χρησιμοποιήσαμε στη διαδικασία της εκπαίδευσης. Επίσης, ενώ το μέγεθος του συνόλου δεδομένων εισόδου δε δίνεται ρητά ως είσοδος στο μοντέλο, αυτό μπορεί να εξαχθεί μέσω της υπογραφής της εφαρμογής, αφού π.χ. μεγαλύτερα σύνολα δεδομένων εισόδου οδηγούν σε μεγαλύτερο μέγεθος σημάτων των μετρικών απόδοσης χαμηλού επιπέδου αντίστοιχα.

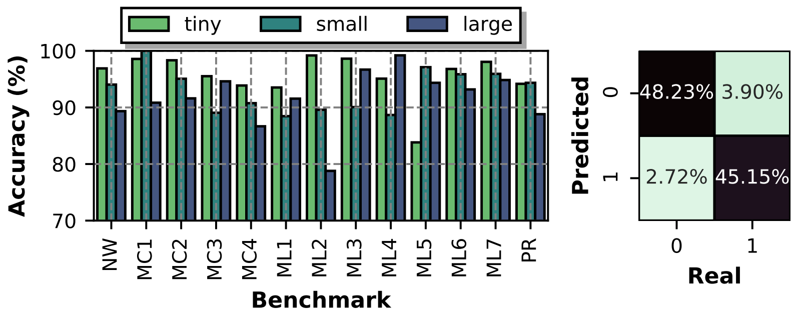
Φάση αυτοματοποιημένης ρύθμισης παραμέτρων και εκτέλεση εφαρμογών

Στη φάση αυτή, το *Sparkle* στοχεύει στο να βελτιστοποιήσει τις παραμέτρους του Spark για εφαρμογές που πρόκειται να εκτελεστούν στην υπαρχουσα υποδομή. Σε περίπτωση άγνωστων εφαρμογών, το *Sparkle* τις εκτελεί χρησιμοποιώντας τις προκαθορισμένες ρυθμίσεις παραμέτρων και καταγράφει και αποθηκεύει τις αντίστοιχες μετρικές απόδοσης χαμηλού επιπέδου που χρησιμοποιούνται ως ῾υπογραφή" τους. Στη συνέχεια, το πλαίσιο ρωτά επαναληπτικά τα μοντέλα πρόβλεψης της απόδοσης για διαφορετικές τιμές παραμέτρων, προκειμένου να προσδιορίσει μια βέλτιστη διαμόρφωση.

Το *Sparkle* έναν μετα-ευρετικό τρόπος για βελτιστοποίηση, προκειμένου να εξερευνήσει αποτελεσματικά τον υποκείμενο χώρο παραμέτρων και να εξασφαλίσει τη σύγκλιση προς τις βέλτιστες λύσεις. Συγκεκριμένα, το *Sparkle* ακολουθεί μια προσέγγιση ανοικτού βρόχου, όπου η διαδικασία βελτιστοποίησης εφαρμόζεται πάνω στο εκπαιδευμένο μοντέλο ΔNN για την αξιολόγηση της δυναμικής του χώρου λύσεων. Αυτή η προσέγγιση οδηγεί σε ένα σύνολο βέλτιστων λύσεων κατά Pareto [374], οι οποίες παρουσιάζουν μία αντιστρόφως ανάλογη σχέση μεταξύ της απόδοσης (χρόνο εκτέλεσης) και του συνολικού κόστους που απαιτείται για την εκτέλεση (σε χρήματα). Σχετικά με τον αλγόριθμο βελτιστοποίησης, το *Sparkle* χρησιμοποιεί τον αλγόριθμο NSGA-II για να διασχίσει το χώρο λύσεων, λόγω της ικανότητάς του να ξεφεύγει από τοπικά βέλτιστα και να παρέχει γρήγορη σύγκλιση σε αποδοτικές λύσεις [6, 261, 375]. Ο NSGA-II είναι ένας εξελικτικός αλγόριθμος και λειτουργεί για έναν αριθμό γενεών, όπου σε κάθε γενιά δίνεται η ευκαιρία στις ελίτ ενός πληθυσμού να μεταφερθούν στην επόμενη. Με βάση τους τελεστές μετάλλαξης και διασταύρωσης, δημιουργεί νέους πληθυσμούς απογόνων που θα εξεταστούν στις επόμενες γενιές. Προκειμένου να προσδιοριστεί ένα βέλτιστο σύνολο τιμών όσον αφορά τις υπερπαραμέτρους της NSGA-II (μέγεθος πληθυσμού, γενιές, πιθανότητα μετάλλαξης και ρυθμός διασταύρωσης), διερευνούμε την επίδραση κάθε υπερπαραμέτρου του αλγορίθμου NSGA-II στο τελικό μέτωπο Pareto.

5.4. Πειραματική Αξιολόγηση

Σε αυτήν την ενότητα δείχνουμε κάποια ενδεικτικά αποτελέσματα σχετικά με την αποδοτικότητα και αποτελεσματικότητα του πλαισίου *Sparkle*. Συγκεκριμένα οι άξονες που εξετάζουμε είναι το κατά πόσο μπορεί το πλαίσιο *Sparkle* να i) πραγματοποιήσει ακριβείς προβλέψεις σχετικά με τον εκτιμώμενη κλάση καθώς και το χρόνο εκτέλεσης των εφαρμογών για διαφορετικές ρυθμίσεις των παραμέτρων του Spark και ii) να καταλήξει σε πιο αποδοτικές λύσεις (τόσο από άποψη χρόνου εκτέλεσης όσο και από άποψη κόστους) συγκριτικά με την προκαθορισμένη ρύθμιση παραμέτρων του Spark. Προκειμένου να αξιολογήσουμε



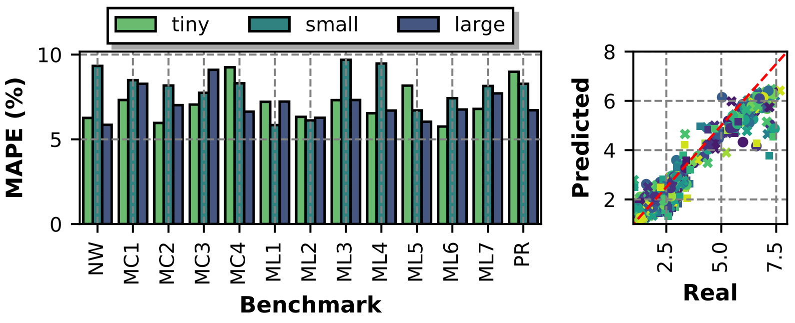
Σχήμα 23.. *Αριστερά:* Ακρίβεια μοντέλου αναφορικά με το πρόβλημα της ταξινόμησης. *Δεξιά:* Πίνακας Σύγχυσης.

το προτεινόμενο πλαίσιο, χρησιμοποιήσαμε εφαρμογές από τη σουίτα HiBench [2], ενώ τα πειράματά μας διεξήχθησαν σε ένα πραγματικό σύστημα συστάδας, αποτελούμενο από 10 διακομιστές Intel©Xeon©E5-2690 v3.

Ακρίβεια προβλέψεων μοντέλου

Αρχικά, αξιολογούμε την ικανότητα του *Sparkle* να μοντελοποιεί την απόδοση των εφαρμογών για διαφορετικές ρυθμίσεις των παραμέτρων του Σπαρκ. Συγκεκριμένα, μετράμε την ακρίβεια του μοντέλου μας για την πρόβλεψη της *ι)* κατηγορίας απόδοσης (ταξινόμηση) και της *ιι)* πραγματικής επιτάχυνσης που παρέχεται σε περιπτώσεις στις οποίες η ρύθμιση παραμέτρων οδηγεί σε επιτάχυνση (παλινδρόμηση). Προκειμένου να εκπαιδεύσουμε και να αξιολογήσουμε το μοντέλο μας, χωρίζουμε το σύνολο δεδομένων μας σε δύο υποσύνολα, ίσα με 90% (σύνολο εκπαίδευσης) και 10% (σύνολο δοκιμής) των δειγμάτων, όπου το σύνολο δεδομένων δοκιμής αποτελείται από τριπλέτες εφαρμογών-συνόλου δεδομένων-ρύθμισης παραμέτρων που δεν χρησιμοποιήθηκαν κατά την εκπαίδευση.

Ακρίβεια ταξινόμησης: Στο Σχήμα 23 βλέπουμε τα αποτελέσματα αναφορικά με το ποσοστό ακρίβειας του μοντέλου μας στο να προβλέπει την κλάση (επιτάχυνση/επιβράδυνση) στην οποία ανήκει ένας συγκεκριμένος συνδυασμός εφαρμογής-συνόλου δεδομένων-ρύθμισης παραμέτρων. Παρατηρούμε πως για την πλειονότητα των περιπτώσεων, το Sparkle παρέχει εξαιρετικά υψηλές ακρίβειες άνω του 90% ανεξάρτητα από το μέγεθος του εξεταζόμενου συνόλου δεδομένων, δείχνοντας την ικανότητά του να προβλέπει σωστά αν μια δεδομένη ρύθμιση των παραμέτρων είναι πιο αποδοτική ή όχι σε σύγκριση με την προεπιλεγμένη. Παρουσιάζουμε επίσης τον πίνακα σύγχυσης, ο οποίος απεικονίζει το ποσοστό των σωστών/λάθους θετικών/αρνητικών προβλέψεων που έκανε το μοντέλο, όπου το 0 δηλώνει την κατηγορία επιβράδυνσης και το 1 την κατηγορία επιτάχυνσης. Βλέπουμε ότι, συνολικά,



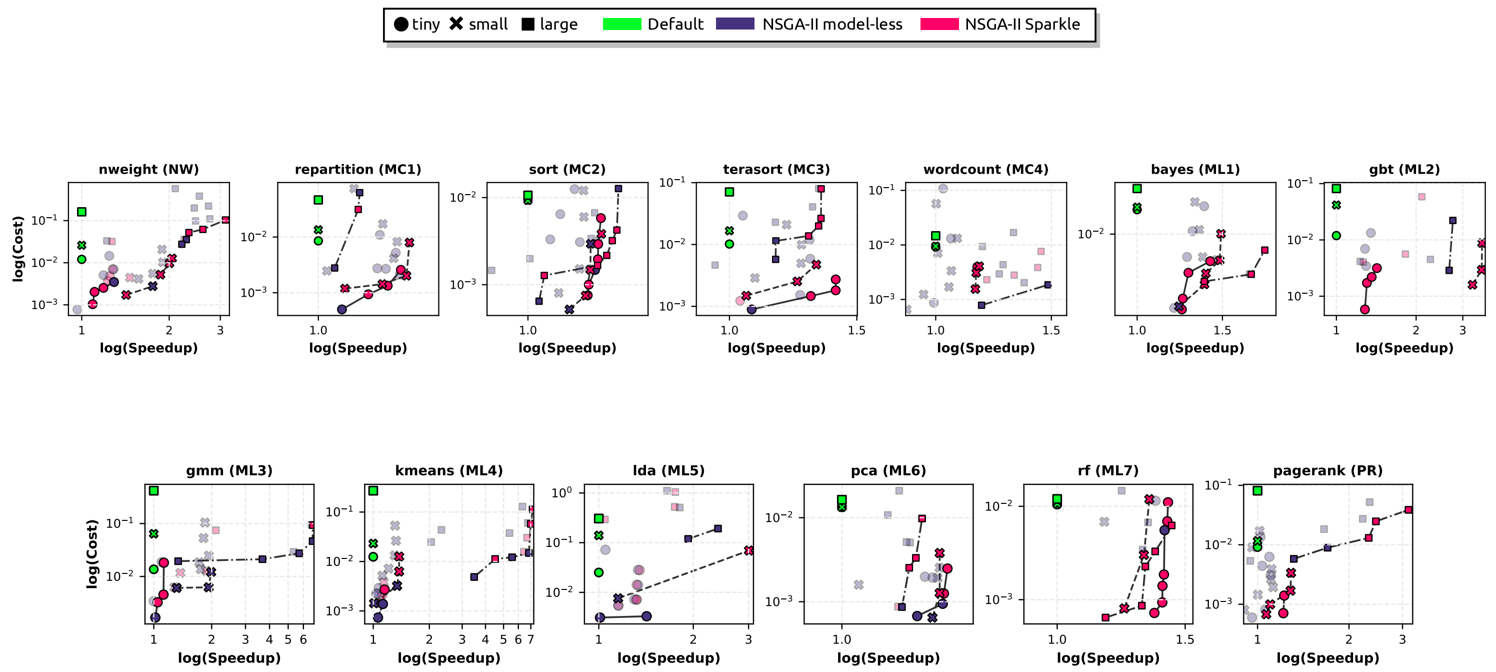
Σχήμα 24.. *Αριστερά:* Ακρίβεια μοντέλου αναφορικά με το πρόβλημα της παλινδρόμησης (MAPE)*Δεξιά:* Προβλεπόμενες τιμές έναντι πραγματικών.

το Sparkle επιτυγχάνει μέση ακρίβεια ≈ 93%. Τέλος, το ποσοστό των ψευδώς αρνητικών προβλέψεων, δηλαδή της λανθασμένης πρόβλεψης ότι μια ρύθμιση παραμέτρων θα παρέχει επιτάχυνση, διατηρείται σε χαμηλά επίπεδα, με περίπου 2.7% των περιπτώσεων να ανήκουν σε αυτή την περίπτωση, γεγονός που δείχνει ότι το Sparkle δεν είναι επιρρεπές σε λανθασμένες προβλέψεις που θα οδηγήσουν σε εκτελέσεις με επιβράδυνση.

Ακρίβεια παλινδρόμησης: Στη συνέχεια, αξιολογούμε την ακρίβεια του μοντέλου πρόβλεψης της επιτάχυνσης του Sparkle, αξιολογώντας το Μέσο Απόλυτο Ποσοστιαίο Σφάλμα (MAPE). Το Σχήμα 24 παρουσιάζει τα αντίστοιχα αποτελέσματα, δείχνοντας ότι το Σπαρκλε παρέχει ισχυρές προβλέψεις ανεξάρτητα από την εφαρμογή και το μέγεθος του συνόλου δεδομένων της, με μέσο όρο 7.2% MAPE συνολικά. Επίσης, εξετάζοντας τις πραγματικές και προβλεπόμενες τιμές επιτάχυνσης, παρατηρούμε ότι το σφάλμα αυτό κατανέμεται ομοιόμορφα σε όλες τις εφαρμογές και τα σύνολα δεδομένων, καθώς τα περισσότερα σημεία βρίσκονται κοντά στη γραμμή παλινδρόμησης 45^o. Τέλος, παρατηρούμε ότι για μεγάλες τιμές επιτάχυνσης (> 5.0), το σύστημά μας υποεκτιμά την πραγματική επιτάχυνση. Μια τέτοια συμπεριφορά είναι αναμενόμενη, δεδομένου ότι το 77% των τιμών επιτάχυνσης είναι κάτω από 2.0, ενώ σχεδόν 84% είναι κάτω από 2.5 και άρα το μοντέλο μας δεν έχει αναλύσει αρκετές δεδομένα κοντά σε αυτές τις επιταχύνσεις κατά τη διαδικασία της εκπαίδευσης, επομένως δεν είναι σε θέση να πραγματοποιήσει ικανοποιητικές προβλέψεις για αυτές τις περιπτώσεις.

Γενετική βελτιστοποίηση

Τέλος, εξετάζουμε το βήμα γενετικής βελτιστοποίησης του *Sparkle*, αξιολογώντας την ικανότητά του να παρέχει αποδοτικές ρυθμίσεις παραμέτρων όσον αφορά την απόδοση και το κόστος ανάπτυξης. Για τον υπολογισμό του κόστους, λαμβάνουμε υπόψη την



Σχήμα 25.. Λύσεις Pareto οι οποίες προτάθηκαν από *i)* γενετικό αλγόριθμο NSGA-II χωρίς την ύπαρξη μοντέλου πρόβλεψης και *ii) Sparkle*. Τα σημεία με εμφανή χρωματισμό δείχνουν την τελική καμπύλη Pareto η οποία περιλαμβάνει τα μη κυριαρχούμενα σημεία των δύο προσεγγίσεων. Τα πράσινα σημεία δείχνουν τις εκτελέσεις με την προεπιλεγμένη ρύθμιση των παραμέτρων του Spark.

τιμολόγηση 15 διαφορετικών τύπων διακομιστών από την πλατφόρμα Υπολογιστικού Νέφους AWS, με διαφορετικά χαρακτηριστικά όσον αφορά τους εικονικούς πυρήνες και την προσφερόμενη χωρητικότητα μνήμης. Συγκεκριμένα, υπολογίζουμε το κόστος κάθε ανάπτυξης εντοπίζοντας τη λιγότερο δαπανηρή περίπτωση η οποία να ικανοποιεί τον αριθμό των πυρήνων του κάθε executor και τις απαιτήσεις του σε μνήμη. Στη συνέχεια, πολλαπλασιάζουμε αυτήν την τιμή με το συνολικό αριθμό των executors και τον συνολικό χρόνο εκτέλεσης της αντίστοιχης εφαρμογής.

Sparkle έναντι προεπιλεγμένης ρύθμισης παραμέτρων: Πρώτον, αξιολογούμε την επιτάχυνση και τη μείωση του κόστους που προσφέρει το *Sparkle* σε σύγκριση με την προεπιλεγμένη ρύθμιση παραμέτρων του Spark. Συνολικά, το *Sparkle* προσφέρει λύσεις που διατηρούν τα έξοδα εκτέλεσης των εφαρμογών μεταξύ 0.5% υψηλότερα (θυσιάζοντας το κόστος για επιτάχυνση) έως και 54% χαμηλότερα, με μέσο όρο κέρδους κόστους 13%, σε σύγκριση με την προεπιλεγμένη εκτέλεση.

Όσον αφορά την αύξηση της ταχύτητας, το *Sparkle* επιτυγχάνει επιτάχυνση που κυμαίνεται από ×1.05, στην περίπτωση ζευγών εφαρμογών/συνόλων δεδομένων που δεν επηρεάζονται εγγενώς από τον συντονισμό παραμέτρων, έως ×6.8, με μέσο όρο αύξησης ταχύτητας ×1.72 σε όλες τις εξεταζόμενες περιπτώσεις.

Sparkle έναντι γενετικής βελτιστοποίησης χωρίς μοντέλο πρόβλεψης: Τέλος, συγκρίνουμε το σύνολο των λύσεων Pareto που παρέχει το πλαίσιό μας με μια προσέγγιση γενετικής βελτιστοποίησης χωρίς μοντέλο, όπου κάθε προτεινόμενη λύση ανά γενιά (διαφορετική ρύθμιση σε παραμέτρους του Spark) εκτελείται στο σύστημα συστάδας και λαμβάνονται οι πραγματικές τιμές χρόνου εκτέλεσης και κόστους. Θέτουμε το κριτήριο τερματισμού του αλγορίθμου βελτιστοποίησης στην 1 ώρα, επιτρέποντας στην προσέγγιση χωρίς μοντέλο να αξιολογήσει έναν επαρκή αριθμό διαφορετικών ρυθμίσεων των παραμέτρων του Spark.

Στο Σχήμα 25 παρουσιάζονται τα αποτελέσματα ανά σημείο αναφοράς, όπου τα σημεία με μπλε χρώμα αποκαλύπτουν τις λύσεις Pareto που προτείνονται από τον βελτιστοποιητή χωρίς μοντέλο και με ροζ χρώμα αυτές που προτείνονται από το *Sparkle*. Στην περίπτωση του *Σπαρκλε*, εκτελούμε στο σύστημά μας το τελικό σύνολο λύσεων Pareto που προτείνεται, για να λάβουμε την πραγματική απόδοση ανά διαφορετική ρύθμιση των παραμέτρων και όχι την εκτίμηση του μοντέλου. Με εμφανή χρωματισμό, υποδεικνύουμε τα τελικά μέτωπα Pareto, τα οποία περιλαμβάνουν τις μη κυρίαρχες λύσεις που παρέχονται από κάθε προσέγγιση βελτιστοποίησης. Παρατηρούμε πως το *Sparkle* υπερισχύει έναντι της προσέγγισης βελτιστοποίησης χωρίς μοντέλο, καθώς η πλειονότητα των σημείων που βασίζονται στο τελικό μέτωπο Pareto ανήκει στο δικό της σύνολο προτεινόμενων λύσεων. Συγκεκριμένα, το πλαίσιο *Sparkle* καλύπτει περίπου το 65% του τελικού μετώπου Pareto, ενώ το υπόλοιπο 35% ανήκει στην εγγενή προσέγγιση.

5.5. Επίλογος

Στο κεφάλαιο αυτό παρουσιάσαμε το *Sparkle*, ένα αυτοματοποιημένο πλαίσιο ρύθμισης των παραμέτρων του Spark, το οποίο βασίζεται σε μοντέλα βαθιάς μηχανικής μάθησης για τη μοντελοποίηση και πρόβλεψη του χρόνου εκτέλεσης των εφαρμογών. Συγκριτικά με προηγούμενες προσπάθειες μοντελοποίησης της απόδοσης των εφαρμογών Spark, το *Sparkle* παρέχει μια καθολική προσέγγιση μοντελοποίησης της απόδοσης καθώς και επεκτείνεται σε ολόκληρο το χώρο παραμέτρων, εξαλείφοντας έτσι πλήρως την ανάγκη για ανθρώπινες ή στατιστικές προσεγγίσεις για τον προσδιορισμό της σημασίας κάθε παραμέτρου.

6. Επίλογος και μελλοντικές επεκτάσεις

Σήμερα, βρισκόμαστε σε μια κομβική στιγμή για τη δημιουργία και τον μετασχηματισμό ενός ψηφιακού κόσμου, με το υπολογιστικό νέφος να αποτελεί ζωτικό πυλώνα προς αυτήν την κατεύθυνση. Από τη σκοπιά των παρόχων υπηρεσιών υπολογιστικού νέφους, η ελαχιστοποίηση του συνολικού κόστους ιδιοκτησίας των υποδομών τους χωρίς να θυσιάζεται η ποιότητα των υπηρεσιών που προσφέρονται στους πελάτες αποτελεί ύψιστη προτεραιότητα. Ταυτόχρονα, από την πλευρά των τελικών χρηστών, ο απώτερος στόχος είναι η μεγιστοποίηση της απόδοσης των εφαρμογών με ταυτόχρονη μείωση του λειτουργικού κόστους τιμολόγησης. Προκειμένου να ικανοποιηθούν και οι δύο κόσμοι, η αποτελεσματική διαχείριση των πόρων του νέφους είναι απαραίτητη, έτσι ώστε να μπορούμε να χρησιμοποιούμε στο έπακρο τους διαθέσιμους υπολογιστικούς πόρους. Προς την κατεύθυνση της δημιουργίας πιο αποδοτικών πλατφορμών Υπολογιστικού Νέφους, η διαχείριση πόρων με βάση τεχνικές μηχανικής μάθησης εμφανίζεται ως μια εξέχουσα λύση, ικανή να χειριστεί και να διαχειριστεί την τεράστια πολυπλοκότητα τέτοιων συστημάτων. Ωστόσο, ερωτήματα όπως ¨πώς¨, ¨πότε' και ¨πού' είναι καλύτερο να ενσωματωθεί η ΜΛ στη διαχείριση πόρων του νέφους είναι ακόμη ασαφή.

Στην παρούσα διατριβή, εξετάσαμε την εφαρμογή τεχνικών βαθιάς μάθησης για τη βελτιστοποίηση της απόδοσης και της αποδοτικότητας των πόρων σε συστήματα Υπολογιστικού Νέφους. Διερευνήσαμε την αποδοτικότητα βαθιών νευρωνικών δικτύων σε διαφορετικά επίπεδα βελτιστοποίησης, πιο συγκεκριμένα στον τομέα της προβλεπτικής παρακολούθησης, καθώς και στη βελτιστοποίηση σε επίπεδο συστήματος, συστάδας καθώς και στο επίπεδο της εφαρμογής. Συγκεκριμένα, η εργασία μας μπορεί να συνοψιστεί ως εξής: ⋄ Όσον αφορά την παρακολούθηση των συστημάτων Υπολογιστικού Νέφους, αναπτύξαμε το *Rusty*, ένα πλαίσιο προβλεπτικής παρακολούθησης με επίγνωση των παρεμβολών για συστήματα Υπολογιστικού Νέφους πολλαπλών μισθωτών, ⋄ Όσον αφορά τη διαχείριση σε επίπεδο συστάδας, παρουσιάσαμε το *Adrias*, ένα πλαίσιο ενορχήστρωσης πόρων για συστήματα αποσυντεθειμένης μνήμης. ⋄ Όσον αφορά τη βελτιστοποίηση σε επίπεδο εφαρμογής, παρουσιάσαμε το *Sparkle*, ένα πλαίσιο αυτόματης ρύθμισης των παραμέτρων του πλαισίου Spark το οποίο βασίζεται σε βαθιά νευρωνικά δίκτυα προκειμένου να μοντελοποιεί την απόδοση και το κόστος των υπό εκτέλεση εφαρμογών.

Ως μελλοντικές επεκτάσεις της παρούσας διατριβής, τα προτεινόμενα πλαίσια θα μπορούσαν να επεκταθούν προς τρεις βασικούς άξονες. Πρώτον, στο κομμάτι της "Πράσινης Πληροφορικής και Ενεργειακής Βιωσιμότητας", όλα τα πλαίσια μπορούν να επεκταθούν έτσι ώστε να λαμβάνουν επιπλέον υπόψη την κατανάλωση ενέργειας, πέρα από την απόδοση των εφαρμογών. Δεύτερον, τα προτεινόμενα πλαίσια θα μπορούσαν να επεκταθούν προκειμένου να μπορούν να εφαρμοστούν για να βελτιστοποιούν τη διαχείριση πόρων σε περιβάλλοντα Edge-Cloud τα οποία είναι πιο αποκεντρωμένα και παρουσιάζουν μεγαλύτερη πολυπλοκότητα όσον αφορά το κομμάτι της δρομολόγησης των εφαρμογών, καθώς, εκτός των άλλων, παρουσιάζονται περαιτέρω προβλήματα, όπως π.χ. παρεμβολές στο δίκτυο, απομακρυσμένη αποθήκευση δεδομένων, κ.ά. Τέλος, από τη σκοπιά της Τεχηνητής Νοημοσύνης και της Μηχανικής Μάθησης, μπορούμε να εξετάσουμε εναλλακτικές αρχιτεκτονικές προκειμένου να μοντελοποιούμε τα διαφορετικά προβλήματα. Ιδιαίτερο ενδιαφέρον παρουσιάζουν οι αρχιτεκτονικές μοντέλων που βασίζονται στην ανάλυση φυσικής γλώσσας, όπως για παράδειγμα το μοντέλο GPT [387]. Με γνώμονα τέτοιες αρχιτεκτονικές θα μπορούσαμε να κατασκευάζουμε μοντέλα τα οποία να είναι σε θέση να μοντελοποιούν την απόδοση των εφαρμογών κατευθείαν μέσα από τον πηγαίο κώδικά τους, χωρίς να χρειάζεται η πολυδάπανη διαδικασία της συλλογής δεδομένων και χαρακτηρισμού των εφαρμογών.

Contents

List of Figures

List of Tables

Chapter 1.

Introduction

Today, we are going through the "Digital Era" or "Digital Revolution", where, the pace at which technology is evolving is ever increasing across the globe. This digital tsunami is driven by a culmination of technologies and socioeconomic factors and trends, including, but not limited to, the rapid advancements in the field of networking and connectivity [9], the explosion in the number of connected devices on the internet [10], the huge amount of data generated by these devices and the valuable insights that one can derive from them [11] as well as the need for rapid and agile reactions in case of natural or man-made disasters or crisis situations [12].

Businesses and organizations from all sectors are embracing this digital transformation in nearly every aspect of business, transforming old processes and methods of human interaction and creating new innovations within the digital business culture[1]. On top of that, social phenomena such as the COVID-19 pandemic only expedite this transition, forcing businesses of all kinds to employ innovative digital solutions to survive[2,3]. In fact, according to a recent report, by the end of 2019, 70% of businesses had realized a digital transformation plan[4], encompassing 40% of technology expenditures.

Cloud computing has been, and still is, a key enabler towards sustaining this ever-increasing demand for computing resources [13, 14]. With Cloud computing, companies can make profit from cost-effective, scalable solutions to the various needs of their information technology (IT) part, thus, boosting their overall business. Moreover, it has shown to be a vital part for other new technologies such as artificial intelligence [15, 16] and the internet of things [17, 18], which are projected to be critical in the future of innovation during the 21st-century.

[1]How Cloud Migration & Digital Transformation Are Driving the Digital Revolution
[2]The Rise of the Hybrid Workplace
[3]How The Pandemic Has Accelerated Cloud Adoption
[4]Digital transformation research report 2018: Strategy, returns on investment, and challenges

1.1. The role of Cloud computing in the Digital Era

Cloud computing refers to the on-demand delivery of computing services including computing power (servers), storage capacity, databases, networking, analytics, and many others [19] over the Internet ("Cloud") offering faster innovation, flexible and agile resources, thus, realizing economies of scale [20]. The rationale behind its inception is simple: *Information and communication technology (ICT) providers with available but unused compute and storage capacity provide them to IT consumers, who need computing resources but do not acquire them.*

While it may seem that the Cloud computing paradigm has been developed during the last decade, the origin of ideas related to it can actually be traced back to around the 1960s, just when also the Digital Era began. Figure 1.1 shows significant events during these years that contributed to the formation of Cloud computing as we know it today[5].

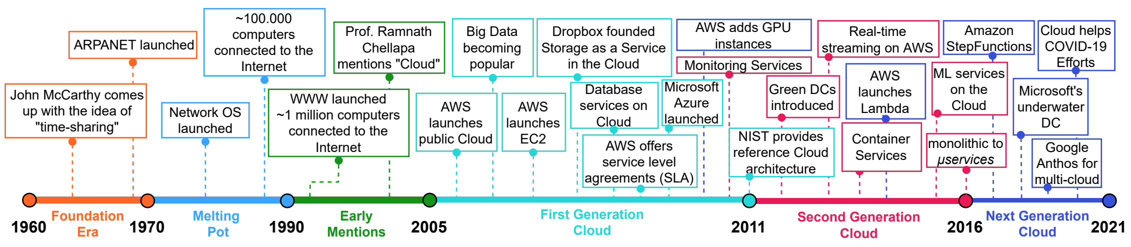
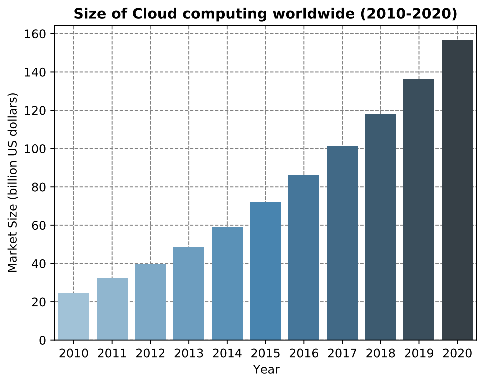

John McCarthy comes up with the idea of "time-sharing"
ARPANET launched
Network OS launched
~100.000 computers connected to the Internet
WWW launched ~1 million computers connected to the Internet
Prof. Ramnath Chellapa mentions "Cloud"
AWS launches public Cloud
Big Data becoming popular
AWS launches EC2
Dropbox founded Storage as a Service in the Cloud
Database services on Cloud
AWS offers service level agreements (SLA)
Microsoft Azure launched
AWS adds GPU instances
Monitoring Services
NIST provides reference Cloud architecture
Green DCs introduced
Real-time streaming on AWS
AWS launches Lambda
Container Services
ML services on the Cloud
monolithic to μservices
Amazon StepFunctions
Microsoft's underwater DC
Cloud helps COVID-19 Efforts
Google Anthos for multi-cloud
1960
Foundation Era
1970
Melting Pot
1990
Early Mentions
2005
First Generation Cloud
2011
Second Generation Cloud
2016
Next Generation Cloud
2021

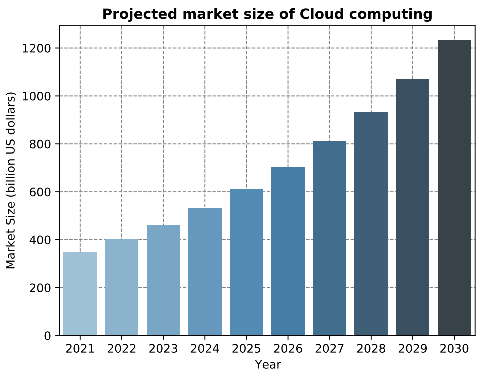
Figure 1.1.. History of Cloud computing[5]

In fact, one of the core ideas of Cloud computing (i.e., *multi-tenancy*) can be directly mapped to the idea of *time-sharing* proposed in the early 1960's, in which a computing resource can be shared among many users at the same time. With time-sharing acting as the foundation of Cloud computing, a chain of events (such as the invention of Internet and World Wide Web) led to first generation Cloud in around 2005, where centralised infrastructures in datacentres are utilized to host a lot of compute and storage resources. In this period of time, Amazon was the first one launching its first public cloud services[6]. After that, we observe a cataclysm of events taking place, leading to the second (2012-2017) and next (2017-today) generation of Cloud, where Clouds are becoming extremely heterogeneous (introduction of GPU/FPGA/TPU accelerators as services in the Cloud), applications are shifting from traditional monoliths to microservices or standalone functions, the paradigm of the IoT-to-edge-to-Cloud computing continuum is introduced and many others.

[5]History of the cloud
[6]Amazon Web Services Launches



(a) Size of the cloud computing market worldwide from 2010 to 2020[7]



(b) Projected size of the cloud computing market worldwide from 2021 to 2030[8]

Figure 1.2.. Worldwide market size of Cloud computing

As expected, Cloud computing has formed and still does a vital pillar in the context of digital transformation, as it allowed stakeholders to quickly digitize their business, without the need to make large upfront investments in hardware and to spend a large amount of time managing this hardware. The rapid adoption of this new paradigm is evident and highlighted by recent market reports. In fact, as shown in Figure 1.2a, over the last 10 years, the size of the cloud computing market has increased more than 600%[7] and was valued at USD 368.97 billion in 2021, while, as shown in Figure 1.2b, is anticipated to expand at a compound annual growth rate (CAGR) of 15.7% from 2022 to 2030[8]. On top of that, this development is fueled by the equivalent explosive growth of kin fields, such as the Internet of Things (approximately 80 billion connected devices by 2025[9]) and Machine Learning (expected to grow from USD 21.17 billion in 2022 to USD 209.91 billion by 2029, at a CAGR of 38.8%[10]), which rely on Cloud backend for data offloading, processing and analysis [21–23], thus further increasing the density of demand for Cloud services.

[7]Size of the cloud computing and hosting market market worldwide from 2010 to 2020
[8]Cloud Computing Market Size, Share & Trends Analysis Report
[9]IoT platforms: enabling the Internet of Things
[10]Machine Learning (ML) Market Size, Share & COVID-19 Impact Analysis

1.2. Improving capacity of Cloud infrastructures

Hyperscale datacenters, with their huge processing and storage capacity, form today's de-facto processing backbone engine in the cloud, that transforms the ever increasing diversity and amount of data into value for numerous applications. Despite their dominance, datacenter infrastructures pose a plethora of open issues. *Capacity* and *Scalability* form two of the most challenging problems within datacenter facilities. Capacity refers to the total amount of applications that can be hosted by a single server (or DC in general), whereas scalability refers to the ability of a datacenter to increase or decrease resources as needed to meet changing request. For example, nowadays, it is common for specific applications, such as search engines and social networks, to serve millions of people[11,12]. To enable this explosive growth, providers have to expand and scale their compute capacity accordingly. Increasing the scale of datacenter can be realized in two ways: *i)* by reducing the cost expenses of commodity hardware and, thus, being able to host more machines with the same operational expenses (OPEX) or *ii)* by increasing its resource efficiency and, thus, being able to host more application on the same number of machines.

Cost efficiency has been traditionally achieved by the replacement of specialized machines with cheap, commodity hardware, which can be easily replaced, replicated and scaled according to the operator's needs [13]. A governing principle of commodity computing is that it is preferable to have more low-performance, low-cost hardware working in parallel (scalar computing) than to have fewer high-performance, high-cost hardware items, thus increasing the overall performance per dollar of the underlying infrastructure[14]. However, as the majority of datacenters have already transitioned to the replacement of specialized with commodity servers, this workaround is reaching an expiration point, thus ending up to be incapable of providing further advantages. Consequently, to maintain scalability, operators have either to build more datacenters, or depend on microprocessors manufacturers to deliver chips with higher performance. However, even such solutions are not viable, with the former requiring huge capital expenditures (CAPEX), long production periods [24, 25] and are soon to be restricted by environmental laws[15] and the latter being confined by the design limitations of modern integrated circuits, such as the end of Dennard scaling [26] and the projected expiration of Moore's Law [27] by 2025[16].

[11]Google - Statistics & Facts

[12]Number of monthly active Facebook users worldwide as of 1st quarter 2022

[13]How to understand the rise of commodity servers in the Cloud

[14]Google data centers

[15]Reading the runes: EU data center regulations are coming sooner than you think

[16]After Moore's Law

On the other hand, resource efficiency can be achieved by making better use of the resources inside the datacenter's facilities. Since datacenter infrastructures are very expensive to build and operate, providers have a strong incentive to optimize their use [28]. Moreover, improving the resource efficiency of a datacenter also helps in reducing the power usage effectiveness (PUE), which forms a top priority of modern cloud operators[17]. Nevertheless, with the tremendous complexity introduced in modern datacenters, operating such infrastructures in an efficient manner is extremely challenging. This complexity originates from various elements that operators have to deal with, both from a HW/SW as well as supply/demand standpoint. A typical example is the conflicting requirements between operators and end-users, where the latter demand their workloads to be given enough resources to achieve high Quality-of-Service (QoS), such as low latency and high throughput for user-facing services or quick analytics execution times and the former anticipate high resource use in order to accommodate as many tasks as feasible with a certain set of resources. While better resource efficiency can be achieved by applying optimizations on many levels of the stack (from system to cluster to application level), state-of-the-art resource management frameworks are not able to handle the complexity of modern Cloud infrastructures. As a consequence, datacenters operate at high underutilization levels as operators tend to sacrifice computational power for better performance [29, 30].

To overcome the issue of underutilization, several workarounds have been proposed in the past, such as consumption-based pricing [31,32] targeting over-provisioning issues, serverless architectures [33,34] targeting the issue of idle reserved resources, and others. Besides that, the *multi-tenant architecture* (also known as *multi-tenancy*) has been a core idea to enhance resource efficiency. In a multi-tenant environment, different users are sharing the same computing resources, thus, increasing the overall utilization of the underlying infrastructure. Each tenant's data and workloads remain isolated (mainly through virtualization mechanisms [35, 36]), even if they happen to run on the same physical machine or group of machines. While multi-tenancy offers reduced costs for operators, greater flexibility and increased efficiency, it also introduces performance loss due to *interference* in shared resources of the system. Interference can exist in multiple levels of the underlying hardware, including CPU, caches, memory bus, storage and network [37–40]. In fact, prior work has shown that interference-unaware schedulers can introduce up to two times less performance for certain workloads [41].

Going one step further, a proposed prominent solution, that has appeared lately in the foreground, for mitigating interference and enabling more fine-grained organization of resources within datacenters is completely reconsidering the design of datacenters and shifting towards a novel computing paradigm, known as *hardware disaggregation* [1,42–44]. In

[17]Google Datacenters - Efficiency

a disaggregated data center, CPU, memory, and storage are independent resource blades that are joined by a network fabric, as opposed to the monolithic server method that datacenters are currently using. As a result, datacenters of the future aim to have more flexibility and gains in terms of utilization efficiency and energy consumption. However, despite its clear benefits, the evolution of the underlying physical hardware introduces several new challenges, e.g., accesses through fabric can result in higher memory latency [1], identifying the right type of resource to allocate and others.

From the above, it is evident that even though many solutions have been proposed for improving scalability and resource efficiency within datacenter infrastructures, there is still a lot of space for improvement. Satisfying the contradicting requirements of operators and end-users, as well as determining the level (or levels) at which an optimization should be applied, forms a many-objective optimization problem, which is extremely difficult to solve. If we also consider the introduction of novel architectures (w.r.t. both hardware and software) the complexity explodes. Modern resource management frameworks of top cloud providers and datacenter owners (e.g., Google, Facebook, Azure) used to, or even still, rely on naive scheduling policies and/or static policies to increase the resource efficiency of their infrastructures [45–53], which, however have several shortcomings. First, they're fine-tuned offline using a small number of benchmark workloads as representatives. Threshold-based policies, for example, often involve hand-tuned thresholds that must be applied for a wide range of workloads [54]. Second, static policies often necessitate reactive responses, which can result in unnecessary costs and customer harm. Consider a commonly used policy for scheduling containers into servers, such as best fit[18]. It's possible that some co-located containers will interfere with each other's use of resources (for example, shared cache space), requiring (reactive) re-allocation of resources or even live migrations, which is costly and can result in service downtime.

Concluding, even though accurate and fast performance estimation is a necessity for Cloud system optimization, conventional heuristic-based designs can not guarantee scalability and optimality, especially in the case of the increasingly complicated datacenter infrastructures. As such, it seems natural to move towards more automated and powerful methodologies for computer architecture and system design.

[18]A Brief Analysis on the Implementation of the Kubernetes Scheduler

1.3. Towards ML-driven Cloud resource management

Today, Machine Learning (ML) and Artificial Intelligence (AI) have an impact on almost every element of the globe. ML is being adopted by businesses, small and large on a massive scale and is a huge enabler for analyzing and leveraging insights from huge amounts of data[19]. In this direction, lately we notice signs of the application of ML to computer architecture and systems, which encompasses two meanings: *i)* the reduction of burdens on human experts designing systems manually to improve designers' productivity, and *ii)* the closing of the positive feedback loop, i.e., architecture/systems for ML and simultaneously ML for architecture/systems, forming a virtuous cycle to encourage improvements on both sides [55].

In the context of Cloud computing, automated ML-driven systems are paving the way for more efficient datacenters. With the adoption of ML techniques, operators are able to process, analyze, gain insights and identify correlations within a huge pool of gathered monitored data, which would be impossible solely through human reasoning. In fact, for public cloud providers AI and ML are already part of their datacenter deployment and operations. For example, Google previously explained how it employs DeepMind AI for cooling and how it was able to reduce Power Usage Effectiveness (PUE) by 15% by automating the management of variables such as fans, cooling systems, and windows[20]. Deepmind was also employed by the corporation to predict wind turbine output up to 36 hours ahead of time, which it used to forecast electricity needs for its facilities connected to wind farms[21]. Moreover, latest efforts regarding efficient resource allocation within datacenters are shifting towards the adoption of ML techniques either as prediction tools or as integrated components for resource management [28, 56–59]

While ML/AI is a prominent solution for delivering more efficient datacenters, it is currently unclear what is the best way to integrate ML into cloud resource management. Prior techniques, in fact, vary in a number of ways. For example, as mentioned before, in some circumstances, ML is highly connected with the resource management in certain circumstances, but it is completely distinct in others [28]. Towards this direction, more research is required in order to understand how, where and when to employ ML techniques for improving resource efficiency in Cloud systems.

[19]51 Machine Learning Statistics to Get You Thinking
[20]Google uses DeepMind AI to cut data center PUE by 15%
[21]Google's DeepMind uses AI to predict wind farm output

1.4. Ph.D. Thesis Scope & Contributions

The scope of this thesis is to examine the applicability of deep learning techniques for improving resource efficiency in Cloud infrastructures. While resource efficiency of datacenters can be improved through several components of the underlying hardware and software infrastructure, in this dissertation we give special focus on optimizing the resource allocation and application scheduling in Cloud systems. Given the strong performance relationship between hardware and software, we base our deep learning models on hardware oriented performance counters, thus aiming to project low-level events to higher level metrics of interest. Specifically, the principles that guided our research activities are the following:

1. Given the close relationship between hardware and software, modern Cloud systems should make orchestration decisions taking into consideration the dynamics of both the application and the underlying system. In this direction, we believe that low-level performance events can provide extremely useful insights regarding performance bottlenecks of Cloud systems.

2. We argue that the complexity of modern Cloud systems is enormous, thus, forming naive scheduling/monitoring solutions incapable of handling efficiently this deluge of available data and optimization knobs. In this direction, we examine the efficacy of machine learning and artificial intelligence in several aspects and optimization layers of the cloud domain.

We examine both these principles on different problems in the field of cloud computing, ranging from application-specific up to system-level optimizations. Figure 1.3 presents, in an abstract manner, a high-level overview of the positioning of the contributions of this thesis with respect to the different optimization layers described in Section 2.1.4.

Overall, the major contributions of this thesis are threefold:

- **Rusty:** A novel sophisticated monitoring solution for cloud infrastructures. Rusty leverages Long Short-Term Memory (LSTM) networks to enable fast and accurate resource and energy consumption predictions of systems under interference. Through Rusty, our ambition is to establish predictive monitoring as the de-facto solution for cloud monitoring, aiming to adopt it for guiding proactive resource allocation mechanisms.

- **Adrias:** A monitoring and orchestration framework for memory disaggregated systems. Adrias continuously monitors the underlying system and gathers application-

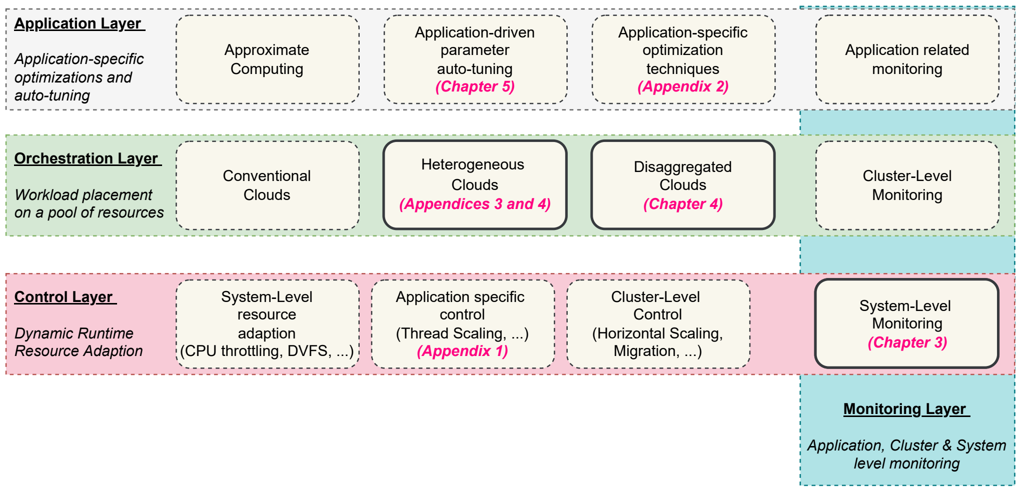

Application Layer
Application-specific optimizations and auto-tuning
Approximate Computing
Application-driven parameter auto-tuning (Chapter 5)
Application-specific optimization techniques (Appendix 2)
Application related monitoring
Orchestration Layer
Workload placement on a pool of resources
Conventional Clouds
Heterogeneous Clouds (Appendices 3 and 4)
Disaggregated Clouds (Chapter 4)
Cluster-Level Monitoring
Control Layer
Dynamic Runtime Resource Adaption
System-Level resource adaption (CPU throttling, DVFS, ...)
Application specific control (Thread Scaling, ...) (Appendix 1)
Cluster-Level Control (Horizontal Scaling, Migration, ...)
System-Level Monitoring (Chapter 3)
Monitoring Layer
Application, Cluster & System level monitoring

Figure 1.3.. Schematic overview of the contributions of this thesis.

and system- wide performance events. By leveraging deep learning approaches, Adrias utilizes the monitoring information to dynamically place incoming applications on memory disaggregated cloud systems.

- **Sparkle:** A deep learning driven parameter autotuning framework for Spark applications. Sparkle leverages a modular DNN architecture along with low-level performance monitoring events and expands to the entire Spark parameter configuration space, thus, completeley eliminating the need for human and/or statistical reasoning in the loop. By employing a genetic optimization approach, Sparkle quickly traverses the parameter design space and identifies optimized Spark configurations.

Besides its major contributions, this thesis also examines four additional problems, i.e., *i)* application placement on heterogeneous Cloud systems, with focus given on cost-efficient deployments on clusters consisting of CPU, FPGA and GPU devices, *ii)* resource-aware workload deployment on Kubernetes clusters equipped with GPU accelerators, *iii)* employment of Natural Language Processing (NLP) techniques for modeling performance of GPU accelerated applications and *iv)* DNN partitioning and offloading in heterogeneous edge computing environments.

In the following subsections, we highlight the major contributions of this thesis and explain how these contributions extend prior state-of-the-art approaches and limitations, as described later in Section 3.

1.4.1. Rusty (Chapter 4)

Addressed problem: *Lack of interference-aware and predictive performance monitoring frameworks to drive proactive scheduling decisions.*

Multi-tenancy forms the de-facto deployment model of modern Cloud systems. Despite its benefits, multi-tenancy leads to interference in shared resources of a system, thus damaging performance of applications. As a result, modern schedulers should be able to dynamically predict per application needs, to perform optimized online resource allocation decisions. While prior research have proposed several monitoring solutions for identifying interference effects that lead to performance degradation, they do so either by *i)* analyzing high-level metrics of interest, e.g. applications' CPU/Memory utilization and logs [60, 61], *ii)* investigating each shared resource seperately in a random-like manner [40], or *iii)* by pausing the execution of running applications and injecting synthetic microbenchmarks for root cause analysis [38, 62]. In order to tackle these inefficiencies, we propose Rusty.

Rusty extends prior by:

- Designing a non-intermittent monitoring solution for Cloud server systems;
- Delivering an extensive, interference-aware analysis, showing the variation of performance events under different interference effects;
- Presenting an in-depth insights regarding the parameters that affect the accuracy and complexity of the its predictive model.
- Providing extremely accurate predictions of hardware performance metrics under interference;
- Imposing minimal performance overhead over running applications on the system.

1.4.2. Adrias (Chapter 5)

Addressed problem: *Absence of resource management frameworks for memory disaggregated Cloud systems.*

Hardware disaggregation is the next big step for efficient and fine-grained management of Cloud infrastructures. While the problem of resource orchestration and application scheduling has been extensively examined in the context of traditional Cloud infrastructures [41, 63, 64], to the best of our knowledge, no prior work has examined its aspect on disaggregated infrastructures. In such composable infrastructures, resource allocation frameworks should be able to take cognitive decisions regarding the the type and topology of resources to be used.

In order to tackle these aspects, we propose Adrias.

Adrias extends prior art by:

- Performing an extensive analysis on the impact of memory disaggregation on state-of-the-art in-memory applications;
- Delivering a set of deep-learning based performance models for applications deployed on top of memory disaggregated Cloud systems;
- Developing a resource orchestration framework responsible for placing applications on memory disaggregated systems with minimal performance overhead.

1.4.3. Sparkle (Chapter 6)

Addressed problem: *Inadequate approaches for performance prediction of Spark applications, which rely on human-in-the-loop or statistical approaches to identify importance of Spark parameters and follow application-specific performance modeling.*

With businesses generating big-data at a rapid pace, analyzing the data to leverage meaningful insights is essential. Apache Spark forms the de-facto big-data analytics framework, as its in-memory processing architecture provides enhanced performance improvements compared to its predecessor Hadoop. Spark provides over 150 parameters that can be configured to alter several aspects of the runtime engine, for further increasing performance. However, manually tuning these parameters is extremely challenging and requires deep knowledge and understanding of the Spark engine. While previous research works have proposed automated frameworks to model performance of Spark applications w.r.t. different configurations, they typically consider only a subset of Spark parameter space [4,6,7] and also rely on application-specific modeling [4, 6], thus, requiring extensive profiling for each new application deployed on the cluster. To overcome these inefficiencies, we propose Sparkle.

Sparkle extends prior art by:

- Providing an in-depth analysis on the impact of all performance-related, tunable Spark parameters found under Spark v3 on different applications and dataset sizes;

- Considering the entire Spark configuration space for modeling performance, hence, not relying to statistical or human reasoning for identifying the importance Spark parameters;
- Delivering a hybrid DNN architecture to exploit both application- and configuration parameter-related characteristics, thus, being able to provide accurate performance predictions of Spark deployments using a single, universal model.

1.5. Thesis Organization

The rest of this thesis is organized as follows:

- Chapter 2 provides a brief background on Cloud computing, reviews the related state-of-the-art research works and pinpoints their limitations. It also presents the positioning of this thesis in the context of Cloud computing and summarizes its major contributions.
- Chapter 4 presents Rusty, a predictive and interference-aware monitoring framework for Cloud systems that leverages Long Short-Term Memory networks for forecasting system events in multi-tenant systems.
- Chapter 5 presents Adrias, a monitoring and orchestration framework that leverages deep learning techniques for placing applications in memory-disaggregated Cloud systems.
- Chapter 6 presents Sparkle, an end-to-end, deep-learning driven, parameter auto-tuning framework for high-dimensional Spark in-memory analytics
- Chapter 7 concludes this thesis by summarizing the presented results and discusses the future extensions of this work.
- Appendix A provides an analysis for cost-effective acceleration for Cloud healthcare analytics, based on the work done in the EU H2020 project AEGLE.
- Appendix B examines the problem of resource-aware scheduling of machine learning inference engines on GPU-enabled Kubernetes infrastructures.
- Appendix C explores the application of natural language processing (NLP) tech-

niques for automatic frature extraction and performance prediction of CUDA-accelerated GPU kernels.

- Appendix D presents a partitioning and offloading framework for DNN inference at the edge/cloud computing continuum.

Chapter 2.

Background on Cloud Computing

In this chapter, we give a brief background on Cloud computing, as well as pinpoint the major challenges that arise inside data-center environments regarding the efficient management of computing resources.

2.1. Background on Cloud computing

Over the last years, the number and size of Cloud infrastructures have experienced a rapid increment [24]. This expansion originates from the fact that more and more stakeholders from different and diverse domains, including but not limited to, healthcare [65], automotive [66] and agriculture [67], are embracing Cloud computing as their de-facto model for application execution. Overall, Cloud computing offers many benefits to adopters, with the most important being:

- *Resource flexibility:* Users can upscale or downscale the capacity of their computing resources on demand, to fit their need, support growth and handle busy periods.
- *Efficiency and Performance:* Users can get applications to market quickly, without worrying about underlying infrastructure management and maintenance, while also taking advantage of high performance, constantly upgraded computing resources, as well as guaranteed service uptimes, through Service-Level-Agreements (SLAs) provided by cloud operators.
- *Cost Reduction:* Users can reduce the total expenses of their company by leveraging pay-as-you-go schemes, while also eliminate upfront costs for hardware and software purchases (CAPEX) and operational costs in terms of energy consumption, cooling and server maintenance (OPEX).

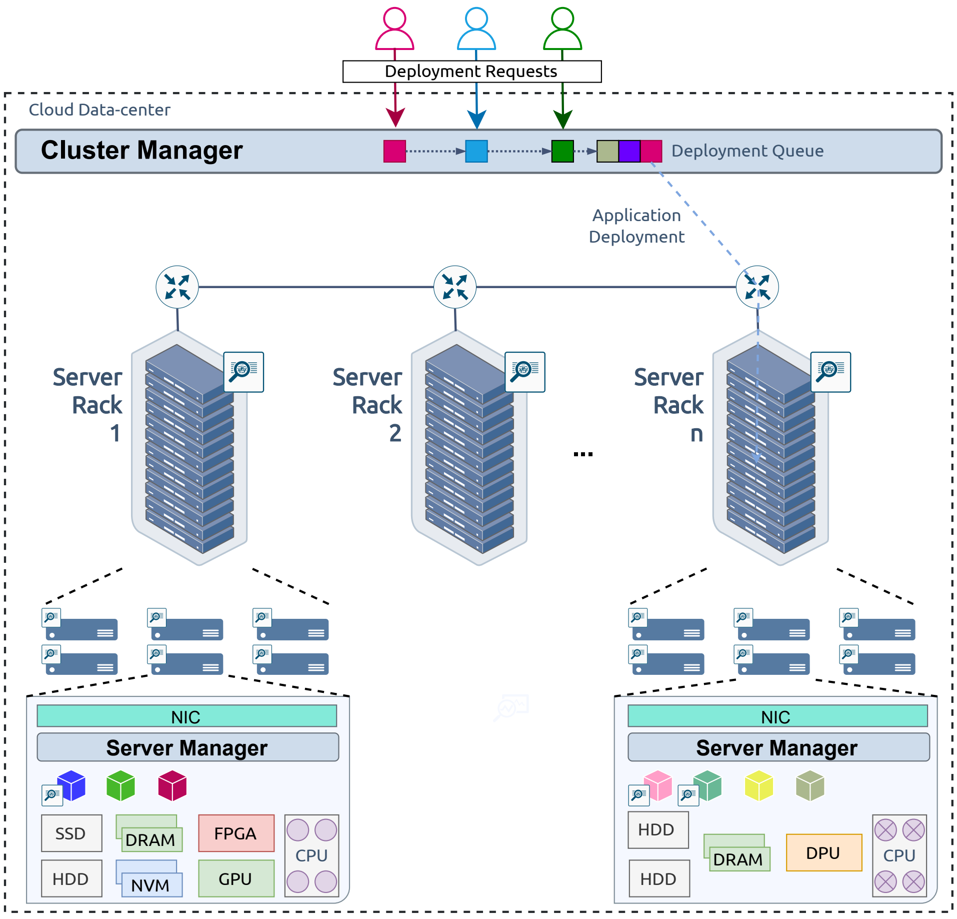


Figure 2.1.. Abstract overview of a conventional datacenter HW/SW infrastructure architecture

Typically, public Cloud services are hosted inside datacenter (DC) infrastructures, which, in turn, house all the required hardware resources to run these services. At its simplest, a datacenter is a physical structure that cloud providers utilize to host their services, deploy end-users' applications and store data. The design of a datacenter is built on a network of computer and storage resources that allow the delivery of shared applications and data. Routers, switches, firewalls, storage systems, servers, and application-delivery controllers are all important components of the design of a datacenter. Figure 2.1 shows a high-level overview of a conventional datacenter's architecture, also depicting in an abstract manner the interrelationship between users' requests, cluster's management components and hardware infrastructure.

2.1.1. Inside a datacenter's hardware architecture

From a hardware perspective, data-centers typically consist of thousands of server racks, each of which hosts a number of physical servers. As shown in Fig. 2.1, servers and racks are usually inter-connected using multi-level routers and network switches [68]. Lately, we observe a dramatic change in the composition of cloud servers inside datacenters[1]. Traditionally homogeneous cloud systems are gradually changing to heterogeneous designs, either through special purpose chips, such as Google's TPUs[2], or reconfigurable fabrics, such as Microsoft's Catapult [69] and Brainwave initiatives [70]. This leads to physical servers presenting extreme heterogeneity in terms of their computing resources, e.g., storage, memory type, equipped accelerators, CPU and others [41, 71]. This hardware diversity is also evident by taking a closer look to the solutions offered by public cloud providers, such as Amazon Web Services (AWS), Google Cloud Platform (GCP), IBM Cloud, Microsoft Azure and others. For example, AWS [72] offers more than 50 different virtualized and/or baremetal instance types, where each instance type includes one or more instance sizes, allowing you to scale your resources to the requirements of your target workload [3].

This heterogeneity is a result of several reasons, some of which being:

1. *The relationship between applications (SW) and server's configuration (HW):* Prior research works have shown that different applications might perform more efficiently on specific CPU architectures and/or different server's composition [41, 62, 73–76]. This is also evident by introspecting AWS's instance types, which are optimized to fit different use cases (e.g., compute/memory/storage optimized).

2. *Hardware-specific requirements of applications:* Modern applications present different demands regarding specialized hardware . For example the rise of the ML and AI domains has pushed DC operators to populate their facilities with heterogeneous GPU accelerators [23, 77, 78]. Another example is modern database applications, which are shifting towards the use of persistent memory solutions (e.g., Intel® Optane$^{\text{TM}}$), which store data in system's memory, plugging directly into the high-speed, low-latency memory bus [79–81].

3. *Constant upgrades of DC's infrastructure:* As a result of the previous points, datacenter providers are gradually replacing the hardware resources of their infrastructure, in order to keep up with the latest advancements in hardware technol-

[1]The Increasing Heterogeneity of Cloud Hardware and What It Means for Systems
[2]Google Cloud Platform - Cloud Tensor Processing Units (TPUs)
[3]https://aws.amazon.com/ec2/instance-types/

ogy innovation and keep up with the demanding requirements of modern applications [24, 73, 82]. Therefore, a DC can accommodate several different generations of servers, accelerators, memory technologies and others [41].

2.1.2. Hosted applications

Today's cloud providers are called to handle and execute a diverse set of applications, such as data analytics, web streaming, web searching, scientific simulations and others [83–87], while also accounting for delivering performance- and cost-efficient solutions. Typically, these applications are divided in two main categories [41,88–91]:

- **Best-effort (BE) applications:** These are basically batch workloads [2, 92, 93] that are not accompanied by strict performance requirements and aim to maximize their computational throughput. Typical examples are stock analytics that are executed once at the end of each day [94], scientific workloads such as genomics sequence analytics [95–97], training of machine learning and deep learning workloads [98, 99], video analytics [100, 101] and others.

- **Latency-critical (LC) applications:** These are user-interactive services, with some examples being web search applications [102], mailing services [103], interactive key-value stores [104] and memory object caching systems [105]. These online services most of the times are coupled with strict latency/response time Quality-of-Service (QoS) constraints expressed in the form of tail latency [106, 107], or specific Service-Level-Agreements (SLAs) for ensuring uptime of the deployed services [4].

2.1.3. Datacenter monitoring

Datacenter Monitoring forms a top priority feature of modern cloud services and infrastructures, as it allows to evaluate the status and performance of the underlying infrastructure and applications on a modular level [108, 109]. Proper monitoring of the whole hierarchy, from application up to system level, is essential to provide insights regarding both the performance of running applications and the load that the cluster's nodes experience, as well as identify possible bottlenecks or failed knobs [61, 110]. Datacenter monitoring can be implemented at various levels, with the most common ones being:

[4]Amazon Compute Service Level Agreement

- **Internal Environment Monitoring:** The datacenter should include enough sensors to detect room temperature, humidity, and other environmental factors, which heavily affect the performance of the underlying hardware infrastructure [111–113].

- **Hardware Monitoring:** Monitoring data regarding energy usage, network bandwidth and speed, storage hardware performance, backup and disaster recovery hardware maintenance, and so on must be collected on a regular basis. These aspects must never be overlooked and must be checked on periodically to ensure a proper functionality.

- **Application Monitoring:** Application monitoring is the practice of analyzing an application's performance, availability, and user experience in order to detect and remedy problems before they affect end-users. Because of the dynamic nature of today's hybrid cloud and cloud native settings, application monitoring is tough. To provide total visibility into application performance, the most effective modern systems combine whole stack monitoring from the front-end, user experience, to the back-end infrastructure [61,109,110,114,115].

2.1.4. Datacenter's mechanisms for resource orchestration

Datacenters host several software components, responsible for performing various activities regarding the management and operation of the underlying infrastructure and applications, including but not limited to orchestration of computing resources [51,116–119], fault tolerance [120–122], security [123–125] and reliability [68,126]. In this thesis, we focus on the topic of efficient orchestration with respect to computing resources. Within cloud clusters, resource orchestration is performed in a multi-level manner, i.e., by applying application-level, cluster-level and system-level optimizations [51]. Figure 2.2 shows a simplified, abstract overview of the orchestration procedure, which consists of three separate components,briefly explained below:

⇒ The Application Optimizer (❶): The purpose of this component is to optimize the deployed application itself in order to be executed more efficiently on the underlying infrastructure. Commonly applications deployed on a Cloud infrastructure are unknown to providers, since end-users can arbitrarily deploy anything to the cluster. However, the latest shift towards the Everything-as-a-Service (XaaS) paradigm (e.g., MLaaS[5] and FaaS[6]), allow Cloud operators to have more fine-grained perception over the workloads running on their infrastructure, which, in turn, allows them to perform specific application-driven

[5]Amazon Comprehend, Amazon Translate, Google Dialogflow
[6]AWS Lambda, Azure Functions, GCP Serverless

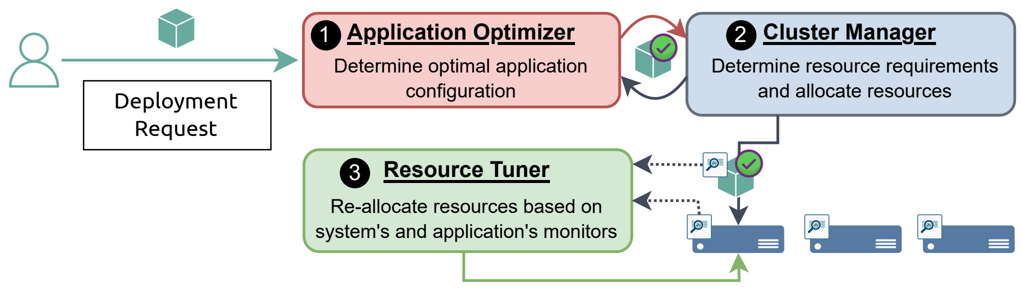

Deployment Request
1 Application Optimizer
Determine optimal application configuration
2 Cluster Manager
Determine resource requirements and allocate resources
3 Resource Tuner
Re-allocate resources based on system's and application's monitors

Figure 2.2.. Resource orchestration and optimization flow within a Cloud cluster.

optimizations.
Optimization and fine-tuning (also referred to as *autotuning*) of deployed applications can be applied on multiple layers, i.e., i) ***code-explicit***, e.g., parallelization, loop organization, memory structures and allocation policies, approximation techniques etc. [127–132], ii) ***code-implicit***, i.e. compiler-level tuning, application's hyperparameter tuning [6, 7, 133, 134], iii) ***heterogeneous code mappings***, e.g. CPU vs. GPU vs. FPGAs [8, 135] and others. Source-code level optimization can be achieved by combining information regarding the execution characteristics of the application itself (e.g., memory access pattern) and the underlying hardware (e.g., cache size). This merge of information allows providers to co-design applications with respect to both HW and SW, thus, delivering solutions tailored to the specifications of the underlying system [8, 127–129]. Moreover, the intrinsic properties (e.g., error resiliency) combined with the predefined key performance indicators (e.g., accuracy percentage of an ML model) of deployed applications may allow for further optimizations, such as employment of approximate computing techniques [130, 136–138], or even enable user-agnostic deployments [139, 140]. Last, a great amount of modern Cloud applications are built over open-source frameworks, which expose a plethora of tuning parameters. Spark [93], Hadoop [141], Pytorch [142] and Tensorflow [143] form representative examples of such frameworks, on top of which end-users develop and deploy big data analytics and machine learning applications. These frameworks offer numerous and complex parameters that control the configuration of the their backend and which can be tuned to enhance performance of applications deployed over them [6, 144–149].

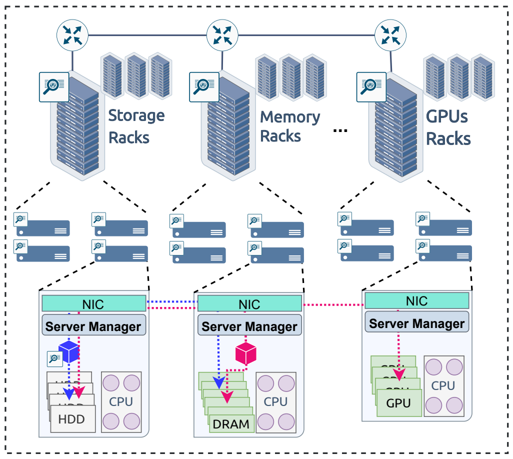
⇒ The Cluster Manager (❷): The main goal of the cluster manager is to allocate resources and schedule incoming applications on the underlying shared infrastructure. Cluster managers may also be responsible for other operations, e.g., performing virtual machine/application migration [150, 151], scaling applications horizontally [152, 153] and others, which, however, are out of the scope of this thesis. The resource allocation process consists of two steps: *i) discover the resource requirements of the deployed application* and *ii) determine the most appropriate set of resources to allocate*, so that the resource

requirements of the application are satisfied and the resource efficiency of the cluster is maximized. Appreciating the resource requirements of an application can be either done by the end-users [29, 154, 155], i.e., application developers measure, specify and reserve exclusively the necessary resources for their application to run properly on specific hardware, or by the cloud operators, i.e., the cluster manager should estimate the required resources of potentially unknown applications deployed on the cluster [41, 116, 156].
In this level, the optimization process can be twofold. First, accurate estimation of the required resources of an application (*resource sizing*) is critical, otherwise over- and under-provisioning issues may arise, which, in turn, affect the resource efficiency of the datacenter [29, 30] and the performance of applications [157, 158] respectively. Second, the cluster manager should be able to also determine the most appropriate node (*resource type*) to deploy the application among the various, heterogeneous servers hosted on the datacenter, while also eliminating the interference effects between applications due to multi-tenant colocation [38, 41, 64, 73, 159, 160].

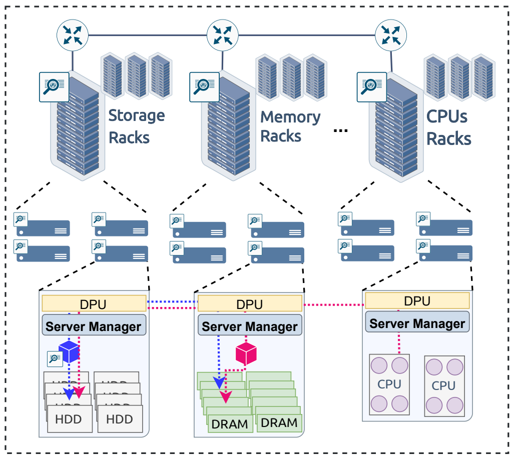
⇒ The Resource Tuner (❸): Finally, the purpose of the resource tuner is to regulate the allocated resources of applications and/or alternate the server's configuration at runtime, in order to face performance issues due to the unpredictable and dynamic behavior of modern applications. This unpredictability and variability in the performance of applications is a result of multiple factors, including but not limited to *i)* the different phases that applications experience throughout their lifetime [161,162,162–164], *ii)* the varying load of interactive web services during the day [30,40,41,88,165] and *iii)* interference due to sharing of resources: Applications deployed on a DC continuously contend for shared resources both in the local host (e.g., CPU cores, processor caches, memory bandwidth, and network bandwidth), as well as for global resources (e.g., network switches and shared file systems) which can lead to severe performance degradation [38,41,106,166]. Thus, the resource tuner should be able to determine such irregular patterns and regulate allocated resources accordingly. Usually, as shown in Fig. 2.2 the resource tuner is implemented as a closed loop system with the application and server monitoring components, where the resource re-allocation is performed in a feedback-based manner, with respect to the historical monitoring data gathered.

2.1.5. Realizing datacenters of the future

As mentioned in Section 2.1.1, traditional datacenters have a relatively static computing architecture, consisting of a number of servers, each with a fixed number of CPUs and RAM and with potentially different types of hardware accelerators. Datacenter operators have used this monolithic server model for years, however, as the variety of hosted applications, the hardware heterogeneity and the adoption of cloud computing increase, so does



(a) Disaggregated DC



(b) Fungible Hyper-Disaggregated DC

Figure 2.3.. Overview of a Disaggregated and Hyper-disaggregated datacenter's architecture

the complexity of operating efficiently such an infrastructure [7]. Typical reasons, among others, are the limited resource utilization of modern infrastructures, the difficulty of integrating new HW devices and the handling of HW failures.

Towards realizing infrastructures of scale, datacenter operators are moving towards a new computing paradigm, referenced in the literature as *hardware resource disaggregation* [42, 44] or *composable hardware infrastructures* [167, 168]. Figure 2.3 shows a simplified overview of the hardware disaggregation concept. As shown in Fig. 2.3a, in a hardware disaggregated system, there is a transformation of general-purpose monolithic servers into network-attached resource pools that may be constructed, managed, and scaled separately. Similar to conventional datacenters (Fig. 2.1), servers are organized inside racks, which, however, are resource-specific (e.g., storage, memory, etc.). Each server on the rack combines a pile of a particular set of resources with CPUs to manage them and Network Interface Cards (NICs) to communicate with other resources on the cluster. Still, the existence of CPUs and NICs in the critical path of the computation and communication weakens the performance and cost efficiency of the resources laying behind them. The ultimate goal for datacenter operators is the development of *Fungible Datacenters*[8], as shown in Fig. 2.3b. Fungible DCs aim to completely eliminate the presence of CPUs and NICs in the loop, by introducing a novel class of microprocessors called Data Processing Units (DPUs) [169]. This architecture employs a tightly inte-

[7]Datacenter Resource Disaggregation
[8]Scale-Out Data Centers: The Best is Yet to Come

grated SW and HW co-optimization and envisions to confront the limitations of typical disaggregated systems.

Chapter 3.

Review of State-of-the-Art Challenges and Limitations

Despite its overwhelmingly positive impact, the rise of cloud computing has undoubtedly created many new challenges for the development and operations of applications. Modern datacenter infrastructure providers face a plethora of challenges, both regarding their operation per se, e.g., infrastructure cooling [170], network interconnection and management [171], power management [172], resource utilization optimization [29,30], as well as more general ones, e.g., global environmental concerns and sustainability issues[1], supply chain disruptions[2], just to name a few. While all of these challenges are of utmost importance for the efficient management of datacenter infrastructures, in this thesis we give special focus on the topic of resource orchestration, i.e., we examine methods to optimize the resource efficiency of such systems.

3.1. Efficient and fine-grained monitoring

As mentioned in Section 2.1.3, datacenters introduce several monitoring layers, both regarding the infrastructure itself (e.g., temperature monitoring) as well as the performance of the cluster and of the running applications. We give special focus on the latter part, with special emphasis given on system monitoring.

Cloud monitoring is critical for both providers and consumers [109]. On the one hand, it is an important tool for controlling and managing hardware and software infrastructures; on the other hand, it provides data and Key Performance Indicators (KPIs) for both platforms and applications. The continuous monitoring of the deployed applications

[1]Data Center Operators Vie for Leverage as Europe Eyes Efficiency Rules
[2]The End Of The Semiconductor Supply Chain In The Auto Sector As We Know It

and the underlying hardware infrastructure provides information to both providers and consumers about the workload generated by the latter and the performance and QoS provided by the former, as well as guides resource and other management activities within the datacenter. Resource management activities typically imply two fundamental steps: *i)* monitoring, that keeps track of hardware and software performance metrics; and *ii)* data analysis, that processes such metrics to infer system or application states for resource provisioning and many other activities [173]. While prior scientific research has pinpointed the fact that fine-grained monitoring can provide useful knowledge regarding the behavior of running applications [174] and, thus, guide more efficiently scheduling decisions, state-of-the-art orchestration frameworks, such as Kubernetes [175] and Mesos [119], still rely on naive metrics to place applications on the pool of available resources. To tackle these challenges, a wide range of solutions have been proposed both from academia and industry.

From an industrial point of view, we observe that top cloud providers provide their own solutions regarding monitoring of both the underlying infrastructure and the applications deployed on the cloud premises. For example, Amazon Web Services [72] provides Amazon CloudWatch and Amazon CodeGuru [176], which are specialized solutions that apply machine learning models to identify anomalous application behavior and proactively surface critical issues before they cause outages or service disruptions. Moreover, AWS X-Ray performs distributed tracing across multiple applications and systems to identify n identify and troubleshoot the root-causes of performance issues and errors. Similar to the above, Azure Cloud [177], Google Cloud Platform (GCP) [178] and IBM Cloud [179] also offer their own monitoring solutions [180–182], with the intention of providing to end-users finer observability and visibility into the performance, availability, and health of your applications and infrastructure. Except for specific cloud vendors solutions, services like Prometheus [183] allow for custom monitoring of running workloads, forming a promising area for fine-grained monitoring. In addition, several innovative monitoring solutions have raised funding in cloud industry to address the aforementioned requirements, focusing mainly on fast event logging at scale while also offering machine-learning powered analytic capabilities [184–187].

From an academic perspective, several frameworks have been developed to enable logging and fusion of micro-architectural events [188, 189] and significant research has been undertaken about monitoring approaches [190]. Diagnosing performance anomalies through monitoring data is not a new topic of interest. The proposed approaches are either *reactive*, meaning that they detect performance anomalies after they occur, for example by analyzing logs and runtime metrics [114,191–200] or *proactive*, meaning that they predict anomalies before they take place [201–205]. Furthermore, Seer and Cloudseer [60, 206] are academically presented advanced monitoring methods. Seer is an online cloud performance debugging solution that uses deep learning techniques to discover spatial and temporal patterns that translate to QoS violations from huge amounts of cloud system trace data. Cloudseer, on the other hand, uses interleaved logs to enable effective process monitoring and identify divergences during execution. While the majority of the aforementioned tools are able to provide valuable insights regarding the performance of applications and the underlying hardware, they pose specific limitations. First, the majority of prior art and monitor high-level metrics of interest (e.g., CPU/memory utilization, use of applications' logs, etc.), thus being unable to determine the impact of each specific resource on the performance degradation of applications. While prior works have shown that exploiting performance characteristics of a system through hardware performance counters has been identified as a prominent step for improving the efficiency of data centers [74, 207], there has been minimal work on how to "make profit" out of them. Second, the majority of prior efforts either do not consider interference effect or their supported predictive analytic capabilities are quite coarse, imposing a "intercept-measure-and-predict" scheme to infer performance analysis logging, failing to support continuous fine-grained monitoring, which is ideally required in modern resource allocation schemes [40, 208, 209].

Limitations of SotA monitoring frameworks

In short, the limitations of prior works regarding Cloud monitoring can be summarized as follows:

- *Inefficiency to identify root causes of performance degradation,* by monitoring mostly high-level metrics of interest.
- *Intermittent techniques to identify resource interference or complete ignorance of its impact*, that forms one of the major drawbackcs of modern Cloud systems.
- *Delivery of historical monitors*, thus, not being able to drive proactive scheduling decisions.

3.2. Resource orchestration and efficiency

Efficient management of Cloud computing resources has been in the center of attention of many research and industrial groups. The spectrum of prior work covers a wide range of optimization mechanisms applied at different layers of the orchestration hierarchy, as described in Section 2.1.4. Moreover, much research has been conducted regarding the efficiency of the disaggregated computing paradigm, described in Section 2.1.5. In the rest of this section, we group and analyze prior art with respect to:

- Cloud resource orchestration: We present a plethora of prior research that focus on improving resource efficiency inside Cloud infrastructures either by optimizing the placement of applications within the datacenter or by sizing allocated resources according to runtime applications' needs. We also highlight recent attempts that tackle the problem of resource allocation in disaggregated infrastructures.

- Application-level optimizations: Special focus is given on works that optimize applications running on top of Apache Spark [93], an analytics engine for large-scale data processing.

3.2.1. Cloud resource orchestration

Efficient management of Cloud resources forms a really challenging problem, typically due to the multiple conflicting objectives that have to be satisfied simultaneously. In this direction, a plethora of works have tackled the problem of VM placement and/or migration inside Cloud infrastructures [210–216], which however is out of the scope of this thesis. Regarding resource efficiency of multi-tenant infrastructures, much research has been conducted in the past. Predicting and controlling performance degradation of workloads executing under interference has been in the center of attention of many research groups, either by scheduling workloads arriving on a cluster [38, 41, 51, 62, 64, 73, 88, 160, 166, 209, 217–221], or by regulating resource occupation by workloads throughout their execution lifetime [38, 40, 62, 63, 63, 159, 165, 208, 209, 222–227] to improve per-application performance. A significant portion of these approaches require a priori knowledge of the target applications running on the cluster [217, 222, 224, 228, 229] or are application-specific [64, 146], thus making them unsuitable for cases of public clouds, where uncharacterized applications might arrive. Other scientific approaches have built benchmark warehouses that can be used to build performance models of applications [230]. However, such static approaches do not consider the dynamic system fluctuations and thus cannot be used for continuous resource tuning. Another considerable part detects interference and performance degradation of applications by gradually pausing collocated workloads in a coarse- [38, 40, 62, 231] or fine- [208] grained manner, followed by either injecting synthetic microbenchmarks to identify resource contention or by evaluating the performance degradation compared to the isolated execution. Those approaches effectively predict resource interference, nevertheless, pausing of applications can lead to relentless efficiency decline, considering that modern servers feature up to 100 cores and, thus, can host numerous workloads simultaneously. Some recent research works focus on controlling and tuning workloads in a feedback-like manner, where applications are gradually provided resources until no QoS violations are witnessed [209, 215]. Last, production ready resource orchestrators have also been presented in the past [51, 52, 117, 221, 232], showcasing the need for efficient supervision of resources due to interference in shared resources.

Lately, management of disaggregated, heterogeneous resources inside data-center facilities is attracting more attention [233–235]. Recent research efforts focus on mechanisms that provide dynamic disaggregated memory allocations for VMs [236–238], efficient prefetchers and replacement policies [239, 240] and cost/performance tradeoffs between heterogeneous memory pools [241]. Other scientific approaches examine the problems of application orchestration on disaggregated memory systems [242, 243] and performance modeling [244], however they either rely on emulated prototypes [244] or totally neglect the implications of resource interference in shared resources [242, 243].

Limitations of SotA Cloud resource orchestration works

In short, the limitations of prior works regarding resource orchestration of Cloud infrastructures can be summarized as follows:

- *Employment of synthetic microbenchmarks to identify sources of interference,* leading to performance degradation of running applications.
- *Application of dynamic resource allocation decisions in a feedback-based manner*, thus, not being able to proactively provide for potential performance degradation of running applications.
- *Leveraging per-application performance modeling,* which is a time-consuming task and ignores the deployment of unseen applications.
- *Neglect the presence of novel computing paradigms (e.g., hardware disaggregation).*

3.2.2. Parameter auto-tuning for Spark in-memory analytics

As mentioned in Section 2.1.4, resource efficiency through application-specific optimizations can be achieved with various techniques. In the context of this thesis, we focus on parameter autotuning for Spark applications.

Optimization and parameter autotuning of distributed frameworks (e.g., Hadoop [141], Spark [245], Storm [246]) has gained a lot of attention lately, due to the huge performance gains they can offer, when configured properly. Indeed, Spark's official documentation provides guidelines for tuning Spark applications [247], highlighting the importance of proper hyperparameter configuration. While such guidelines provide detailed insights regarding Spark parameter tuning, they mostly rely on network and memory related parameters, thus, neglecting the impact of the rest of the configuration space on performance. Moreover, they do not provide an automated fine-tuning procedure, hence, requiring from developers to comprehend Spark's mechanisms in-depth, to effectively tune their applications.

Exploiting the lack of robust tuning guidelines, many research groups have proposed automated frameworks and/or methodologies that enable efficient deployment of big-data analytics frameworks [248]. These approaches can be separated into three categories, i.e., *i)* rule-based, *ii)* experiment-driven and *iii)* model-based tuning. Rule-based tuning relies on experts' knowledge to manually tune Spark parameters [249–254]. However, experts typically consider only a few parameters, while the tuning process follows a "trial and error" approach tailored to the investigated Spark application, thus rendering this approach unable to scale on large configuration spaces and diverse workloads. The second method, search-based tuning, relies on repeated executions of Spark applications with different configuration parameters [255–260] and employs optimization algorithms to traverse through the design space. Again, this approach is bounded by the time required to explore the parameter configuration space, which increases exponentially with the number of Spark parameters considered, the application's dataset sizes examined, as well as the configuration of the optimization algorithm itself (e.g., population size and number of generations in a genetic algorithm [261]).

Last, model-based tuning methodologies detour the native execution of Spark applications, by replacing the actual execution with performance models in the loop, which predict the latency of a deployed application for a given parameter configuration. A vast amount of prior research falls into the latter category, that focus on proper parameter configuration of Spark deployments to maximize their performance. These works adopt different performance modeling techniques, namely, linear prediction models [146], simple machine learning approaches, e.g., Random Forest [147, 262, 263], Support Vector Machines [5] and Hierarchical Modeling [6] techniques, or even more sophisticated concepts, such as ensemble learning methodologies [264], application of machine learning algorithms for cluster-wise performance modeling [7] and optimization using deep neural network architectures [148]. While the aforementioned works deliver accurate performance modeling per se, they are bounded by certain design characteristics, that restrain their prevalence in more general configuration settings.

Such design characteristics include:

- *Number of Spark parameters considered:* The majority of prior scientific works consider and tune only a subset of Spark parameters, chosen either empirically [5, 6, 149, 265] or via statistical approaches [7, 266]. Furthermore, others focus only

Table 3.1.: Comparison between previous Spark auto-tuning approaches and *Sparkle* in terms of *i)* number of parameters examined *ii)* modeling approach followed *iii)* cluster size and *iv)* dataset size range considered

Ref.	**Codename**	**Model Used**	**#Params**	**Perf. Model**	**Cluster Size**	**Dataset Size**
[265]	**Wang et. al**	Decision Trees, SVM, Neural Network, Logistic Regr.	13	Per-application	3 nodes	Variable
[146]	**Ernest**	Non-Negative Least Squares	1	Per-application	20 nodes	Static
[5]	**Luo et. al**	Support Vector Machine	28	Per-application	4 nodes	Static
[147]	**ACS**	Random Forest	29	Per-app & dataset	4 nodes	Variable
[6]	**DAC**	Hierarchical Decision Trees	41	Per application	5 nodes	Variable
[149]	**ATCS**	Generative Adversarial Nets	20	Per-application	2 nodes	Variable
[148]	**ReLocag**	Graph Convolutional Networks	1	Cross-application	24 nodes	Variable
[7]	**Nikitopoulou et. al**	Random Forest	23	Cluster-wise	1 node	Variable
[4]	**Phronesis**	Random Splines, Neural Network, Random Forest	28	Per-application	4 nodes	Static
-	***Sparkle***	Hybrid CNN+LSTM+FC	101	Universal	10 nodes	Variable

on the optimization of the number of Spark executor instances [146, 148], completely ignoring the impact of the rest of the configuration space. However, as discussed earlier and also shown in Chapter 6 of this thesis, such approaches are not enduring, due to the continuous upgrade of the Spark engine itself (more parameters continuously added), the high correlation between each parameter and the deployed application, as well as the uncertainty regarding the interrelationship between different parameters [267].

- *Application-specific performance modeling:* Most proposed solutions rely on an application-specific performance modeling approach [5, 6, 146, 149, 265], while latest efforts also examine cluster-wise or cross-application techniques [7, 148]. Even though such approaches provide accurate performance predictions, they require to repeat the entire modeling procedure for every unobserved application or family of applications. In most cases, this forms a very time-consuming task, since it requires profiling of new applications for a variety of different configurations.

- *Coarse-grained data size variety:* Dataset scaling can significantly affect the compute and I/O intensity of Spark applications [268, 269] and, consequently, their performance. Nevertheless, a large amount of prior auto-tuning frameworks either model performance of Spark applications for a given dataset size [4, 5], or dataset sizes in the same order of magnitude [5, 6, 148, 265]. Yet, it is common for Spark applications to run repeatedly in the cluster for different datasets both from a qualitative as well as a quantitative point of view [270,271]. Thus, properly encoding this information in the modeling approach is crucial for maximizing prediction accuracy.

Table 3.1 summarizes the above discussion, offering an abstract qualitative comparison regarding the aforementioned state-of-art in Spark parameter auto-tuning.

Chapter 4.

Deep-Learning Driven, Interference-Aware Predictive Monitoring for Modern Multi-Tenant Systems

Modern Cloud providers are leveraging multi-tenancy as a first class system design concern. The increasing number of co-located workloads into server facilities stresses resource availability in an unpredictable manner. To efficiently manage resources in such dynamic environments, run-time observability and forecasting are required to capture workload sensitivities under differing interference effects.
In this chapter, we present Rusty, a predictive monitoring system that leverages the power of Long Short-Term Memory networks to enable fast and accurate runtime forecasting of key performance metrics of cloud-native applications under interference. We evaluate Rusty under a diverse set of interference scenarios for a plethora of representative cloud workloads, showing that Rusty i) achieves extremely high prediction accuracy, average R^2 value of 0.98, ii) enables very deep prediction horizons retaining high accuracy, e.g. R^2 of around 0.99 for a horizon of 1 sec ahead and around 0.94 for an horizon of 5 sec ahead, while iii) satisfying, at the same time, the strict latency constraints required to make Rusty practical for continuous predictive monitoring at runtime.

4.1. Introduction

Over the last few years, the number of workloads executed on the Cloud has increased rapidly and is expected to grow more in the future [272]. The rise of "cloud-native" platforms, such as Kubernetes [175], that facilitate the deployment of applications on lightweight containers and expand their capacity to dynamically scale resources, further raises the density of modern cloud systems. Moreover, current Cloud solutions, such as Amazon AWS [72], Google Cloud [178], Microsoft Azure [177] and others, provide users with elasticity and resizability of their computing capacity, leading to a dynamic provisioning of resources. This increment in the density and dynamicity of cloud workloads implies that DC operators should perform advanced resource allocation techniques, to provide both better quality-of-service (QoS) to their users, as well as maximize their profit. However, this two-factor optimization goal is rather challenging, since maximizing performance requires applications to be executed in isolation, whereas profit increment is achieved through multi-tenant job scheduling.

Recent scientific works have proposed schedulers and resource allocation techniques able to efficiently place workloads into physical/virtual servers as well as minimize their slowdown caused by multi-tenant job scheduling [38,41,63,229]. Even though such approaches improve the overall performance of co-located workloads, they usually operate at a quite coarse-grained level, i.e., either requiring prior knowledge through offline characterization of isolated executions [38, 41] or in the best case gathering and aggregating runtime performance metrics through periods of enforced execution stalling/pausing to estimate interference effects [40, 231]. In addition, they fail to measure, model or exploit the dynamic impact of each specific resource on the performance degradation caused by interference, thus imposing either low accuracy estimations and/or significant performance overheads [64, 273].

To have a better understanding of the real bottlenecks of the system and specify the root cause of performance degradation, one should take a closer look at lower level architectural events and at system-level characterization to get insights regarding the state of the system [274]. Towards achieving the above goals, monitoring and analysis of system signals from within the data-center has been proven to be really beneficial and insightful. For example, Alibaba and Google provides real open-source traces [29] from their cluster systems and encourage researchers and practitioners to analyze them. In addition, Google has identified that monitoring low-level performance counters can drive better scheduling resource management and scheduling decisions [74]. In addition, the highly dynamic characteristics of cloud applications require very small monitoring and reaction times, e.g. with 1 second data sampling granularity becoming the new gold standard in cloud deployments [185].

Some hardware-enabled approaches [275–277] enabling continuous and fine-grained interference effects estimation have been proposed in the past, however their implementation requires specialization of processor's hardware, limiting their applicability in current and near future servers. Interestingly, this shift towards time granularity shrinking, is also reflected in recent micro-architectural and system hardware advancements that allow fine-grained resource tuning. For example, Intel®'s Cache Allocation Technology in the latest Xeon® processors provides software-programmable control over the amount of cache space that can be consumed by a given thread, or container [278]. Moreover, containers provide control over the exact amount of CPU and RAM resources committed to applications, while power capping frameworks [279] enable direct and fine-grained power allocation policies to be applied.

To take advantage of these low-level features, runtime monitoring that feeds and guides the scheduling algorithms should be able to dynamically predict per application resource needs in order to enforce optimised online decisions. Prior work has shown that applications experience different phases throughout their lifetime [161, 162, 208], which lead to erratic behaviors regarding memory access patterns and CPU utilization. Such behaviors become even more inconsistent when considering the interference caused by co-located applications. Therefore, fine-grained run-time predictability is crucial to elevate online resource management decisions.

From the above discussion, it is evident that new sophisticated monitoring solutions are needed to efficiently handle the emerging field of cloud-native applications. Exploiting the underlying hardware infrastructure capabilities, application and infrastructure monitoring should shift towards providing i) faster observability, to perceive the extreme diversity and dynamism in the variable workloads, and ii) continuous runtime and interference-aware predictability, i.e. to drive resource allocation decisions in a more educated manner. During the last years, several innovative monitoring solutions have raised funding in cloud industry to address the aforementioned requirements, focusing mainly on fast event logging at scale while also offering machine-learning powered analytic capabilities [184–187]. Seer and Cloudseer [60, 206] form similar advanced monitoring solutions proposed from academia. However, the supported predictive analytic capabilities of those frameworks are quite coarse, imposing an "intercept-measure-and-predict" scheme to infer interference effects, thus failing to efficiently support continuous fine-grained monitoring, ideally required in modern resource allocation schemes [40, 208, 209]. Even though extended research has focused on run-time system predictability, those approaches mostly rely on the application of relatively simple empirical or regression models [280–282], being usually reactive in action, not capturing interference and neglecting the long-term dependencies and recurrence found in system level monitoring signals, thus reducing their predictive capabilities.

In this chapter, we present Rusty, a monitoring framework that leverages Long Short-Term Memory (LSTM) networks to enable fast and accurate resource and energy consumption predictions of a system under interference. Specifically, in this work:

- We provide the first non-intermittent predictive monitoring system that is able to forecast low-level performance counters of a system under interference. Specifically, through Rusty, we are able to predict the IPC and Last-Level Cache (LLC) misses of applications running concurrently in a multi-core and, also, the energy consumption of it. Opposed to prior-art approaches that are based on short online micro-benchmarking to model interference [41, 62, 208], Rusty utilizes LSTM predictive capability to enable continuous on-the-fly predictions of interference-aware performance metrics.

- We deliver an interference-aware analysis on the low-level metrics distributions from a large pool of diverse cloud workloads, i.e. the scikit-learn [283], the Cloudsuite [83] and the SPEC2006 [284] benchmark suites. The aforementioned analysis decomposes interference effects in a per-resource manner offering an in-depth view on the sensitivity of system metrics to different stressing scenarios.

- In contrast to existing approaches, e.g. [184, 185, 206], that focus on a straightforward appliance of regression models, in this work, we provide in-depth analysis and specific insights on LSTM parameters that affect the accuracy and complexity of the model. Through systematic exploration, analysis and fine-tuning of the LSTM's architecture and hyper-parameters, Rusty enables extremely lightweight predictive models to be deployed and operate online for continuous monitoring and adaptation.

- We evaluate Rusty in terms of its prediction precision and also demonstrate the superiority of Rusty's LSTM network over simpler machine learning approaches. Our experimental results show that Rusty exhibits very high prediction accuracy, i.e. average R^2 value of 0.98 and enables very deep prediction horizons retaining high precision, e.g. R^2 of 0.99 for a horizon of 1 sec ahead and around 0.94 for an horizon of 5 sec ahead. Finally, we show that Rusty is both i) really lightweight, introducing minimal time overheads and providing predictions in terms of milliseconds ii) and also, that it can be seamlessly transferred between systems of diverse specifications, with a slight decline in accuracy, thus forming a promising solution for runtime predictive resource allocation.

The rest of this chapter is organized as follows: Section 4.2 describes our experimental setup and also demonstrates the aforementioned per-resource interference analysis. Section 4.3 presents our proposed predictive monitoring framework. Finally, sections 4.4 and

Table 4.1.: Target System specifications

Processor Model	Intel® Xeon® E5-2658A v3
Cores per socket	12 (24 logical) @2.20GHz
Sockets	2
L1 Cache	32KB instr. & 32KB data
L2 Cache	256KB
L3 Cache	30MB, 20-way set-associative
Memory	256GB @2133MHz
Operating System	Ubuntu 16.04, kernel v4.4

4.5 show our results and conclude the chapter, respectively.

4.2. Experimental Setup & Specifications

4.2.1. Target system characterization

Rusty targets multi-core server systems usually found in DC environments, which are able to provide information regarding performance metrics of the system. All of our experiments have been performed on a high-end server, outlined in Table 4.1. In order to monitor low-level performance counters, we utilize the Performance Counter Monitoring (PCM) API [189], a tool initially developed by Intel® and currently maintained in a separate github repository [285], which provides a plethora of hardware performance counters for each logical core, each socket, as well as the whole server system. We focus on specific performance counters, which we discuss in detail in section 4.2.2. To simulate a cloud environment, all the referenced benchmarks running in the system have been containerized, utilizing the Docker platform [286]. Moreover, to be able to extract per-workload metrics, all the applications are assigned on a randomly selected core of the system, using the affinity rules of the docker engine.

4.2.2. Target metrics characterization

Emphasis is given to low-level system metrics, provided directly from the PCM tool [285]. For the rest of this chapter, we focus on the following three performance counters, however our framework can be utilized for any metric provided by the PCM tool:

- **Instructions Per Cycle (IPC):** IPC gives insight information of the performance of the executed workload and its predictability can assist to dynamically boost/relax resources to meet certain constraints. In prior works, IPC has been used as a metric of interest to depict performance related behaviors in data center environments [208,231,273,287]. For example, in [208], IPC is used to determine phase changes of applications executed under interference, whereas in [273] it used as a performance indicator of co-located workloads.

- **Last-Level Cache misses (L3M):** Memory access is considered a major bottleneck in performance. Even in modern NUMA architectures inefficient memory contention management can lead to a severe performance degradation [207]. Especially in data-center environments, it has been shown that the huge instruction sets of cloud workloads are between one and two orders of magnitude larger than the L1 instruction cache can store, and can lead to repeating instruction cache misses, which damage performance [288,289], whereas latest reports from large-scale clouds show that memory is becoming the new bottleneck, destroying performance of applications [29]. The Last-Level Cache is basically the "bridge" between cores requesting data and memory storing them. Last-Level Cache misses provide first-level details regarding the interference, since higher values in LLC misses can depict the tendency of multiple applications running on the system competing for access on the main memory [290].

- **Socket's energy consumption (NRG):** Energy consumption is considered as a first class constraint in modern data-center deployments [291, 292]. Accurate energy predictions at runtime significantly impact energy proportionality of the DC's nodes [293], as well as allow for the deployment of more sophisticated power capping strategies [294, 295]. In the same direction, controlling the energy and temperature levels of a system improves resiliency and components lifetimes, as well as it also reduces the cost required for cooling, which is often amounted to 50% of a DC's overall costs.

4.2.3. Workload characterization

Modern data-center server machines accommodate a large and wide range of workloads, which are basically either batch/best-effort (BE) applications, or user-interactive/latency-critical (LC) applications. The former type of workloads require the highest possible throughput, whereas the latter demand to meet their QoS constraints. In order to cover both BE and LC workloads, we consider workloads from three popular scientific benchmarking libraries, i.e. sci-kit learn [283] and SPEC2006 [284] (as BE) and cloudsuite [83] (as LC) suites. The complete list of examined applications along with a brief description is shown below:

1. **AdaBoost Classifier** [283]: This is a meta-estimator that fits a classifier on a dataset. Then, it fits additional classifiers on the same dataset but the weights of incorrectly classified instances are adjusted, to deal with more difficult cases.

2. **Lasso Regressor** [283]: Lasso is a linear model that estimates sparse coefficients. It is vital in the field of compressed sensing, since it prefers less non-zero coefficients, reducing the features that the solution is dependent to.

3. **Linear Discriminant Analysis** [283]: LDA can be used to perform supervised dimensionality reduction, by projecting the input data to a linear subspace consisting of the directions which maximize the separation between classes.

4. **Linear Regressor** [283]: It fits a linear model with coefficients to minimize the residual sum of squares between the observed and prediction targets in a dataset.

5. **Random Forest Classifier** [283]: This is a meta-estimator that fits a number of decision tree classifiers on sub-samples of a dataset and uses averaging to improve the accuracy.

6. **Random Forest Regressor** [283]: This is the same as 5, but performs regression instead of classification.

7. **Stochastic Gradient Descent Classifier** [283]: SGD is a very efficient approach of discriminative learning of linear classifiers under convex loss functions and has received recently a lot of attention in the field of deep learning.

8. **Stochastic Gradient Descent Regressor** [283]: This is the same as 7, but performs regression instead of classification.

9. **astar** [284]: A* is a popular 2D path-finding algorithm used by the artificial intelligence engines of online games.

10. **bzip2** [284]: This benchmark performs compression and decompression of data entirely in memory.

11. **cactusADM** [284]: It solves the Einstein evolution equations,a set of ten nonlinear partial differential equations.

12. **h264ref** [284]: Implementation of H.264/AVC, the latest compression standard, used for video broadcasting.

13. **leslie3D** [284]:LESlie3d is the primary solver used to investigate a wide array of turbulence phenomena such as mixing, combustion, acoustics and general fluid mechanics.

14. **sphinx3** [284]: This is a speech recognition toolkit with various tools used to build speech applications. It contains a number of packages for different tasks and applications.

15. **In-Memory analytics** [83]: This workload uses Apache Spark [93] and performs a collaborative filtering algorithm, in order to provide to users recommendations for movies.

16. **Data Caching** [83]: This workload uses the Memcached caching system and simulates the behavior of a Twitter caching server using a dataset available from Twitter.

17. **Data Serving** [83]: This benchmark uses an Apache Cassandra server and is based on the Yahoo! Cloud Serving Benchmark (YCSB), which is a framework used to benchmark data store systems.

18. **Media Streaming** [83]: This benchmark comprises of two seperate containers, a server and a client. The server container utilizes an Nginx web server in order to stream hosted videos of various lengths and qualities to the clients. The client generates a mix of requests for different videos, using httperf's wsesslog session generator,in order to stress the server.

19. **Web Search** [83]: This benchmark is based on the Apache Solr search engine framework.

20. **Web Serving** [83]: This benchmark is based on the Apache Solr search engine

framework.

Through scikit-learn we examine workload skeletons that are representative of modern machine learning applications. Each instance performs the training phase of the specific workload, with datasets comprised of 40.000 instances with 784 features per instance. The spec2006 benchmarks represent computational heavy workloads as well as every-day operations performed in the cloud. We use the default configurations and datasets provided by the spec2006 suite. Finally, the cloudsuite benchmarks represent services hosted in modern data-center cloud environments. The Data Serving relies on the Yahoo! Cloud Serving Benchmark [296] and the Cassandra data store [297]. In-Memory Analytics utilize Apache Spark [93] and runs a collaborative filtering algorithm on a dataset of movie ratings. Media Streaming utilizes NGINX [298] as a server for streaming videos of various lengths and qualities. Moreover, Web-Search depends on Apache Solr [299] search platform and simulates real-world clients that send requests to index nodes. Finally, Web-Serving utilizes three servers, an NGINX [298] web server, a Memcached [105] caching server and a MySQL [300] database server, simulating modern services hosted on the cloud. For the cloudsuite benchmarks, we use the default configurations and datasets as provided by the respective github repository [301] and for client-server benchmarks, we focus and monitor the server-side workload. Table 4.3 summarizes the benchmarks used throughout the chapter. For each of the examimed applications, Table 4.3 reports the mean value of 6 system wide low-level metrics, i.e. characterizing the IPC, on-chip cache memory behavior (**L2** cache **M**isses and **L3** cache **M**isses), energy consumption (**NRG**) and memory I/O bytes **R**ea**D** from and bytes **WR**itten to memory), when executed in isolation, sampled every 100 milliseconds. For the rest of this chapter, each workload will be identified by its ID (e.g. SK1 for AdaBoost Classifier) for simplicity.

Impact of interference on low-level metrics: To showcase the sensitivity of the considered workloads w.r.t. differing resource interference, we utilize the iBench suite [3], which provides contentious micro-benchmarks, each of which stresses a different shared resource in a multi-core chip (processing cores, cache capacity and memory capacity and bandwidth). Specifically, to demonstrate the impact of per-resource inteference on the low-level system metrics, we generate and deploy different system-wide interference scenarios, by spawning random jobs from the iBench suite (up to the number of available threads) with various intensity levels, each one assigned on a randomly selected core of the system. We have extended the original iBench to accommodate extra levels of user tunable intensity to cover larger a wider range of interference scenarios. We note that single resource interference is considered only in this section, while the rest of the chapter considers multi-resource interference either through iBench, or through real workload collocations. The hardware counters collected, provide information regarding the state of

Table 4.3.: Benchmarks used as representative cloud applications. Scikit-learn and spec2006 workloads can be considered as Best-Effort (BE) applications, whereas cloudsuite workloads as Latency-Critical (LC) ones. Columns 4-9 report the average Instructions Per Cycle (IPC), L2 cache misses (L2M), L3 cache misses (L3M), energy consumption (NRG) and bytes read from (RD) and written to (WR) memory when executed in isolation.

	ID	**Benchmark**	**IPC**	**L2M(M)**	**L3M(M)**	**NRG(J)**	**RD(GB)**	**WR(GB)**
scikit-learn [283]	SK1	AdaBoost Classifier	1.10	2.42	1.21	4.21	0.10	0.01
	SK2	Lasso	1.77	0.88	0.35	4.34	0.17	0.01
	SK3	Linear Discriminant Analysis	1.94	1.17	0.18	4.28	0.05	0.03
	SK4	Linear Regression	1.83	0.91	0.16	4.23	0.04	0.02
	SK5	Random Forest Classifier	1.78	0.64	0.16	4.39	0.03	0.01
	SK6	Random Forest Regressor	1.29	2.69	0.22	4.29	0.02	0.01
	SK7	Stochastic Gradient Descent Classifier	1.85	0.46	0.21	4.32	0.06	0.01
	SK8	Stochastic Gradient Descent Regressor	1.86	0.39	0.13	4.19	0.03	0.01
spec2006 [284]	SP1	astar	0.86	0.86	0.03	4.20	0.006	0.003
	SP2	bzip2	1.34	1.24	0.02	4.17	0.006	0.004
	SP3	cactusADM	1.40	1.19	0.43	4.29	0.071	0.027
	SP4	h264ref	1.91	0.27	0.01	4.25	0.005	0.002
	SP5	leslie3d	1.50	0.85	0.47	4.32	0.322	0.153
	SP6	sphinx3	2.05	1.45	0.01	4.20	0.004	0.002
cloudsuite [83]	CS1	Data Serving	0.62	1.37	0.14	3.93	0.009	0.004
	CS2	In-Memory	1.45	0.96	0.18	4.24	0.033	0.026
	CS3	Media Streaming	0.55	0.04	0.02	2.26	0.022	0.002
	CS4	Web-Search	0.50	0.02	0.01	2.66	0.006	0.003
	CS5	Web Serving - Nginx	0.54	0.29	0.15	2.74	0.005	0.002
	CS6	Web Serving - Memcached	0.36	0.03	0.01	2.74	0.005	0.002
	CS7	Web Serving - MySQL	0.58	0.07	0.01	2.74	0.005	0.002

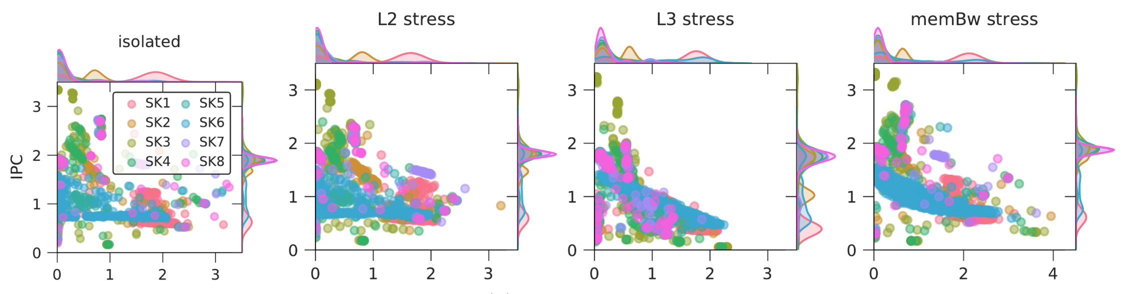
the underlying system at a per-resource manner, thus allowing us to detect true causes of interference and bottlenecks, i.e. which resource is responsible for the performance degradation our system (and consequently our workloads) is experiencing. Even though individual performance metrics can, in some cases, depict the resource of interference, this is not always the ground-truth. For example, a high number of LLC misses could imply interference in the cache, however, the same could apply for a case of low LLC misses, as we discuss below.

Figure 4.1 illustrates the tendency of IPC and LLC misses with differing resource stress interference, showing the high diversity in terms of IPC and L3 cache misses between our target workloads. This analysis reveals three major upshots. Firstly, all examined workloads are insensitive to interference at the level of L2 cache, since the distributions of both IPC and LLC are slightly shifted in all cases. Secondly, interference at the LLC of the system induces the highest performance degradation, realizing LLC as one of the major bottlenecks in modern server systems. This is even more obvious for the cloudsuite benchmarks, where, there is a clear spread and shift of the IPC distributions towards 0, even though similar services have been revealed as memory bandwidth sensitive in prior works [40]. Finally, what is of great interest in the case of L3 stress scenario, is that, in spec2006 benchmarks the LLC misses increase, whereas in scikit-learn the misses decrease. This irregularity appears due to the combination of two factors; the performance degradation that the workloads experience due to interference and the sampling rate of the PCM tool. Performance degradation causes the signal traces to stretch, thus spreading the LLC misses in time. This reveals the importance of predictive monitoring in fine-grained timescales versus typical reactive performance monitors, since in cases of high application slowdown, even though the total number of LLC misses increase, a lower sampling rate would report lower LLC misses values, since the misses would be spread more widely in time.

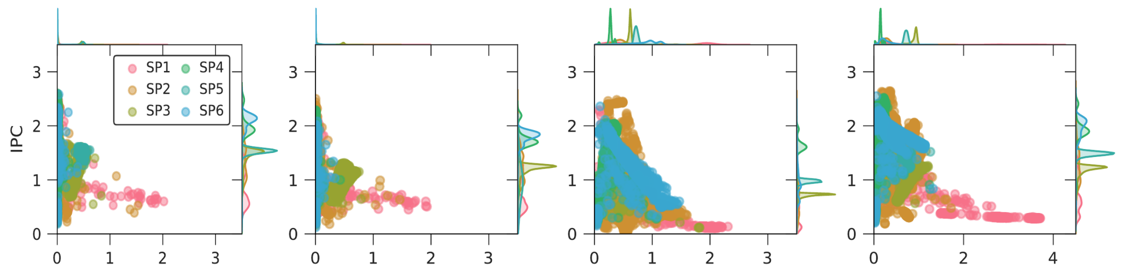
4.2.4. Motivational observations: Why predictive monitoring?

Rusty sets the framework for extending current state-of-the-art systems, by promising extremely accurate runtime predictive monitoring of "low-level" system metrics for deep horizons. The term predictive monitoring may raise several questions, like, *is predictive monitoring important for improving resource efficiency and performance in data-center environments?* or *are low level metrics offering true/important system insights?* etc. In this section, we offer a small debate addressing fundamental issues and motivational observations behind Rusty.

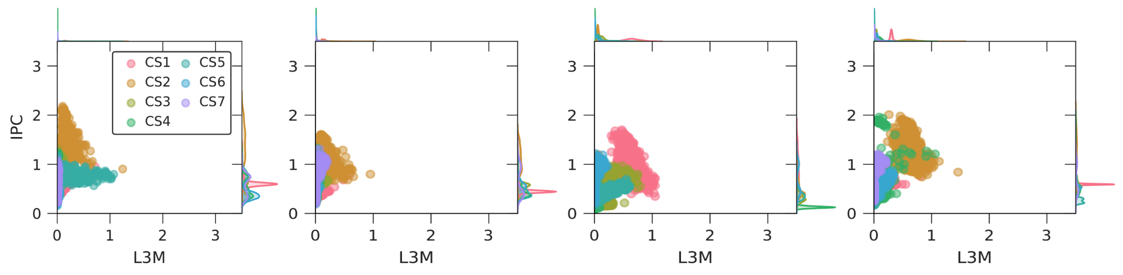
We showcase the importance of predictive monitoring with a typical example obtained



(a) scikit-learn



(b) spec2006



(c) cloudsuite

Figure 4.1.. Absolute values of IPC and L3M distribution for different resource stressing scenarios. In all figures, x-axes and y-axes denote pairs of Last-Level Cache misses (L3M) and IPC as sampled from the PCM tool respectively, whereas the line charts show the relocation of the distributions of L3M and IPC.

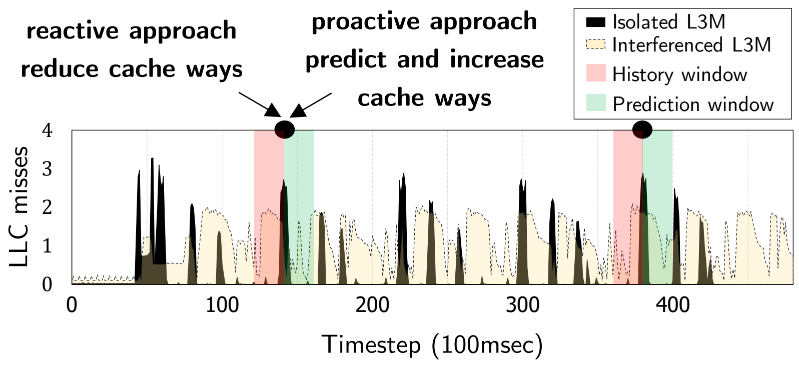


Figure 4.2.. Real trace of Random Forest Regressor benchmark showcasing the advantage of proactive predictive approaches compared to current state-of-the-art reactive approaches.

throughout our experiments. Fig. 4.2 shows part of a real trace of LLC misses from the Random Forest Regressor benchmark. In this example, we assume that a runtime controller would take resource management decisions every 200msec (denoted as blue dotted lines). In addition, we suppose that the resource of interest is the number of cache ways given to the core executing the application. We give emphasis on the point highlighted with the black bullet. As we mentioned before, current state-of-the-art approaches [40, 209, 215] rely on reactive decision making approaches where resources are shared based on the *"history"* state of the system (red window in the figure). Since the LLC misses in this phase of the application fluctuate at low values, a reactive approach would result in lowering the cache-ways given to the application. However, we observe that in the upcoming window (green highlighted), the LLC misses of the application increase and, therefore, the number of cache-ways should be incremented as well. A proactive, predictive approach could determine this behavior before it even occurs, thus prohibiting of such inefficient configurations. Interestingly, we may also notice that the same "behavior" appears repeatedly throughout the lifetime of the application, showing that pure reactive approaches would make the same mistake again and again, probably leading to a catastrophic, for the application, management of resources.

A deeper look at Fig. 4.2 reveals the importance of the other major aspect of Rusty, i.e. the interference awareness in predictive monitoring. As depicted, we have superimposed in Fig. 4.2 the signal of LLC misses when executing the Random Forest Regressor benchmark under interference (the yellow signal). As shown, interference effects heavily modify the signal structure/characteristics of the original isolated LLC misses signal. In this case, as shown, the magnitude of the LLC misses peaks have become smaller, but their duration have been increased. Focusing in the second black dot, around t=400, we may notice that while the reactive cache ways controller will fail for the same reasons as before, the same stands also for the case of the predictive controller that operates in an interference unaware manner, i.e. guided by a predictor trained only on isolated execution traces. More in detail, in this case, the interference unaware predictor will mis-predict resource demands, considering a steeper increase in L3 misses according to its isolated training set. However, as shown, the true L3 misses under interference presents a stable slope for the next period, thus imposing cache ways over-provisioning leading to either QoS degradation of co-running applications or in the worst case in resource starvation.

4.3. Rusty: Predictive Runtime Monitoring

Rusty relies on Long Short Term Memory networks [302], a special kind of Recurrent Neural Networks, that are able to make predictions based on long-term sequential dependencies. The employment of LSTM networks was inspired by recent research activities, where researchers are beginning to explore the employment of LSTM networks for different micro-architectural tasks. In [303], authors propose a dynamic voltage frequency scaling algorithm which uses LSTM networks to predict the workload of cores in an upcoming time period, based on performance counters collected during the previous one. Neither interference not recurrence have been taken into account and also the evaluation is based on simulated workloads and not on real system settings. In [304], the authors utilize LSTM networks in order to build efficient memory pre-fetchers and show that the employment of LSTMs for this task outperforms the state-of-the-art of traditional hardware prefetchers. Finally, in [206], the authors use a heavy-weighted LSTM and CNN network to detect QoS violations of microservices running in the cloud. Even though Seer also utilizes LSTMs, the authors apply them for debugging purposes of cloud workloads, mainly dealing with a classification problem for identifying QoS violations, thus not exploring LSTM's runtime forecasting capabilities, which are useful for predictive resource allocation. In this chapter, we propose and explore the efficacy and robustness of LSTMs for fine-grained system-level predictability under resource interference, clearly extending [303] and being orthogonal to [304] and [206].

In short, an LSTM cell, instead of a single neural network layer, contains four layers, three sigmoids (σ) and one hyperbolic tangent ($tanh$) layer. In brief, the first layer, also called "forget gate layer", applies a sigmoid function to the inputs and decides what information the network will throw away from the cell state. The second sigmoid layer is called the "input gate layer" and decides, along with the $tanh$ layer what new information is going to be stored and updated inside the cell. Finally, the last sigmoid layer acts as a refinement filter that decides which values are going to leave the cell.

4.3.1. Training Rusty

Rusty is dynamic, in terms that it requires no a priori knowledge of workloads running in the cluster. The only thing that Rusty requires offline is a configuration file specifying the target metrics to be predicted and two variables, the *history* and the *horizon*, which we explain below.

Fig. 4.3 shows an overview of the general procedure followed by Rusty to train the LSTM model. As shown, Rusty initially receives four inputs: i) the workload we would like to predict low-level metrics for (sec. 4.2.3), ii) the performance metric to be predicted, which can be any metric from the PCM tool (for the current work we focus on IPC, L3M and NRG - sec. 4.2.2), iii) a variable called *history* and iv) a variable called *horizon*. *History* refers to the number of samples derived from the PCM tool and used as input sequence to the LSTM model, whereas *horizon* refers refers to the number of output features/future values that the LSTM model will predict. It should be noted here that, both *history* and *horizon* are relative to the sampling rate of PCM, i.e., for a sampling rate of 1 second, a *history/horizon* value of 15 would imply samples corresponding to 15 seconds of information, whereas for a sampling rate of 0.1 seconds, the same value would imply samples corresponding to 1.5 seconds. For the rest of this chapter, we set the monitoring interval of PCM equal to 100msec. As shown in prior work [206], the monitoring interval can significantly affect the effectiveness of the model, with intervals greater than 100msec having negative impact on the accuracy.

Interference-Aware Trace Collection: The first phase of Rusty is responsible for collecting applications traces of our target workload under interference. To achieve this, the target application is executed on the system 100 times, each time with differing interference load, which can be either real or synthetic (see sec. 4.3.3). For the rest of this analysis section, we imitate interference by utilizing the iBench suite [3], which provides contentious micro-benchmarks, where each one targets a different shared resource in a multi-core chip (processing cores, cache capacity, main memory bandwidth) and has tunable intensity. Specifically, we spawn random jobs from the iBench suite (up to the number of available threads) with various intensity levels, where each one is assigned on a randomly selected core of the system. During the execution, the framework collects and stores all information regarding performance metrics of the system, through the PCM tool. Following the above procedure, iBench produces static interference for the whole duration of the workload's lifetime. However, by considering multiple iBench co-execution scenarios per application, we note that the interference effects per scenario

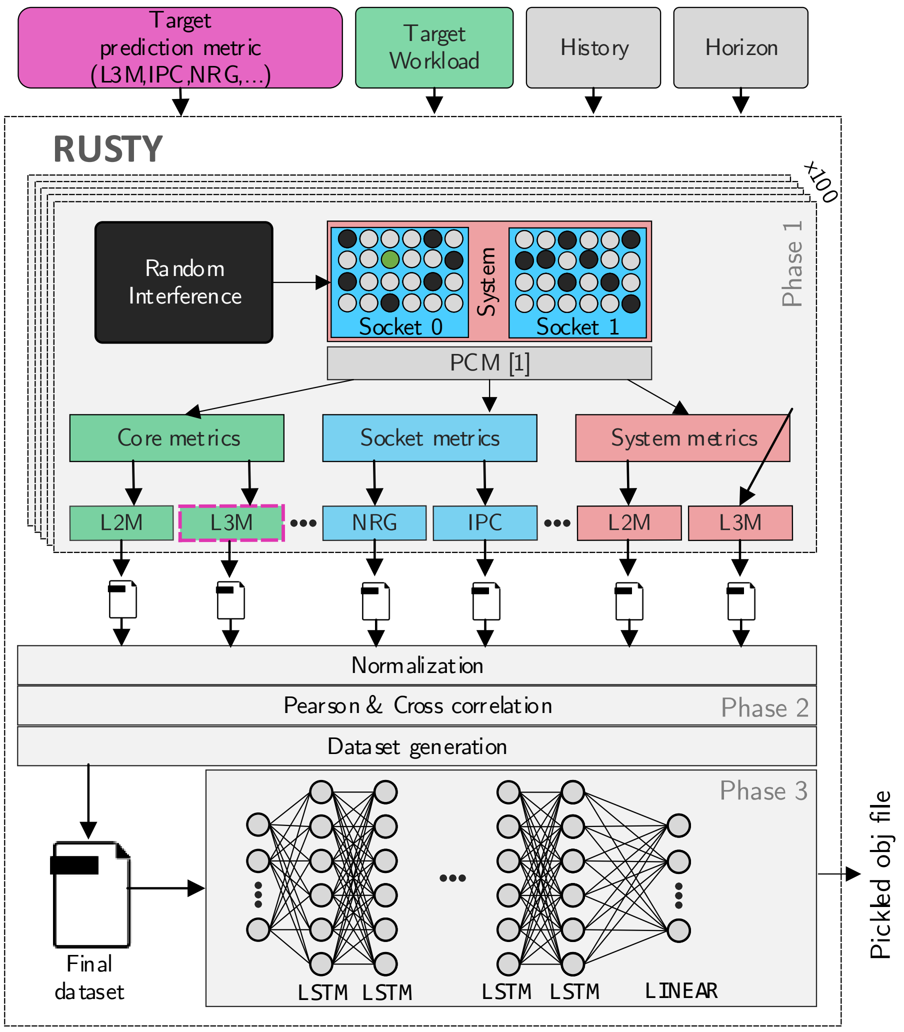


Figure 4.3.. Rusty's architecture overview

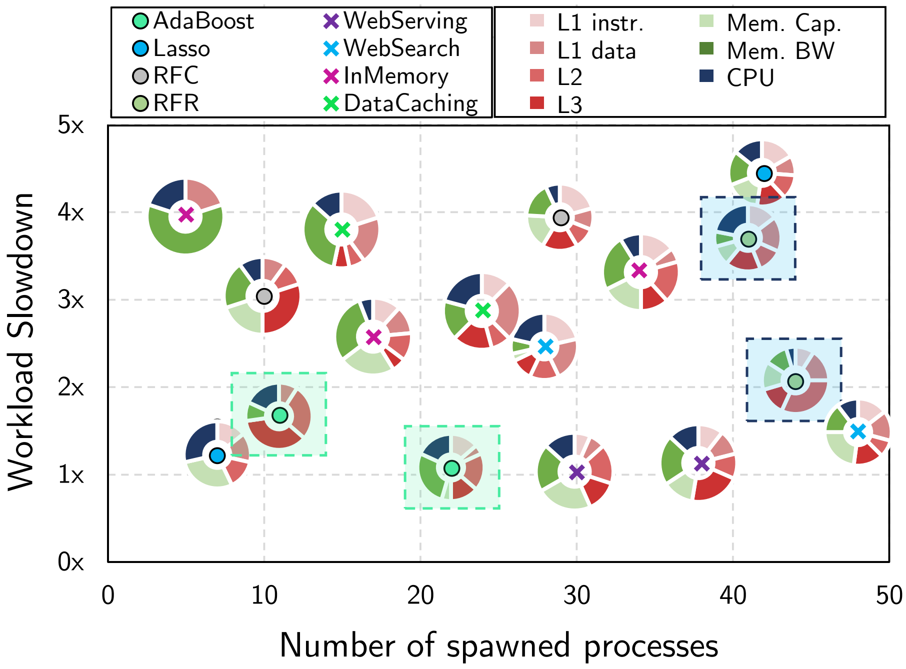


Figure 4.4.. Impact of interference on the performance of target workloads. Each doughnut chart corresponds to a specific application (left-handside legend), experiencing differing per resource interference stress(right-hand side legend). Highlighted dougnuts shows the potential of slowdown variability, i.e. how the same application is affected from different mixture of interfered workloads.

can change dramatically, thus producing disparate impact on the performance of the examined workload.

Figure 4.4 provides key insights of the impact of co-located jobs on the performance of our target workloads. The x-axis depicts the number of spawned interference processes from the iBench suite, whereas the y-axis shows the slowdown our target application experiences due to interference. We measure slowdown as the ratio of the execution time of the application running under interference to the execution time when executed in a completely isolated environment. Each point in the graph, depicts a different benchmark (left legend), whereas the surrounding doughnut chart indicates the distribution of the spawned interference jobs (right legend), for the specific experiment. As shown in the highlighted rings, scaling the number of applications does not inherently imply that the application experiences increased slowdown. The level at which the interference is applied highly affects the performance of the executed workload. For example, regarding the AdaBoost classifier (left highlighted rings) it is evident that interference on the L3 cache provokes higher performance degradation than interference on the bandwidth of the memory.

Data pre-process and dataset generation: The outcome of phase 1 is 100 low-level metric traces, which correspond to the different execution scenarios under varying interference. Phase 2 forms a pre-process phase, where the raw data of Phase 1 are prepared and processed in order to create the final dataset for our LSTM model. First, due to the different scales on the assembled data, we perform a min-max normalization, in order to bring the data values between the range [0, 1]. The min-max normalization is defined as follows:

$$x^{\mu}_{new} = \frac{x^{\mu} - x^{\mu}_{min}}{x^{\mu}_{max} - x^{\mu}_{min}} \tag{4.1}$$

where x^{μ} is the unnormalized value of a point of the metric μ and x_{max}, x_{min} are the max and min values from the respective set.

Then, we calculate some metrics of interest (pearson and cross correlation), which are used to minimize the amount of data fed to our model. We discuss this process in detail, in sec. 4.3.2. The last step of Phase 2, is the dataset generation. Figure 4.5 depicts the procedure followed to create our training and test datasets.

As mentioned earlier, through the execution of random interference-aware scenarios, we collect 100 pairs of signals, each of which corresponds to a scenario with different interference.

Starting with the signals of the first scenario ($j = 0$), we create a sliding window W, where $|W|$ is equal to the history plus the horizon length, where "history" refers to the number of past samples used as input to the model and "horizon" refers to the number of future samples that the model predicts. This window is used to divide the desired metric's signal, as well as its correlated signal (e.g. L2M and L3M) to instances of the same length. These parts correspond to different execution phases of the application, which, at inference time, will match to the respective runtime system metrics collected by Rusty. The sliding value of the window corresponds to the decision making interval of a runtime resource manager. Without loss of generality, we suppose that a runtime manager would take resource management decisions every 2sec (this is a configurable value). Therefore, every time we slide the window 20 points to the right (since we set the PCM sampling rate equal to 0.1sec). Hence, each signal is divided into k parts, where $k \simeq \frac{total\_signal\_length}{sliding\_interval}$. Finally, all these windows form our final dataset, which is then split into training and test set accordingly.

LSTM D.S.E. and training: After the data have been pre-processed, and the re-

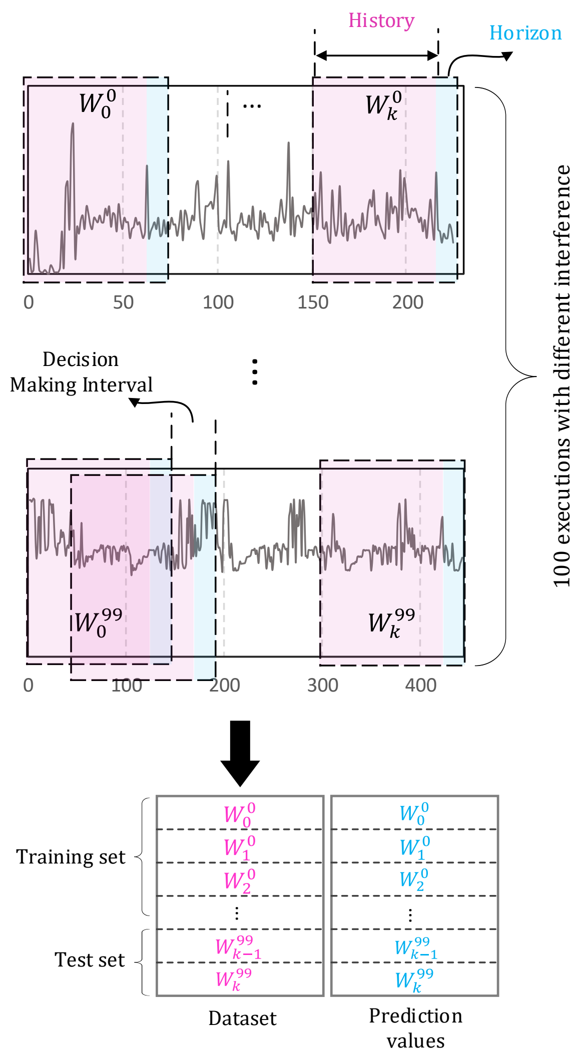
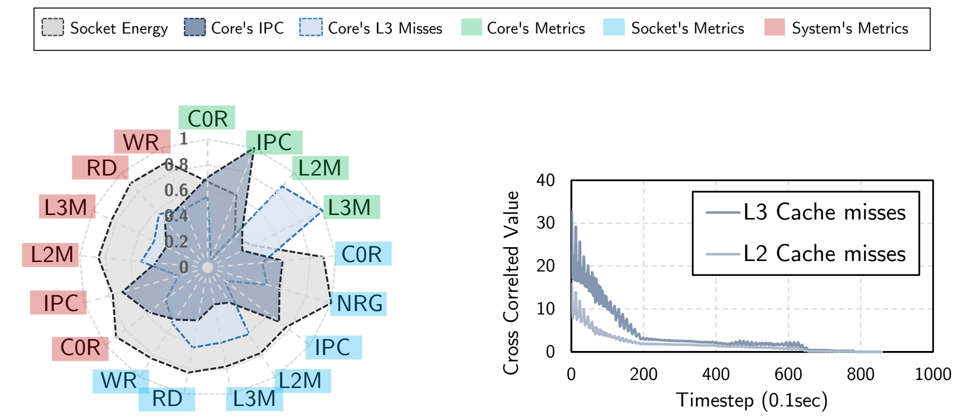


Figure 4.5.. Procedure followed to generate training and testing datasets. Each trace collected in Phase 1 is split into equal windows W with length $history + horizon$, each of which forms an instance of our final dataset.



(a) Pearson correlation of *core*'s IPC,LLC misses and socket's energy consumption with other core, socket and system metrics for Linear Regression.

(b) Cross correlation of L3 Cache misses sequence with L2 Cache misses sequence and itself for the Linear Regression workload.

Figure 4.6.. Correlation exploration for design space pruning.

spective dataset has been created, Phase 3 performs a Design Space Exploration (DSE) over the configurable parameters of the LSTM model, which we describe in detail, in section 4.3.2. Once the overall best parameters are defined, the final LSTM model is trained and the learnable parameters (i.e. weights and biases) are saved to a pickled object file.

4.3.2. LSTM architecture and hyper-parameter tuning

The real-time requirements of runtime monitoring demands that the LSTM model should be as compact as possible, i.e. use as little features as possible and also, carry as little information as possible from the PCM tool for faster processing. In this section, we explore parameters that affect the accuracy and the depth of our model. Our goal is to provide an architecture that can achieve high accuracy, w.r.t. real-time constraints.

Which metrics to choose as input features? For each one of the 100 executions performed in the offline part, certain metrics regarding the cores, sockets and the whole system are obtained. Then, Rusty designates the metrics to train the LSTM model with. To do so, it evaluates the correlation of the target prediction variable with the other performance metrics using the Pearson correlation coefficient.

Figure 4.6a shows the correlation of IPC, L3M and NRG with the rest PCM's performance counters for the Linear Regression workload. The correlation pattern of the LR workload presents great similarities with the patterns exhibited among the others workloads, which are not presented due to space limitations. As shown, there is a high diversity between the correlations of the prediction variables (self-correlation equals to 1). NRG shows high correlation values with all the metrics related to the socket and the whole system, whereas it demonstrates lower correlation values with the core's metrics. As expected, L3 misses are highly correlated with L2 misses, as well as read (RD) and write (WR) operations from/to the memory. The correlation between L2 and L3 misses does not negate the fact that stressing the L2 cache does not impose degradation on the performance of workloads. We expect L2 and L3 misses to follow a highly correlated pattern, since L2 misses consequently lead to higher traffic towards L3. In cases of interference, L2 misses are more likely going to lead to L3 misses, since more workloads contest for last level cache occupancy and, therefore, they continuously erase the contents of the cache. On the other hand, stressing the L2 cache does not inherently implies a large drop in performance, since we still have the chance to find our data in the LLC, thus, remaining on-chip.

How far back to seek for valuable information? LSTMs capture dependencies on sequential data. Thus, an important design decision is determining the length of the sequence provided to our model (*History* value), i.e., how far back to search for valuable information. To extract this information, Rusty calculates the cross-correlation between the metrics provided as input features to the model. Fig. 4.6b shows the auto-recurrence degree found the L3 and L2 misses time series, under interference, for the Linear Regression benchmark. As shown, there is a reducing (as expected) effect, quite high though up to the first 100 points. In all benchmarks, we observed a reducing/decaying effect, quite high though, up to the first 50-70 points for small benchmarks and up to 200 points for larger benchmarks. In order to ensure effective auto-correlation information, Rusty focuses on sequence lengths between 50-70 points, covering the region exhibiting the highest recurrence dependencies. However, we note that Rusty supports configurable "history" windows, easily adapted to differing "history" sizes.

Fig. 4.7 provides insights regarding the R^2 achieved for different *History-Horizon* combinations. Specifically, each cell of the heatmap illustrates the geometrical mean of R^2 scores for the respective, metric, *history* and *horizon*, over all our target workloads. As shown, the LSTM network can effectively predict L3M and IPC for a *horizon* of $25-35$ points, given as input *history* sequences of 12 samples or more. Regarding the socket's energy consumption, we observe that our model can predict equivalent *horizon* windows, even with a single *history* value, showing the extensive expressiveness of LSTMs on low rank fluctuating signals. In the examined case of forecasting during the execution of a specific mixture of co-running workloads, energy signals can be safely consider to expose

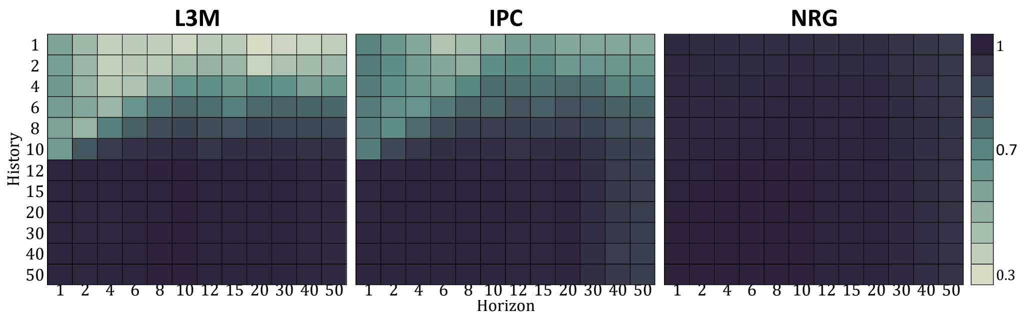


Figure 4.7.. Geomean of *History* vs. *Horizon* accuracy for the LSTM model over all workloads.

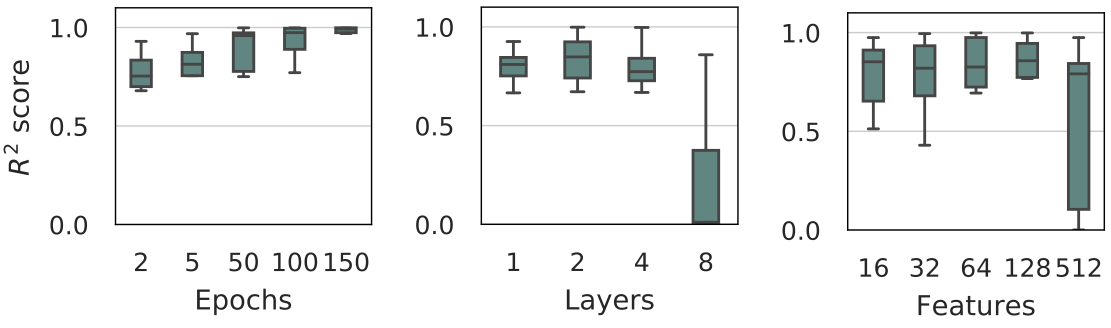


Figure 4.8.. Exploration on the impact of different design pa-rameters on the overall accuracy of the model.

low rank fluctuations, since the major power consumption coefficients are more due to the activation of specific cores, cache memories etc and less due to IPC and L3 misses runtime variances.

How complex should the LSTM architecture be? Since Rusty's design should be as lightweight as possible, we explore and evaluate three different parameters that affect the complexity and precision of the LSTM model, particularly, the numbers of stacked LSTM layers, the number of features per layer and the number of epochs used during training. We use the benchmarks from the scikit suite for this exploration.

Fig. 4.8 illustrates three boxplot diagrams, showing the effect and the corresponding robustness of each examined parameter, w.r.t. the accuracy of the model, assessed through the coefficient of determination - R^2 score. As shown, even though the number of epochs does not affect the complexity of the model, it plays a crucial role in its accuracy, since higher number of epochs increase the precision of the model dramatically, regardless the

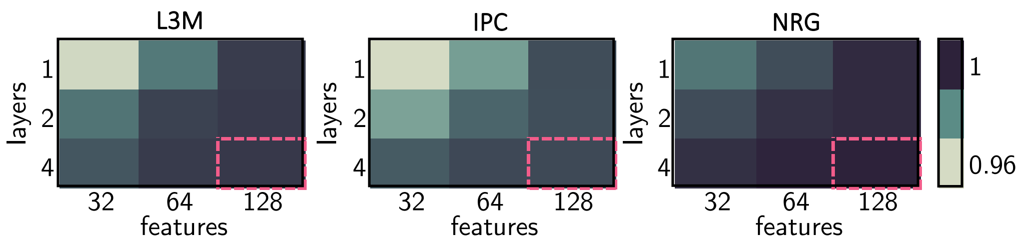


Figure 4.9.. Geometric mean of L3M, IPC and NRG R^2 scores

other design characteristics. In addition, increasing the number of features positively affects the accuracy up to a plateau, after which a significant degradation is observed. The same is true for the number of stacked LSTM layers. Stacking LSTM hidden layers makes the model deeper, thus making the network more eager to recognize certain aspects of input data. Since LSTMs operate on sequencial data, the addition of layers allows the hidden state at each level to operate at different timescale [305]. However, adding frantically more layers can degrade performance of the network. Once the network starts converging, adding more layers can cause its accuracy to get saturated and then degrade rapidly [306].

After filtering out the inefficient configurations, e.g. LSTM layers > 4, we further attempt to determine a single "golden" LSTM architecture, (i.e. the number of layers and features, not the coefficient values of a trained model) that delivers optimized prediction capabilities across the majority of the target workloads. In order to capture the central tendency, we computed the geometric mean of R^2 scores over the various application-specific trained LSTMs configurations of the scikit workloads. As shown in Fig. 4.9, despite the fact that all examined LSTM architectures deliver high geomean R^2 accuracies, i.e. over 0.96, the stacked LSTM network with 4 layers and 128 features per layer outperforms all the other configurations for all the addressed performance metrics.

4.3.3. How to deploy Rusty

Rusty is as an intra-node monitoring system, that predicts future low-level system metrics and, thus, can be replicated across multiple nodes of a cluster to form a realistic solution in a scale-out cluster configuration. Rusty has been integrated with the PCM tool, through `Python` and `C++` embedding, in order to provide runtime predictions, without the need of intermediate logs and files. In addition, all the measured and predicted metrics, from PCM and Rusty respectively, are stored in a `MySQL` [300] database, thus, allowing the integration of our framework with advanced visualization and workload management tools, such as Prometheus [183] or Grafana [307].

Rusty can be utilized under several deployment scenarios, either cloud-native or VM-based from both a cloud-user or cloud-provider perspective. Its only restriction is that of requiring access to model-specific registers (MSR) to monitor low-level system metrics. While historically VMs didn't allow information on low performance counters to be available, recent advances like VMware's virtualized Model-Specific Registers (MSRs) in vSphere technology establish such observability. However, recently VMs are not the only option available for Infrastructure-as-a-Service purchase. Cloud providers made possible to rent bare-metal, physical machines, such as the m5.metal instances on AWS, which were also used later in our evaluation section 4.4.3. In such cases, Rusty can be deployed without any further requirements to provide run-time predictive monitoring on workloads running on the system. From a *cloud-provider* perspective, Rusty can form a advance solution to provide predictive monitoring of the performance of VMs and aid the infrastructure devops so that to enable better Quality of Service to their customers.

Cluster deployment: Figure 4.10 shows how Rusty can be utilized on a cluster of multiple nodes, where the cluster is managed by a master orchestrator (such as Kubernetes) and there is an additional controller on each node of the cluster, which manages resources inside the system according to system state predictions accepted through Rusty. In such a setup, there is a Rusty instance running at each node, responsible for monitoring and making future predictions regarding the specific system's metrics. Specifically, Rusty continuously collects PCM metrics every 100msec and makes future predictions at each decision making interval (as described in section 4.3.1). These metrics are used by both the master/inter-node controller as well as the intra-node controller for decision making regarding management of resources. The inter-node controller leverages such information for more efficient initial placement of newly arriving workloads on the cluster, whereas, the intra-node controller takes advantage of the predictions, to proactively manage resources on the system (e.g. perform dynamic power capping or alter the cache allocation scheme).

Rusty's training modes: Typically, Rusty requires no a priori knowledge of the workloads running on the system. Rusty can be deployed in two ways, either with pre-trained models (pickled object files), which have been trained offline, or in a *stripped*-manner, where the models are trained online, dynamically. This first way covers occasions where custom, personalized models, are necessary, offering higher accuracy for deeper *horizons*. However, this comes with the drawback of not perceiving the real interference the target system might experience. In such situations, interference can be emulated by utilizing synthetic micro-benchmarks for imitating interference, like iBench [3].

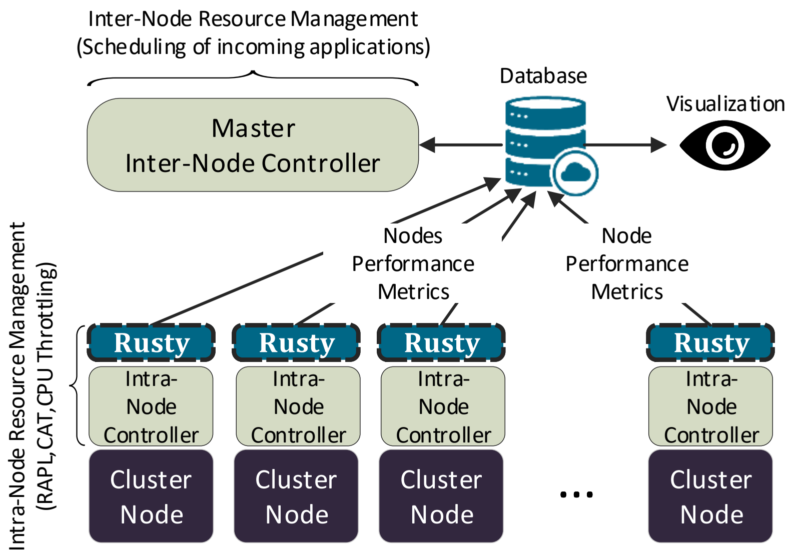


Figure 4.10.. Example of Rusty's integration on a cluster of multiple nodes. Rusty can be integrated with both a potential intra-node controller, to provide per node resource management decisions, as well as an inter-node for better cluster orchestration.

The second way covers occasions, where the LSTM models are trained with real, interference-aware traces, derived directly from the system that Rusty has been deployed to. DC operators have a more holistic view of the underlying infrastructure and workloads running in the cluster and, thus, can build models based on realistic interference effects. In such circumstances, Rusty firstly experiences a *grace period*, during which it collects low-level performance counters from the workloads executing on the system. In a scale-out cluster, where multiple machines execute the same workloads under diverse interference scenarios, the *grace period* can be completed in orders of hours. After the metrics have been collected, Rusty can train either workload-specific models, responsible for more fine-grained predictions, possibly targeting LC workloads, or general, coarse-grained ones, covering a wider range of applications (we show how Rusty extrapolates to new workloads in sec. 4.4.2).

Adapting to abnormalities: Rusty continuously monitors and evaluates the predicted metrics by comparing them with the actual ones. If new workloads arrive on the system, they may expose undiscovered interference effects that have not been revealed during the Rusty's training phase. If a deviation in the prediction efficiency is discovered, then Rusty retrains the model in order to adapt and update the weights and the biases of the model accordingly.

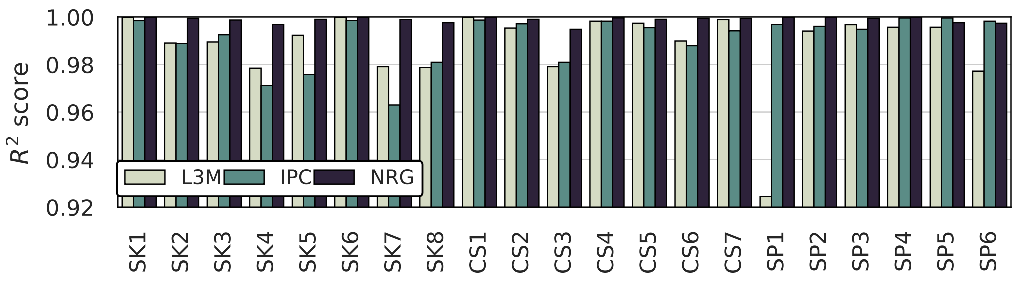
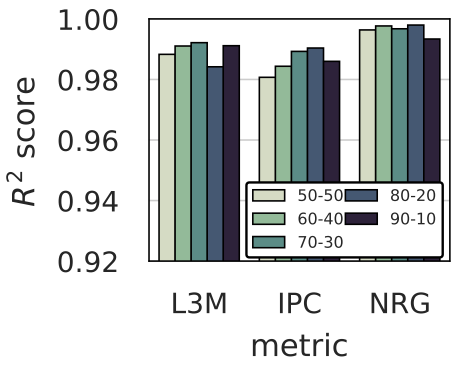


Figure 4.11.. L3M, IPC and NRG R^2 score per benchmark for the overall best configuration (4 stacked LSTM layers and 128 features/layer), *history=50* and *horizon=1*.

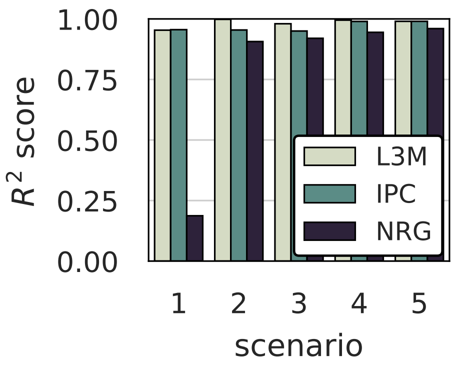
4.4. Evaluation

In this section, we explore Rusty's efficiency utilizing well- established techniques from the machine learning and data engineering domains to validate its efficiency and to enable comparative and reproducible studies w.r.t other SoA solutions. Specifically, we generated several interference scenarios, where each scenario is constituted by a different random number of spawned workloads at random times. We note that the following experimental results are based on workload co-locations as derived from mixing applications found in [83, 283, 284]. We evaluate Rusty on real continuous predictive monitoring deployments, i.e. considering very fine grained time-resolutions in comparison with other LSTM-based based approaches, e.g. Seer [206]. This way we are showing the robustness of Rusty concepts on realistic/pragmatic interference scenarios, in comparison with sections 4.2.3 and 4.3, where we considered synthetic micro-benchmarks from the iBench suite for our analysis.

We partition the datasets produced during execution in two subsets of 90% (training set) and 10% (test set) of the samples respectively. The LSTM models were trained using the PyTorch [142] library. The batch size of our dataset was set equal to 64, the learning rate equal to 0.001 and we utilized ADAM as our optimizer. Furthermore, all the models were trained using the architecture obtained from section 4.3.2 (4 stacked LSTM layers and 128 features per layer).



(a) Rusty's accuracy for different train-test dataset splits.



(b) Rusty's ability to extrapolate to new workloads

Figure 4.12.. Rusty's flexibility on training datasets and workloads.

4.4.1. Rusty's Accuracy

Fig. 4.11 shows the accuracy of Rusty for predicting L3M, IPC and NRG per workload, for a *history* value equal to 50 and *horizon* equal to 1. As we see, Rusty achieves pretty high accuracies from 0.92 up to 0.99 R^2 scores for all the three target prediction variables, with 0.991, 0.988 and 0.994 R^2 score on average for L3M, IPC and NRG, respectively. In addition, it is notable that the considered LSTM architecture, although extracted over the scikit suite, also fits and exposes high predictability on the cloudsuite and spec2006 workloads.

To further validate the robustness of Rusty, we also examine the accuracy for different splits of training and test sets. Fig. 4.12a depicts the average accuracy among all target workloads for training-test split percentages equal to 50-50, 60-40, 70-30, 80-20 and 90-10 of the whole dataset. As shown, Rusty consistently exhibits high accuracies for all our metrics of interest, over different training-test splits, ranging from 0.98 up to 0.998. This forms a very important and promising outcome, showing that Rusty can deliver very high prediction capabilities without resorting to extremely time consuming traversals on the overall LSTM design space each time a new workload is given as input.

4.4.2. Rusty's accuracy on unseen workloads

In order to investigate the ability of Rusty to extrapolate to new workloads, we generated and evaluated 5 different scenarios, regarding the workloads used during training and inference, which are described in Table 4.5. Specifically, for the first 3 scenarios, all but one suites are used as training sets and the remaining one as testing. Since each bench-

Table 4.5.: Different training/inference scenarios

#	Training	Test
1	SK[1-8], SP[1-6]	CS[1-7]
2	SP[1-6], CS[1-7]	SK[1-8]
3	CS[1-7], SK[1-8]	SP[1-6]
4	SK[1-4], SP[1-3], CS[1-3]	SK[5-8], SP[4-6], CS[4-7]
5	SK[5-8], SP[4-6], CS[4-7]	SK[1-4], SP[1-3], CS[1-3]

mark suite captures specific system utilization patterns, we note that the aforementioned scenarios define a set of extremely stressed prediction cases, where the extracted model is called to make inference on unseen patterns, i.e. patterns not trained on. Respectively, in scenarios 4 and 5, the first half of the workloads from each suite are used as training set and the second half as inference, and vice-versa.

Fig. 4.12b depicts the R^2 achieved for all the target prediction variables and for each one of the aforementioned scenarios. This figure also reveals the capability of Rusty to achieve high prediction efficiency as far as the IPC and the L3M metrics are concerned, with an average of 0.965 and 0.988 R^2 scores respectively, among all scenarios. Regarding the NRG metric, we can see that Rusty exhibits high accuracies for scenarios 2-5, but experiences a huge drop on scenario 1. This inaccuracy can be interpreted if we take a close look at Table 4.3. In scenario 1, we used workloads from the scikit-learn and the spec2006 suites as our training dataset and cloudsuite as our test one. Regarding the former workloads (train set) we can observe that the ratio of NRG to IPC is approximately in the range (2,3), whereas for the latter ones (test set) the ratio increases to (5,6). Therefore, in scenario 1, we are enforcing the LSTM network to make predictions on patterns either under- or non-represented in our training data, i.e. reassembling an inefficient/problematic data engineering strategy where careless selection of training set fail to paint the whole picture.

4.4.3. Rusty's accuracy on unseen machine

We further investigate whether Rusty can be directly transferred, i.e. without re-training, among machines with different specifications and micro-architectures. This forms also an extreme stressed evaluation scenario for Rusty, to evaluate its generalization capabilities without adapting the typical full-retraining approach used in other predictive model solutions [206].

We deploy Rusty on a baremetal, m5.metal, machine (to allow access to MSR registers) from Amazon EC2 [72] (Table 4.7). We execute 100 different scenarios, where, at each scenario, a random number of workloads are spawned from the pool of spec2006 and scikit-learn suites.We utilized the Rusty model trained with traces derived from our initial server system (Table 4.1) and evaluate the accuracy of this model on traces from the EC2 instance.

Figure 4.13 shows the R^2 achieved per workload for all the three target metrics, for a *history* value of 50 and a *horizon* of 20. As shown, Rusty experiences a slightly lower, but still high, accuracy, with R^2 scores ranging from 0.76 up to 0.99, experiencing an average

Table 4.7.: AWS EC2 m5.metal specifications

Processor Model	Intel® Xeon® Platinum 8175M
Cores per socket	24 (48 logical) @2.50GHz
Sockets	2
L1 Cache	32KB instr. & 32KB data
L2 Cache	1024K
L3 Cache	33MB, 11-way set-associative
Memory	396GB @2666 MT/s
Operating System	Ubuntu 18.04, kernel v4.15

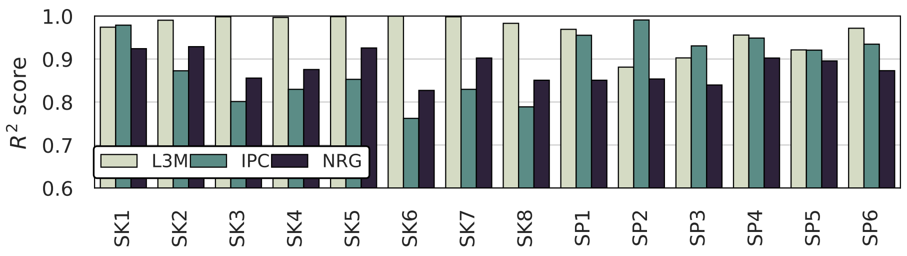
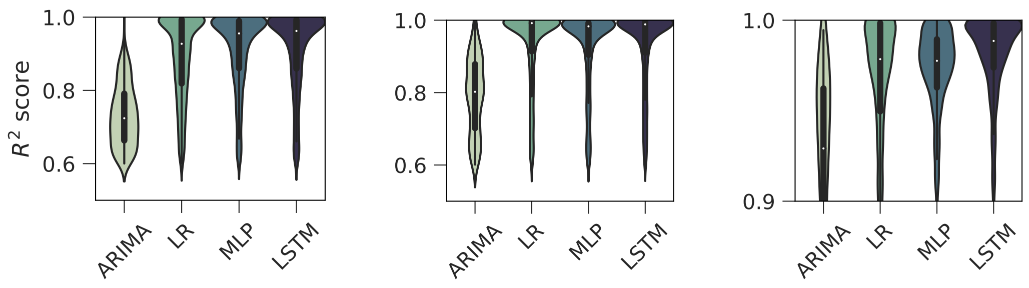


Figure 4.13.. Rusty's ability to extrapolate to new machines.

reduction of 0.01, 0.10 and 0.12 for L3M, IPC and NRG respectively. This experiment reveals that, even though the underlying architecture has changed, there are repeated patterns in the signal traces of the workloads, which Rusty is able to capture, due to the normalization performed during the data pre-process phase.

However, to be able to determine the real values of the metrics, we should maintain the *max* value of each metric used during normalization. Moreover, the accuracy of the model can be further improved by either retraining the network or by applying transfer learning techniques [308]. Nevertheless, this figure reveals the ability of Rusty to extrapolate to new machines, thus forming an ideal monitoring solution for modern data-centers that feature server nodes with differing micro-architecture.

We note that the aforementioned experimental study examined a limited of differing architectures, showcasing as a proof-of-concept Rusty's generalization capabilities. This does not mean that Rusty is transferable retaining its high efficiency among all different architectures available on a data-center without any modifications and/or retraining.



(a) IPC (b) L3M (c) NRG

Figure 4.14.. R^2 score distribution of ARIMA, Linear Regression, Multi-Layer Perceptron ARIMA and LSTM over all workloads for history-horizon pairs ranging in [4,50].

4.4.4. Comparative analysis

To ascertain the superiority of LSTMs over simpler ML models and time-series analysis techniques, we further perform a comparative analysis between the accuracy of our adopted LSTM versus i) an Autoregressive Integrated Moving Average (ARIMA) model, ii) a Linear Regression (LR) model and iii) a Multi-Layer Perceptron Regressor (MLP). LR has been used in prior works for predicting system and higher-level metrics [217,309], whereas MLP was chosen due to its ability to capture non-linear dependencies. For the ARIMA technique we utilized the `auto_arima` function from python's `pyramid.arima` sub-module, where as the LR and MLP models were trained with the `scikit-learn` [283] library. We examine the accuracy of the aforementioned models, trained for differing *history* window sizes as well as prediction *horizons*, both ranging between [4, 50], under various random interference scenarios, i.e. considering randomly co-located workloads from scikit, cloudsuite and spec2006 suites.

Fig. 4.14 shows a violin-plot diagram comparing the four examined models, over all our target workloads and metrics, by considering the distribution of R^2 scores achieved. As shown, LSTM outperforms the other three approaches, providing more robust and accurate predictions, since for all the three examined metrics depicted both the median as well as the 25th and 75th percentiles values of the respective distributions are greater. Moreover, the converged violin shape of the LSTM closer to R^2 scores equal to 1 reveals that even though LR and MLP also provide quite accurate predictions, LSTM tends to behave better providing consistently better forecasting of future metrics. Finally, throughout our experiments we observed that, for deeper horizon values (> 6), the score gap between LR and LSTM rises constantly, showing the deficiency of the linear

Table 4.9.: Rusty's accuracy for additional PCM metrics

L3OCC	**C0-state**	**READ**	**WRITE**	**QPI**
0.988	0.984	0.999	0.998	0.999

model to capture non-linearities in the time-series signal, whereas MLP and LSTM behave almost the same, with LSTM being slightly more accurate in the majority of the cases.

4.4.5. Evaluating Rusty for further PCM metrics

As mentioned in section 4.3.1, Rusty receives as input the low-level metric to be predicted. In this experiment, we evaluate Rusty for additional low-level performance counters provided by the PCM tool. Specifically, we assess the efficiency of Rusty for predicting the LLC occupancy (L3OCC) and the C0-state (C0) of the core executing our target workload, the reads(RD)/writes(WR) of the sockets of the system from/to the memory and also the total Quick Path Interconnect (QPI) traffic of the system for a history of 50 and an horizon of 20 samples, using the overall best architecture obtained from section 4.3.2 and a train-test split of 70%-30% respectively.

Table 4.9 shows the accuracy in terms of R^2 score achieved by Rusty for the aforementioned additional low-level metrics. From this table it is evident that Rusty's methodology is not bound to the explicit low-level performance counters considered throughout the chapter, but can be applied to any metric found in the system. Especially for metrics higher in the system hierarchy (e.g. socket or system metrics) Rusty can provide extremely accurate results, as shown both by the previous evaluation of the socket's energy consumption and also from the accuracy achieved for READs, WRITEs and the QPI traffic. The above analysis shows that Rusty forms an effective universal predictive framework for low-level system metrics, capturing interference from either aggregated traces, e.g. energy, reads or writes etc., found higher in the system hierarchy, up to more primitive ones, closer to the core level, e.g. per-core IPC, L3-occupancy etc.

4.4.6. Rusty's Overhead

Rusty's high frequency monitoring and inference engine inevitably affects the performance of the underlying system. Figure 4.15 provides two heatmaps, showing the average CPU utilization as derived from Unix's `top` command, as well as the average inference time required for Rusty to predict the IPC for different number of OpenMP threads given for inference and cores for which Rusty predicts low-level metrics for. We should note here, that, by default, `top` displays CPU% as a percentage of a single CPU, of the system, thus, on multi-core we can have percentages that are greater than 100%.

These figures reveal that scaling the number of OpenMP threads does not reduce the inference time needed to predict future metrics, while at the same time inficts huge utilization overhead on our system. This is due to the fact that the overhead of dispatching the computation to multiple threads is not counterbalanced by the overall effort required to make the predictions for a single-vector inference batch. In addition, we also see that the inference time needed to make more than 4 predictions on the system cannot follow the high frequency monitoring interval of 100msec. In fact, Rusty requires almost 1 second to predict the IPC for all the 48 available cores on the system. From the above analysis, it is shown that a reasonable strategy regarding Rusty's placement would be to allocate a Rusty monitor in 1 core dedicating around 60% of its resources to predict metrics from 4 other cores of the system. We note that this level resource consumption is not expected to incur any starvation issues in real environments, since recent reports show that the average resource utilization inside data-centers is around 50%-60% [29].

To further evaluate the overhead by Rusty, we also provide the performance degradation imposed to our examined workloads when co-located on the same physical core with PCM-only and Rusty compared to a totally isolated execution. Figure 4.16 shows the per-application slowdown (%). As we see most of the cloudsuite workloads are immune to interference effects imposed by the monitoring tools. The highest slowdown is experienced by application CS2, which uses Apache Spark to run a collaborative filtering algorithm. For all the other cases , we observe a similar pattern on the performance overhead of both PCM and Rusty, with the former imposing an average of 4% and the latter an average of 13% performance overhead respectively.

Finally, for completeness, we also provide the time for changing the cache allocation knob, i.e. CAT policy, and the power capping, using RAPL, of the targeted Intel Xeon server, which is 0.018s and 0.012s on average respectively. Based on the above, we can conclude that the imposed overhead of the proposed methodology is suitable for supporting online decision making and resource allocation.

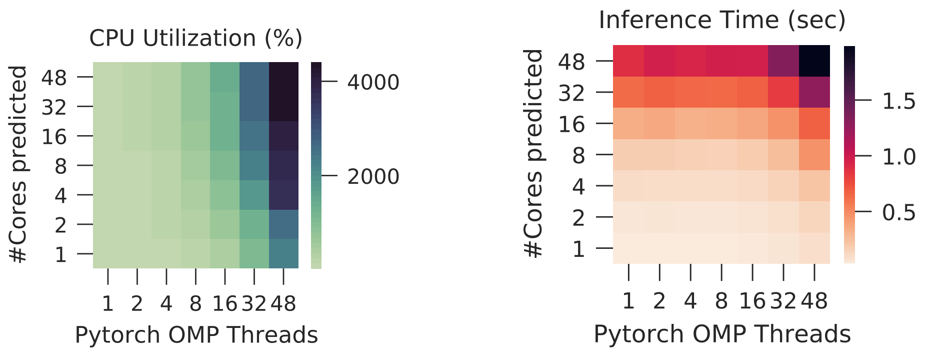


Figure 4.15.. Rusty's CPU utilization overhead and inference time for different number of system cores predicted and OpenMP threads given to Pytorch library.

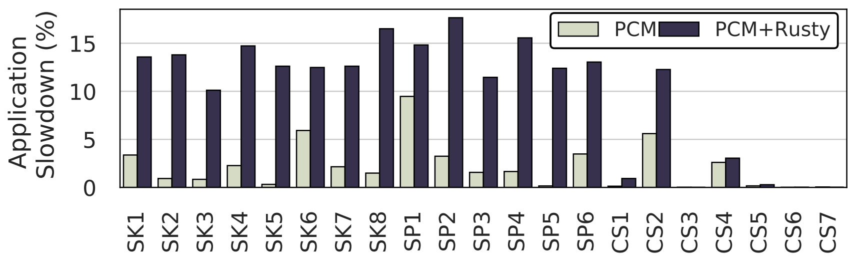


Figure 4.16.. PCM and Rusty performance overhead imposed per examined workload

4.4.7. Correlating lower- with higher-level performance metrics

Predictive monitoring on performance counters provide low level information regarding the state of the underlying system at a per-resource manner, thus allowing us to detect true causes of interference and bottlenecks, i.e. which resource is responsible for the performance degradation our system (and consequently our workloads) is experiencing. However, it is also considered of great importance, these low level insights to be projected to higher-level metrics of interest, such as the slowdown applications experience, upcoming QoS violations etc. In [206], the authors showed that predictive classification of QoS violations is possible by exploiting the raw low-level performance counters. However, gaining more insight, e.g. accurately predicting the actual slowdown of a workload is a much more difficult regression problem, which requires additional models to be applied on top of them.

We showcase the importance but also the applicability of exploiting this low level monitoring information for slowdown predictability, through a rather simple exemplary scenario of training a Multi-Linear Perceptron (MLP) regressor that receives low-level metrics of the workload executed under interference as input features and predicts the respective slowdown experienced. To determine the primary inputs of our model, we evaluated the correlation between application slowdown, defined in this case as $\frac{T_{isolated}}{T_{interfered}}$, and various performance monitoring distribution statistics, and utilize the most highly correlated features, namely the degradation of the mean and median IPC, L3M and LLC occupancy compared to the isolated execution. Our model consists of 6 layers and 64 features per layer. Moreover, our dataset consists of 100 instances per suite's workload, each instance corresponding to execution with diverse interference, where 90% of the dataset is used as training and 10% as test set respectively. We also performed a *6-fold* cross-validation to examine the robustness of the model. Figure 4.17 illustrates the residuals of the regression for the scikit-learn and cloudsuite benchmarks, where the x and y axes, show the real and predictive slowdown the application experienced. As demonstrated, the MLP predictor can quite effectively forecast the slowdown the workloads experienced, with a Mean Squared Error (MSE) of 0.003 for the scikit-learn and 0.03 for the spec2006 suite[1]. Although rather simple in implementation complexity, this exemplary case shows the potential of exposing low-level performance counters to higher-level metrics of interest, forming a promising solution for runtime control using low-level metrics.

[1]For Cloudsuite, the MLP regressor did not provide the similar accuracy results, thus, more descriptive and deep networks might be required, which, however, are out of the scope of this work.

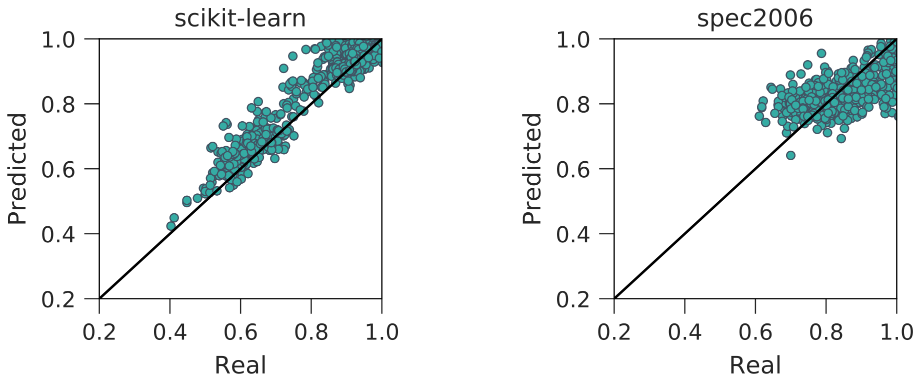
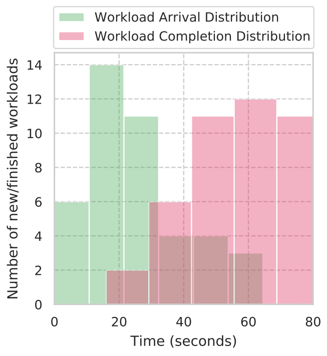


Figure 4.17.. Slowdown prediction residuals.

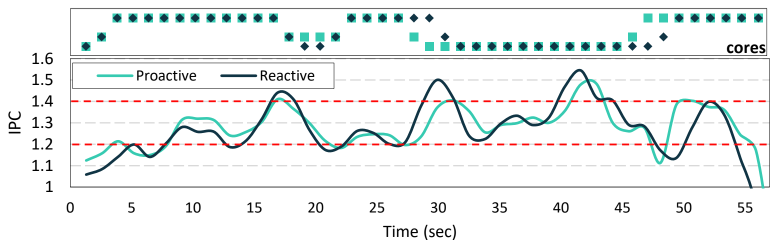
4.4.8. Advantage of predictive monitoring for QoS guarantees

We evaluate the efficacy of predictive monitoring through a simple, but characteristic example. Specifically, we evaluate the ability of a proactive vs. a reactive controller for managing a Service-Level-Agreenment/Quality-of-Service (SLA/QoS) of an application running on the system. For this particular example we examine CS2 as our target application, which uses Apache Spark and runs a collaborative filtering algorithm in-memory on a dataset of user-movie ratings. In addition, as SLA we consider the Instructions-Per-Cycle achieved by the application, which can be considered as a proxy for the actual performance of the application. We set a lower and upper IPC threshold equal to 1.2 and 1.4 respectively. For the reactive controller, we calculate the average IPC over the last 2 seconds and once the former threshold is violated we increase the number of cores given on the application to the next available value in the set [1,2,4], while if the latter threshold is surpassed we decrease the cores respectively. On the other hand, for the predictive controller, we predict and calculate the average IPC of the next 2 seconds (horizon value equal to 20) and perform the same actions as above based on the result. Finally, both controllers have a decision making interval equal to 10 samples, i.e. the decision for boosting/relaxing the application is repeatedly taken every 1 second.

To imitate interference through the experiment, we utilize iBench, by spawning 42 random workloads from the pool of the available ones. The arrival and completion time of each iBench workload is determined by a Poisson distribution with lambda value equal to 2, in order to emulate periods of disparate interference. Figure 4.18a depicts the arrival and completion distributions used in this experiment. As we can see the system first experiences a period of low-interference, followed by a period of rapid stress, which then



(a) iBench jobs arrival and departure distribution



(b) Mean IPC traces for reactive and proactive controllers. The scatter plot shows the number of cores given to the workload at each point in time. The red lines show the respective IPC threshold.

Figure 4.18.. Reactive vs. Proactive SLA-driven control of Spark workload.

gradually declines.

Figure 4.18b shows the respective traces for the complete execution of the spark workload, tuned by the reactive and proactive controllers. The red dotted lines show the IPC thresholds, whereas the scatter plot on top shows the number of cores given to the Spark workload at each point in time (lower is 1, medium 2 and higher 4). This figure reveals the advantage of the proactive approach to avoid IPC violations when feasible. Specifically, we can observe two clear points in the graph, where the proactive approach achieves to sustain the SLA agreement opposed to the reactive one, i.e. point ~ 23 and point ~ 32 in time. In these two points, the "intelligence" of the proactive controller becomes apparent, by keeping steady/scaling down the number of cores given to the Spark workload, in order to avoid violating the lower and upper threshold respectively. In addition, through this figure we also see that there are certain violations which are inevitable, as for example the one happened in point ~ 48 in time. However, even in such cases, we can see the attempt of the proactive controller to pre-increase the number of the cores given to the examined application. As expected, the overall execution time in the proactively regulated case is larger, since we apply the SLA also to the upper achievable IPC.

4.5. Conclusion

Run-time predictability of systems under interference is essential for better management of resources in modern large-scale cloud data centers. We proposed Rusty, a lightweight LSTM-based predictive monitoring framework able to provide fast, accurate, non-intermittent and interference-aware predictions of low-level system metrics in multi-tenant systems. We analyzed and explored several schemes of LSTMs concluding to a generic efficient LSTM architecture in terms of accuracy, responsiveness to run-time constraints and computational cost. We showed that Rusty forms a realistic solution achieving extremely high prediction accuracy R^2 of 0.98 on average under pragmatic workload consolidation scenarios driving modern cloud platforms and also that it satisfies the strict latency constraints imposed by low-level system knob activation. We foresee Rusty to be integrated in existing orchestration frameworks, capturing complex system dynamics and assisting towards more elaborated resource allocation decisions.

Chapter 5.

Deep-Learning Driven, Interference-Aware Memory Orchestration for Disaggregated Cloud Infrastructures

Efficient utilization of resources forms a top priority for data-center providers, as it provides both for better performance delivered to end-users and enhanced cost-efficiency. To tackle the problem of resource under-utilization, workload co-location has become the de-facto approach for hosting applications in such environments. On top of that, hardware disaggregation is introduced as a novel paradigm, which allows fine-grained and dynamic tailoring of cloud resources to the characteristics of the deployed workloads. Towards the realization of hardware disaggregated clouds, novel orchestration frameworks must provide additional knobs to manage the increased scheduling complexity. In this chapter, we present Adrias, an orchestration framework leveraging predictive monitoring for memory disaggregated cloud systems. Adrias leverages low-level performance events and applies deep learning techniques to effectively predict the system status and performance of arriving workloads on memory disaggregated systems, thus, driving cognitive scheduling between local and remote memory allocation modes. We evaluate Adrias on a state-of-art disaggregated testbed and show that it achieves approximately 0.99 and 0.942 R^2 accuracy for system state and application's performance prediction respectively. Moreover, Adrias manages to effectively utilize disaggregated memory, by offloading almost 1/3 of deployed applications with less than 15% performance overhead compared to a conventional local memory scheduling, while clearly outperforms naive scheduling approaches (random and round-robin), by providing up to ×2 better performance.

5.1. Introduction

Over the last years, cloud computing has been established as the new standard for application deployment, and is expected to grow even more in the near future, due to the flexibility and cost effectiveness it offers [310]. From the providers' point of view, maximizing performance delivered to customers while minimizing the total cost of ownership (TCO) is a first-class system design concern. However, such conflicting requirements are difficult to achieve, since maximizing performance requires isolated execution of workloads, which, however, leads to high resource under-utilization [29, 30, 311].

To tackle this problem, cloud providers have adopted workload co-location and multi-tenancy [40, 312] as their de-facto deployment model. While this approach improves the overall resource utilization of cloud datacenters, it also leads to interference in shared resources, which in turn induces variability and degradation in applications' performance [41, 88, 116, 313]. As a result, intelligent orchestration and allocation of computing resources have been developed both from academia [41, 63, 64] and industry [117], where complex cluster-wide software mechanisms control how hardware resources are assigned to applications.

Cloud systems typically involve two layers of resource management which are orthogonal to each other and can be applied independently, i.e., *L1)* The initial static allocation of resources and placement of incoming applications (so called resource orchestration) and *L2)* the dynamic adjustment of allocated resources to meet requirements of applications (so called runtime management). Mechanisms in the first category should be able to identify the resource requirements of incoming (possibly unknown) applications, and avoid placements that lead to resource interference, while also considering the underlying HW heterogeneity [28, 41, 73, 88, 314]. The second layer includes runtime mechanisms that dynamically optimize the performance of running applications, such as SW controllers [40, 315] and/or OS-integrated extensions [241, 316] that regulate resources of deployed applications.

Despite their sophistication, software only mechanisms have proven incapable of fully resolving the resources under-utilization problem, that is mostly a bi-product of the diverse computational requirements of cloud workloads combined with the fixed resource proportionality of cloud-server systems. As a consequence, it is common in modern datacenters to observe a fragmentation of resources that are available yet not consumable by any workload [317, 318].

To overcome this resource wall challenge, *hardware disaggregation* has been proposed as a new design paradigm [44, 319–321]. In a disaggregated cloud, the underlying hardware infrastructure is organized as a pool of heterogeneous resources that can be composed on-demand into compute units tailored on workload-specific requirements. Recent scientific research has examined the applicability of the disaggregated concept on several components of modern Cloud and HPC infrastructures, including processing units [168,322–324], memory [1, 43, 325, 326] and storage [324, 327–329]. In addition, rack-scale operating systems [330] and runtimes [331] have been proposed for managing disaggregated resources. Typically, disaggregated systems either package hardware resources in one big "case" and connect them with buses like PCIe [168, 332], or rely on high-bandwidth, physical network interconnections [1, 244, 324, 330], optical switches [333, 334] coupled with software APIs [330, 335, 336] used to expose and interact with the available cluster resources.

As a result, even though hardware disaggregation offers more fine-grained organization of computing resources, leading to improved resource utilization, elasticity, heterogeneity and failure isolation, it also introduces new optimization knobs (e.g., selection between local vs remote memory allocation), which have to be properly managed to provide increased resource efficiency. Especially for memory disaggregated cloud systems, orchestrated access to memory resources is required given that applications performance can be severely impacted due to the increased latency in accessing remote memory [337], while memory access patterns often reveal unpredictable fluctuations throughout execution [162, 208]. Moreover, recent research has revealed that the binary code footprint of cloud workloads is one to two orders of magnitude larger than the L1 instruction cache and can lead to repeating instruction cache misses, hurting performance [288, 289], whereas latest reports from large-scale clouds show that memory is becoming the new performance bottleneck [29].

Need for memory orchestration: This work focuses on memory-disaggregated infrastructures addressing the newly induced problem of interference-aware memory orchestration. In a memory disaggregated system, orchestrated placement to memory resources is required to minimize the impact on applications performance due to the increased latency in accessing remote memory [337]. While prior research efforts have thoroughly examined dynamic runtime mechanisms (L2 – e.g., page migration/prefetching) for memory-disaggregated and multi-tiered memory systems [239, 316, 338], limited work has been conducted with respect to the problem of interference-aware memory placement in disaggregated clouds (L1), since existing scheduling approaches [41, 88, 339] neither target memory orchestration nor been have extensively examined for such composable systems. However, orchestration of memory resources in disaggregated environments is critical for two main reasons: *i)* In presence of interference, determining an efficient memory mapping can significantly improve the overall performance of the application and *ii)* Optimal initial allocation of memory can minimize the amount of data travelling back and forth through the network, e.g., in applications that present a low ratio of hot versus cold pages

Adrias Contributions: In this chapter we introduce *Adrias*[1], an interference-aware memory orchestration framework that enables effective/optimized data placement decisions on memory-disaggregated cloud infrastructures. The key features of Adrias could be summarized through: *i)* its ability to forecast the tendency of system-wide metrics in the future, thus driving proactive memory orchestration decisions; *ii)* its accurate performance predictions for deployed applications w.r.t. memory heterogeneity (local/fast vs. remote/slow DRAM) and interference and *iii)* its power to leverage disaggregated memory with minimal impact on the performance of deployed applications without the employment of dynamic memory management mechanisms. Adrias exploits system-level performance monitoring information and leverages deep learning approaches to place incoming applications on the pool of available memory resources. To the best of our knowledge, this is the first work tackling the problem of interference-aware memory orchestration, i.e. applications' data placement on memory-disaggregated cloud systems. Our main contributions are:

- We perform an in-depth exploration and provide new insights on the performance sensitivities and capabilities of the state-of-art ThymesisFlow disaggrageted memory testbed [1]. We characterize ThymesisFlow testbed under various interference scenarios for a set of in-memory cloud workloads, namely Redis object storage, Memcached key-value store and several Spark analytics and analyze the impact of memory disaggregation w.r.t. their performance.

- We propose two deep learning models tailored to disaggregated memory systems; *i)* a system state prediction model that forecasts the tendency of monitored performance events in the future and *ii)* a performance prediction model that estimates the performance of applications, when deployed on memory disaggregated systems. Using these models, we are able to accurately predict the tendency of system metrics and performance of incoming applications, achieving up to 0.999 and 0.942 R^2 scores respectively.

- We present Adrias, an orchestration framework for memory disaggregated systems. By leveraging the developed prediction models and a naive, yet effective, scheduling logic, Adrias employs remote memory efficiently, by offloading up to 35% of best-effort applications with less than 15% performance degradation compared to a local DRAM memory allocation approach and also provide comparable QoS guarantees

[1]Adrias was a WWII battleship that hit an underwater mine and was split in half. In spite of the damage suffered, Adrias managed to survive.

for latency-critical ones.

The rest of this chapter is organized as follows. Section 5.2 gives a brief overview of the memory-disaggregated testbed used to design and evaluate Adrias. Section 5.3 describes the experimental setup of this work and also presents an extensive characterization of a real memory disaggregated testbed along with a set of realistic in-memory applications. Section 5.4 presents Adrias, our proposed orchestration framework for memory disaggregated systems and finally, sections 5.5 and 5.7 show our experimental results and conclude the chapter respectively.

5.2. Disaggregated Memory Testbed

Memory disaggregation is not a new subject of study, with several approaches appearing over the years. There are fully software-based approaches that expose remote memory as Linux swap devices or rely on RDMA transfers to be explicitly programmed for moving memory blocks to from/remote memory [326, 330, 336, 340–343]. Efficient use of RDMAs often involves having to reserve and pin chunks of memory beforehand that leads to inefficient utilization of memory resources. A number of full hardware solutions have also been proposed [1, 331, 344] that although different, they are all mostly based on intercepting low level CPU memory operations to process and forward them towards remote systems. In this work we focus on the ThymesisFlow open source hardware [1] that incurs minimal software overheads, and it is easy to integrate with applications and operating systems.

Prototype Setup: We replicate the disaggregated system described in [1]. Specifically, all of our experiments have been performed on the ThymesisFlow [345] open-source real memory-disaggregated testbed prototype consisting of two POWER9 servers, the specifications of which are shown in Table 5.1. Both servers run RedHat Enterprise Linux (Kernel `v5.8.0`) and are equipped with an Alpha Data 9V3 card, that features a Xilinx Ultrascale FPGA. The FPGAs are connected back-to-back using a single copper cable that models a 100Mbps point-to-point connection in a circuit switched disaggregation network fabric.

Hardware Architecture: Figure 5.1 depicts the hardware infrastructure of the ThymesisFlow prototype. On both servers the FPGAs are interfaced to the CPU bus using the OpenCAPI [346] prototcol, that enables among other things coherent access to the CPU memory from an accelerator. The interface between CPU and FPGA is based on 8x25Gbps serial links for a total of 200Gbps. On the borrowing side, the OpenCAPI

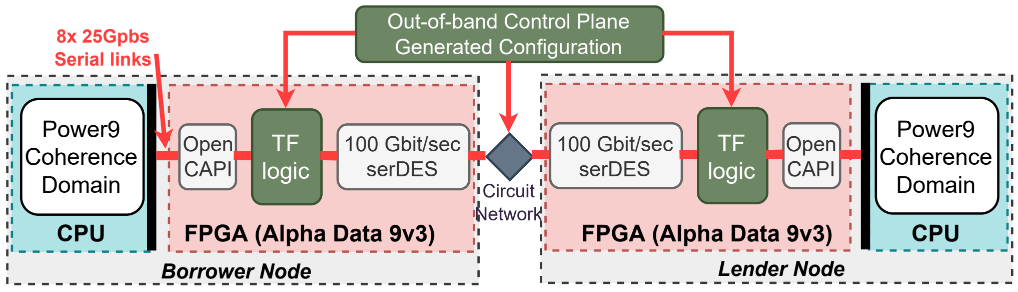


Figure 5.1.. ThymesisFlow [1] architecture overview

accelerator is mapped at a specific physical location in the CPU bus that can be then hot-plugged as regular memory. On the lender side the OpenCAPI accelerator accesses the memory on behalf of the borrower by totally bypassing the lender CPU, thus avoiding any unnecessary overhead. ThymesisFlow enables byte-addressable disaggregated memory that does not require any software support such as in the case of solutions based on RDMA [239, 326] and those enabling disaggregated memory by means of a Linux swap device [340, 342]. Every time a memory access performed at the borrower side causes a last level cache miss or flush, the cache line is refilled/flushed via remote memory. An OpenCAPI transaction (read or write) is generated by the borrowing CPU and received by the FPGA. Here the transaction address is modified to a valid one at the lender side and the operation traverses the circuit network (100Gbps) towards the FPGA at the remote node, where the OpenCAPI transaction is re-issued on the bus towards the memory. Responses in case of a read follow the reverse path.

Software Architecture: From the software standpoint, the two servers in the prototype are not symmetric. ThymesisFlow enables the borrower-lender model, where the lender

Table 5.1.: Target System specifications

Processor Model	IBM POWER9
Cores per socket	16 (64 logical) @3.7GHz
Sockets	2
L1 Cache	32KB instr. & 32KB data
L2 Cache	512KB
L3 Cache	10MB 20-way set associative
Memory	1.2TB @2666MHz
Operating System	RHEL Linux v8.2

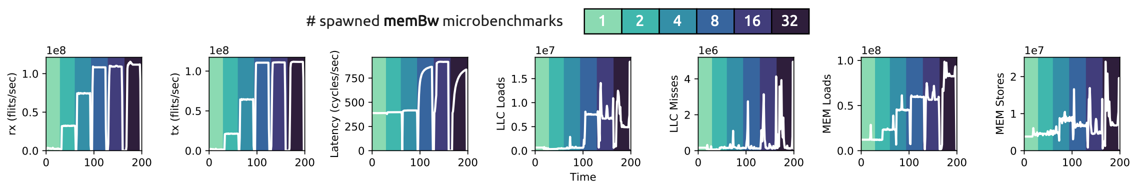


Figure 5.2.. Capacity of our memory disaggregated testbed for different stressing scenarios. Different color shades denote different number of memory bandwidth stressing microbenchmarks [2] deployed on remote memory mode.

gives away part of its local memory for access from a remote borrower. ThymesisFlow provides `libthymesisflow` for managing the attachment and detachment of disaggregated memory. `libthymesisflow` is based on a user-space daemon running on each node and a cli tool used for requesting attachment/detachment of memory. On the borrower server, ThymesisFlow exposes disaggregated memory as a CPU-less NUMA node. This allows users to avail of all Linux default NUMA-aware functionalities and tools. Users can alternatively decide to either hotplug the disaggregated memory to the running Linux system in the borrower server, or to keep it out from the Linux kernel memory management system for using custom (user-provided) memory allocators. In this work we hotplug the disaggregated memory and we control how applications access it by means of special Linux crgoups.

Disaggregation Modes: In this work, we examine only two out of the four available allocation mechanisms of ThymesisFlow, i.e., *local:* all memory requests are served locally and *disaggregated:* all memory needs of the application are satisfied by memory borrowed from a neighbor node.

5.3. Analysing memory disaggregation under interference

In this section, we perform an interference-aware analysis of the ThymesisFlow memory disaggregation testbed. We unveil important insights concerning both potential hardware limits of memory disaggregated systems, as well as resource contention impact on the performance of applications leveraging disaggregated memory allocations.

5.3.1. Examined Workloads

Cloud infrastructures typically host two types of workloads, best-effort (BE) and latency-critical (LC) ones. The former require the highest possible throughput while the latter have strict QoS guarantees. To cover both types of applications, we examine the following open-source, in-memory applications:

- **Redis (LC):** Redis [104] is a NoSQL key/value store that keeps data in memory and is used primarily as an application cache or quick-response database. We examine the impact of local and remote memory allocation modes on Redis server instances, serving user requests. We generate requests using the `memtier_benchmark` [347], a tool for load generation and bechmarking of NoSQL key-value databases. For the majority of our experiments (except section 5.3.3) we consider a constant load of 200 clients with 10000 requests per client and a SET:GET ratio of 1:10.

- **Memcached (LC):** Memcached [105] is a distributed memory object caching system, used to cache data and objects in the main memory. Similar to the Redis case, we examine the impact of disaggregated memory on Memcached server instances and we generate requests using the `memtier_benchmark` tool [347]. We consider 200 clients, 40000 requests per client and a SET:GET ratio of 1:10.

- **Spark in-memory analytics (BE):** Apache Spark [245] is an open-source unified analytics engine for large-scale data processing. We examine 17 different spark applications derived from the HiBench benchmark suite [2], namely `als`, `bayes`, `gbt`, `gmm`, `kmeans`, `lda`, `lr`, `nweight`, `pagerank`, `pca`, `repartition`, `rf`, `sort`, `svd`, `svm`, `terasort` and `wordcount`. These applications are long-running, best-effort (BE) ones and cover representative workloads from different domains, i.e., graph processing, machine learning and web searching. We run every application using the default Spark parameter configuration and the *small* dataset as described in [2]. We study the impact of disaggregated memory only on the executor processes, which perform all the task computations.

In order to emulate real-life Cloud deployments, all the referenced benchmarks running on the system have been containerized using the Docker engine [286].

Load generation: We use asymmetric load generation on the underlying system, by co-locating LC, BE and iBench [3] interference microbenchmarks, as described in Section5.4.2. For LC workloads, we utilize the `memtier_benchmark` [347], i.e. the official Redis Labs' utility for load generation of NoSQL key-value databases. Tail latency is measured using a set of closed-loop memtier clients [348] over asymmetric co-located workloads. More in detail, our setup spawns 4 threads, where each thread spawns 200 clients, i.e. eliminating client-bias [349, 350]. We use a SET:GET ratio of 1:10 and generate a constant load of 10000 and 40000 requests per client. This configuration leads to a total of approximately 30.000 and 100.000 operations served per second for Redis and Memcached respectively, which closely relate to realistic loads found in production, e.g., Facebook services [351, 352].

5.3.2. Limits of HW memory disaggregation on ThymesisFlow

We first assess the capacity of our prototype while performing memory operations. The goal of this characterization is identifying system and cpu metrics related to memory accesses that impact the performance of applications. For this purpose, we utilize the iBench benchmarking suite [3], that provides a set of interference micro-benchmarks, each of which stresses a different resource on a multi-core server. For this analysis, we examine the effect of repeated data movement between the local and remote system; in more detail, we spawn the `memBW` micro-benchmark that is designed for testing the memory bandwidth of a system. We run an increasing number of `memBW` instances (1 to 32) to test different memory bandwidth requirements, and forcing it in using memory borrowed from the remote system, thus generating memory traffic on the communication between the FPGA devices in the ThymesisFlow prototype. In addition to the traffic between the FPGAs, we also track utilization of resources in the local node memory system. Specifically, we measure the number of flits (32B) received (`rx`) and transmitted (`tx`), and the average latency on the ThymesisFlow communication channel. Whereas, for the local system we gather the Last Level Cache (LLC) loads and misses, and the memory loads and stores. Figure 5.2 shows the results and also reveals three important remarks (R1-R3).

R1) Bounded throughput: The throughput (in terms of flits/sec) of the disaggregated memory reveals an upper bound in the amount of both transmitted and received data, with a cap of approximately 320MBps[2]. This value reveals that remote memory has approximately a three orders of magnitude lower bandwidth threshold compared to conventional DDR4 memory systems, which can support a theoretical of 120 GBps sustained memory bandwidth [353].

R2) Communication latency: As shown, low to medium amounts of generated traffic (1 up to 4 memory bandwidth microbenchmarks) the average latency on the communication channel follows a steady state of approximately 350 cycles/sec. However, in cases

[2]Given that each flit is 32B, we compute `rx` and `tx` bandwidth in B/s by multiplying with 32 [1].

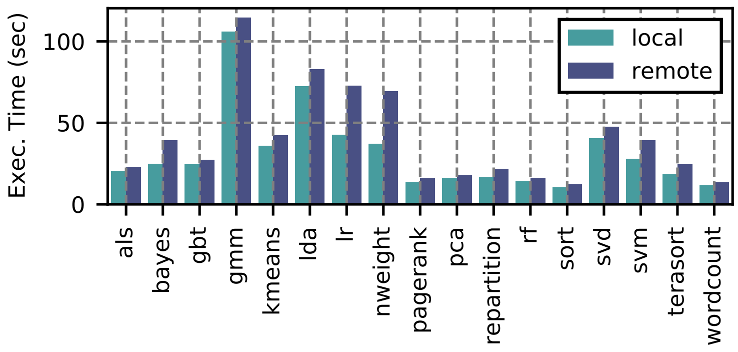


Figure 5.3.. Spark performance characterization (total execution time) when executed in isolation on local and remote memory.

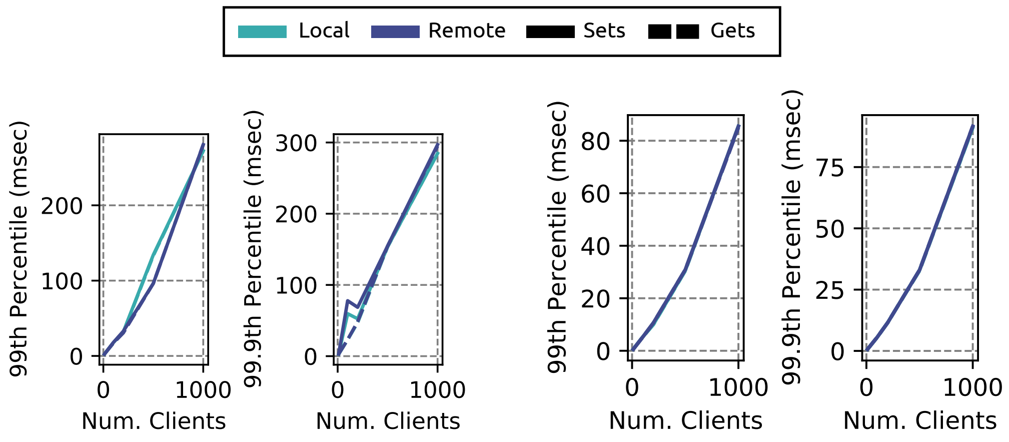
of increased traffic (8 up to 32 micro-benchmarks) this latency is almost tripled, reaching a yield plateau of 900 cycles/sec. Until the 4 micro-benchmarks mark, the ThymesisFlow prototype is capable of handling the memory requests received and the latency remains constant across all executions, while bandwidth increases steadily. From 8 micro-benchmarks onwards, the ThymesisFlow channel is saturated (bandwidth plateau) and the back-pressure mechanism implemented in the FPGAs starts delaying memory transactions, hence the step in latency.

R3) Local system interference: As shown, application deployment on remote memory also induces interference on the memory hierarchy of the local system. This is expected for chip-level metrics (e.g., LLC Loads and Misses), as the cache memory hierarchy lies beneath the abstraction layer of local/remote memory accesses. Regarding memory loads and stores, remote pages are memory-mapped and handled through an enhanced `numactl` memory controller and, thus, all remote traffic is handled on-chip by memory controllers of the local node.

5.3.3. Workload characterization

We further quantify the impact of local and remote allocations on the performance of our applications when executed in isolation and under different interference scenarios.

Execution in isolation: For LC applications (Redis, Memcached), we examine the 99*th* and 99.9*th* response percentiles (tail latency) under different loads by scaling the number of clients requesting `Set` and `Get` operations. For BE (Spark) ones, we examine the execu-



(a) Redis (b) Memcached

Figure 5.4.. Tail latency of LC applications with increasing requests per second when executed in isolation on local and remote memory.

tion time of applications for the two memory allocation modes. Figure 5.4 and Figure 5.3 show the results for LC and BE applications respectively.

R4) Non-uniform performance variation: For Redis and Memcached applications we confirm the results reported in prior work [1] that local and remote memory allocation modes provide almost identical curves in terms of tail latency, for all the configurations examined. This similarity in performance was also expected due to the fact that in-memory caches typically perform many small read and write memory accesses with minimal bandwidth pressure requirements, which can be easily attained based on the specifications of our disaggregated system, as shown in Section 5.3.2. On the other hand, for Spark applications we notice a different performance pattern. Specifically, we observe an average degradation of 20% over all our examined benchmarks. However, the performance gap measured is not constant across all the Spark applications tested, showing that remote memory is suitable for some applications while it is not the best options for others. For example in Figure 5.3, we observe that `nweight` and `lr` suffer almost a ×2 slowdown when ran on remote memory, whereas others, such as `gmm` and `pca` experience less than 10% performance degradation.

Execution under interference: Last, we examine the sensitivity of our considered workloads with respect to different interference effects. Specifically, we investigate the relative impact of resource contention on different levels of the system hierarchy, between local and remote memory allocation modes. Similar to Section 5.3.2, we make use of the iBench [3] suite. First, we deploy our target application and measure its performance on local and remote memory under the four interference scenarios namely: `cpu`, `l2`, `l3` and `memBw`. For both modes (local and remote), we spawn a different number (1 up to 16) of resource trashing micro-benchmarks, targeting different resources on the system (CPU, L2-cache, Last-Level-Cache and Memory Bandwidth), i.e., if the application is deployed on local memory, so are the ibench microbenchmarks and vice-versa. Figure 5.5 shows the respective results, where the density of each cell depicts the performance slowdown of the respective scenario between local and remote memory, with darker colors denoting a greater gap. This figure reveals three major insights:

R5) Performance chasm under contention: We observe that after a certain threshold (typically 16 for `L3` and > 8 for `memBw` micro-benchmarks), the same amount of interference results in much higher performance degradation on the remote memory, reaching up to $\times 4$ additional slowdown in certain cases. Combined with the results presented in Figure5.2, we observe that this threshold corresponds to the saturation point in the communication channel of the FPGAs. This is true both for BE and LC applications, with the latter appearing to be more resistant on interference effects. This shows that remote memory gets saturated much more easily than local DRAM, which also confirms our observation regarding the limitation of remote memory bandwidth made in Section 5.3.2.

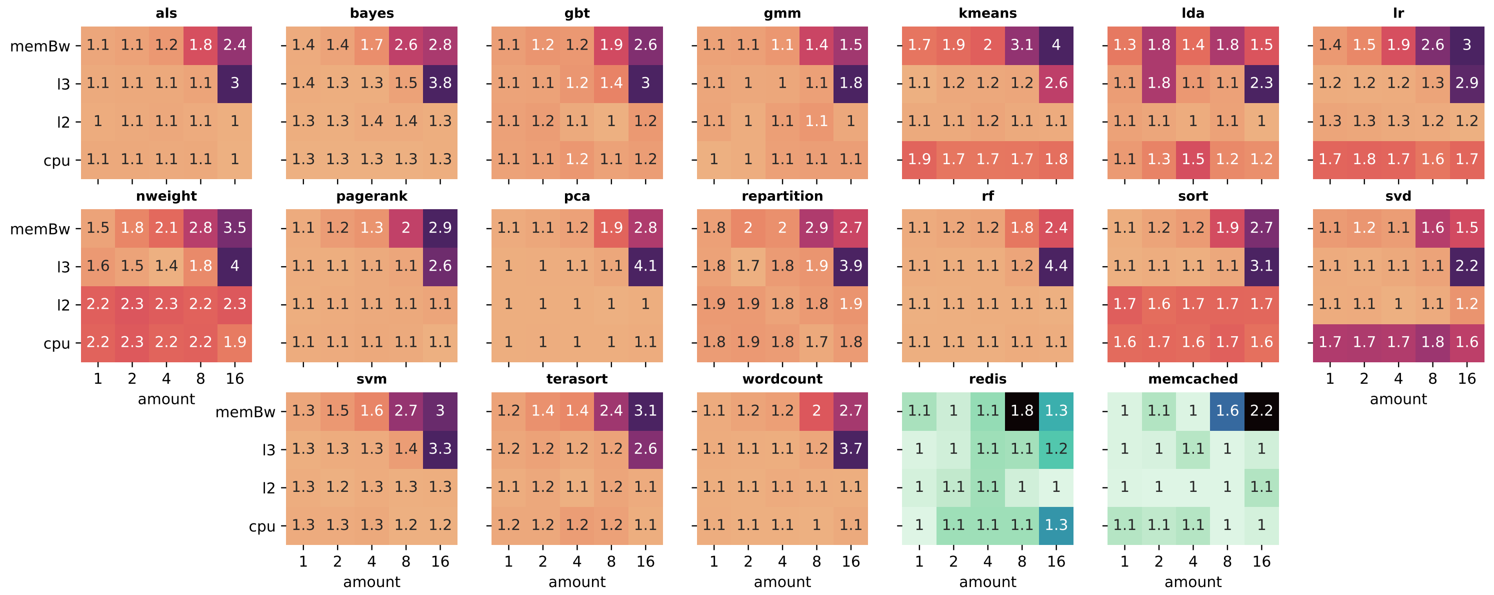


Figure 5.5.. Benchmark performance characterization under interference on local and remote memory. The heatmap density denotes the execution time and 99th percentile slowdown of remote vs. local execution for BE (orange) and LC (teal) applications respectively, under different interference scenarios (`cpu`, `l2`, `l3`, `memBw`) using ibench [3] contention microbenchmarks.

R6) LLC vitality: We see that contention on the LLC has the greatest negative impact for the majority of the BE applications. Typically, interference on the LLC leads to consecutive misses, which in turn are translated to increased memory bandwidth due to read/write accesses from/to the main memory. However, while both LLC and memory bandwidth end up in generating traffic on the channel, we see that data locality and caching is of paramount importance, as intense LLC contention (16 spawned microbenchmarks) leads to the worst possible performance degradation for the majority of the Spark applications. Moreover, a sustained and increased interference effect on the memory network bus, leads to gradual performance degradation, relative to the extent of the underlying interference. Last, since in-memory databases rely heavily on pointer chasing operations, which introduce poor on-chip spatial locality [354, 355], they appear to be less cache sensitive, revealing higher response times only on memory bandwidth interference scenarios.

R7) Stacking interference effects: For certain benchmarks (e.g., `nweight`, `sort`, `kmeans`), we also notice a performance gap between local and remote memory also when imposing interference on lower levels of the system hierarchy (i.e., CPU and L2 cache). We call this a *stacking interference effect*. For such applications, we expect the remote memory allocation mode to be a prohibitive option under realistic scenarios, where different resources are congested simultaneously.

5.3.4. Affinity of system & workload metrics

As shown in Section 5.3.2, low-level system metrics can provide insightful information regarding the state of the underlying disaggregated system. Taking also into account the high performance variability shown in Figure 5.5, it is evident that being able to project low-level performance events to higher-level metrics of interest, such as applications slowdown or increased transactions latency, would allow us to estimate application performance solely through the assessment of lower-level metrics. To investigate whether such a relationship exists, we examine the correlation between low-level system and high-level application metrics prior and during execution when deployed using the remote memory allocation mode. Specifically, we generate several deployment scenarios (similar to the ones described in Section 5.4.2) by randomly co-locating different ibench workloads with the examined benchmarks, and we keep track of the underlying system metrics during execution.

We evaluate the linear correlation between the average system performance metrics 120 seconds prior to application scheduling (x), as well as during execution ($\tilde{x}$), with the application performance, using the Pearson's correlation coefficient. For Spark applications

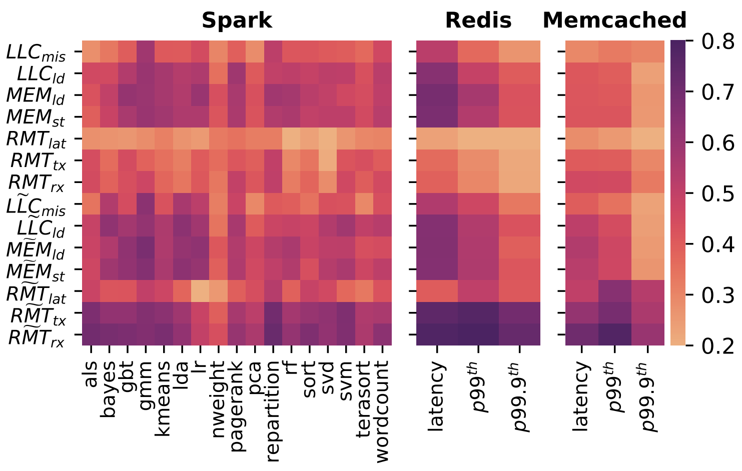


Figure 5.6.. Correlation between historical (x) and runtime ($\tilde{x}$) system performance events and performance of deployed applications on remote memory.

we consider as performance the total execution time, whereas for Redis and Memcached we study the end-to-end latency, the 99^{th} and the 99.9^{th} percentiles. Figure 5.6 shows the respective results and clearly reveals the existence of a correlation between certain metrics and the performance of applications. What is of great interest is that runtime metrics reveal a much higher correlation compared to the historical ones, forming our concluding remark:

R8) Predictive monitoring capability: Proactive runtime assessment of the state of the underlying system is feasible and provides useful insights both regarding the system itself, as well as the performance of deployed applications.

5.4. Adrias Design

The main goal of Adrias is to efficiently orchestrate applications arriving in a disaggregated system, by deciding between local and remote memory allocation modes. The idea behind Adrias' architecture is driven by the remarks made during the characterization process (R1-R8), and have a one-to-one relationship with the framework's instrumentation mechanisms. In short, a *Watcher* component continuously monitors and gathers performance events of the underlying system. The *Predictor* exploits Long Short-Term Memory (LSTM) models for forecasting the future state of the system and performance of deployed applications. Finally, the *Orchestrator* utilizes the predictions to

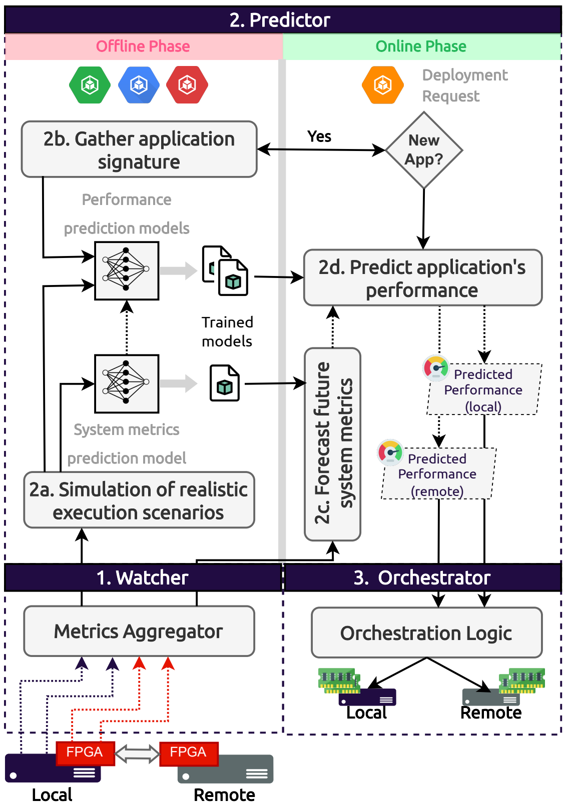

2. Predictor
Offline Phase
Online Phase
Deployment Request
New App?
Yes
2b. Gather application signature
Performance prediction models
Trained models
2d. Predict application's performance
System metrics prediction model
2c. Forecast future system metrics
Predicted Performance (local)
Predicted Performance (remote)
2a. Simulation of realistic execution scenarios
1. Watcher
3. Orchestrator
Metrics Aggregator
Orchestration Logic
Local
Remote
FPGA
FPGA
Local
Remote

Figure 5.7.. Overview of Adrias architecture

decide the memory allocation policies accordingly. Figure 5.7 shows an overview of Adrias' architecture. We note that even though Adrias uses the ThymesisFlow prototype, its design is not bound with specific memory-coherence protocols (e.g., OpenCAPI).

5.4.1. Watcher

The *Watcher* component is responsible for gathering performance events from the underlying hardware infrastructure, as well as the deployed containers on the system (❶). It monitors low-level performance events, providing insights on the data flowing through the memory hierarchy of the system. Focus is given on cache- and memory-related performance counters, as well as metrics that depict the status of the communication channel between the local and the remote memory sub-system.

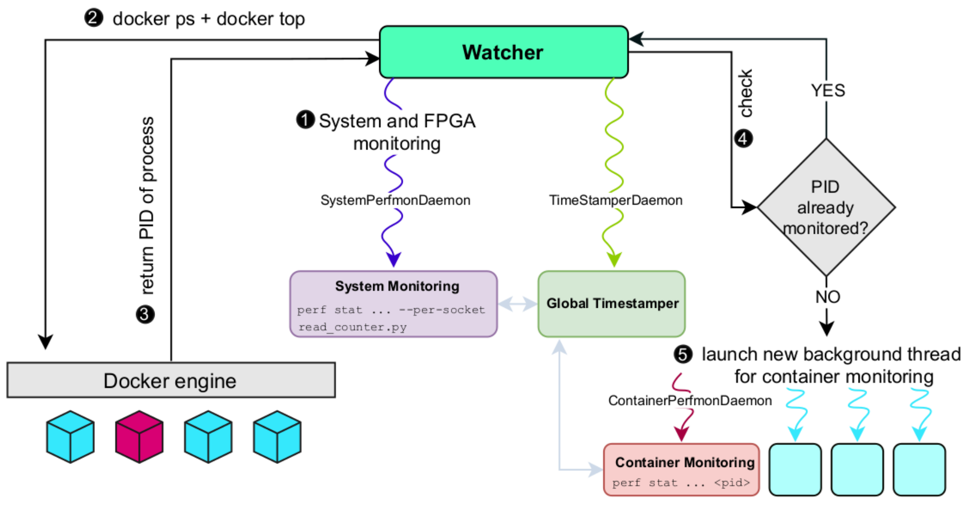


Figure 5.8.. Overview of Watcher component

Driven by our analysis in Section 5.3.2, the *Watcher* component continuously monitors the following metrics: Last-level cache misses (LLC_{mis}), Last-level cache loads (LLC_{ld}), Local DRAM memory loads (MEM_{ld}), Local DRAM memory stores (MEM_{st}), FPGAs communication's channel average latency (RMT_{lat}) and FPGAs receive (RMT_{rx}) and transmit (RMT_{tx}) throughtput. For the CPU performance events of the local system, we utilize Linux's `perf` tool, while the events related to the FPGA channel are directly provided by the ThymesisFlow framework [1]. We should note that the list of monitored events can be seamlessly extended to support more metrics. Finally, we set the monitoring interval of the the Watcher equal to 1 sec, which, as shown in [61], is a "sweet spot" between inference overhead and QoS violations increment.

5.4.2. Stacked-LSTM Predictor

The purpose of the *Predictor* (Figure 5.7) is to forecast the future state of the underlying system, as well as predicting the performance of incoming applications depending on the memory allocation mode (local vs remote). The prediction process consists of two phases, *offline* and *online*. The offline phase (design-time) involves three main activities: *i)* collection of representative system metrics' traces that correspond to "realistic" execution scenarios (❷), *ii)* generation of the dataset used for training and testing (❸) and *iii)* design, train and validation of the prediction models (❹). In the online phase (run-time), the Predictor utilizes the trained models to predict the aforementioned prediction metrics of interest. The next three subsections provide details on the two prediction phases as well as the deep learning models used.

Offline phase: Interference-Aware trace collection

The first step of the offline phase concerns the collection of interference-aware traces of low-level system metrics (by utilizing the Watcher component of Section 5.4.1). The data collected is used as input dataset for training our deep learning models that will be described later in this section. This step is vital for the overall functionality of Adrias, since gathering representative data is essential to increase accuracy of any DNN model. Adrias simulates different execution scenarios that imitate different realistic deployment schemes on local and remote memory allocation modes. Figure 5.9 shows the flowchart for generating each simulation scenario.

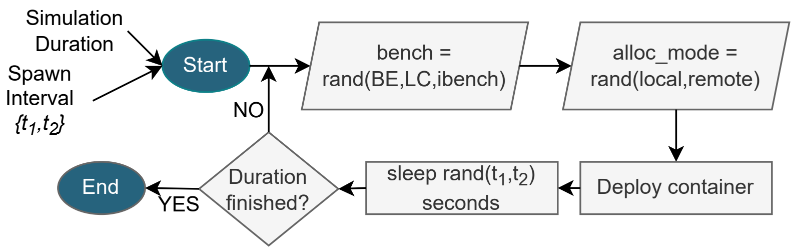


Figure 5.9.. Flowchart of interference-aware trace collection

Scenario generation: We simulate different execution scenarios by employing a random scenario generation approach. Each scenario is characterized by a *spawn interval* $\{t_1, t_2\}$, which denotes the arrival time range of consecutive application deployments on the system. For instance, a spawn interval of $\{5, 40\}$ means that each new application arrives after a random interval between 5 and 40 seconds. Within each interval we pick a random benchmark either from the examined applications, or from the iBench pool and we deploy it randomly on local or remote memory. Through iBench micro-benchmarks, we aim to replicate supplementary interference scenarios that cannot be directly generated by our examined LC and BE workloads. To capture the high dynamicity found in Cloud environments [30], we examine different arrival rates, with sets ranging from $\{5, 20\}$ up to $\{5, 60\}$, where the former imitate more congested scenarios and the latter indicate a more relaxed application arrival pattern. Figure 5.10 depicts three exemplary but representative scenarios, assuming heavy ($\{5, 20\}$), moderate ($\{5, 40\}$) and less ($\{5, 60\}$) congested application deployment distributions[3]. As shown, the specific setup exposes a wide vari-

[3]Through the random scenario generation, the maximum number of applications running simultaneously on the disaggregated testbed is 35, with Spark applications spawning 2 worker instances with 4 threads each.

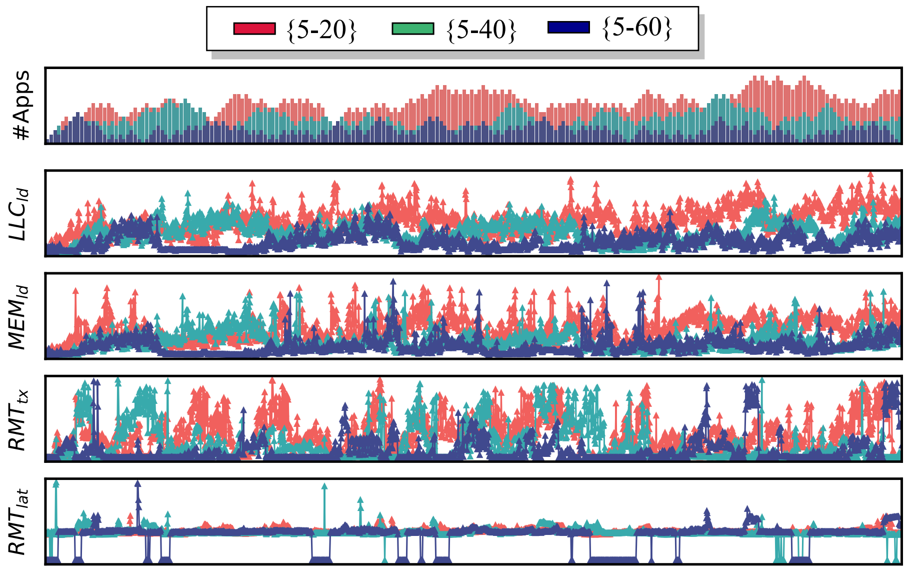


Figure 5.10.. Number of concurrent applications (top) and performance metrics over time for three representative scenarios. Adrias' data acquisition scheme captures different congestion phases, both within the same and among different scenarios.

ety of phases, both regarding the number of concurrent applications and the spectrum of the monitored metrics.

Insights from scenario execution: Overall, we have simulated 72 diverse 1-hour scenarios with different arrival rates. Figure 5.11 and Figure 5.12 show the performance distribution of our examined benchmarks over all the 72 execution scenarios. Regarding Spark benchmarks (Figure 5.11) we observe that, as expected, the use of remote memory has a substantial performance impact compared to only using local DRAM. As a result the performance distributions for the scenarios using remote memory exposes a tendency towards higher values. However, what is of great interest is that certain benchmarks (e.g., `gmm`) present overlapping performance distributions. This denotes that for certain benchmarks there might be cases where remote memory is actually better than local, or that it results in a similar application performance. Considering best-effort applications, that typically do not have strict performance requirements, we would be able to sacrifice performance to take advantage of the disaggregated memory. On the other hand, there are also benchmarks (e.g., `nweight`), which, as previously described in 5.3.3, do not take advantage of remote memory at all, due to stacking interference effects.

Regarding Redis and Memcached, we focus on the 99^{th} and 99.9^{th} percentiles, since

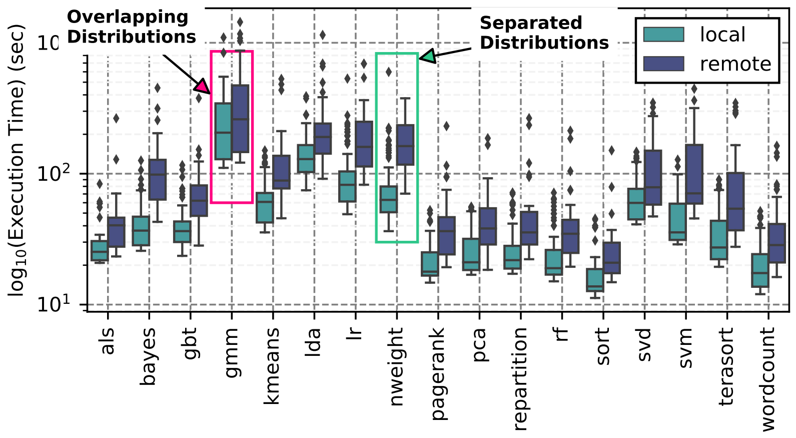


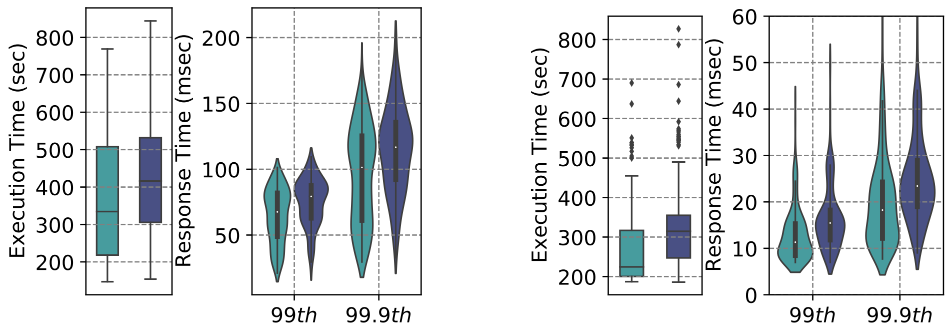
Figure 5.11.. Spark performance distribution over different execution scenarios.

typically LC applications are accompanied by similar QoS requirements. We observe that remote memory provides higher response times, however we also notice again an overlap between the two distributions. Overall, we expect disaggregated memory to be prohibitive for stricter QoS constraints, especially in the Redis case. However, when more relaxed QoS requirements are set, remote memory could be leveraged without violating any constraints.

Offline phase: Prediction models training

Adrias utilizes a stacked-model architecture by combining two prediction models one after another: a system state prediction model, that forecasts the future values of low-level system metrics, and a performance prediction model, that receives this prediction (among others) and infers the performance of an application if deployed on local or remote memory. Figure 5.13a and Figure 5.13b show an abstract view of the architecture of each model.

System State Model: The rationale behind this model originates from our observations made in Section 5.3.4 that run-time system metrics are more closely correlated to applications' performance. The model receives as input a feature vector $S =< (f_1)_{t-r}^{t}, (f_2)_{t-r}^{t}, ..., (f_7)_{t-r}^{t} >$, where $(f_i)_{t-r}^{t}, i \in \{LLC_{mis}, LLC_{ld}, ...\}$ is a sequence containing the values of metric i over a history window of length r ranging from a past timestamp $t-r$ up to current time t. As output, the model provides a vector that corresponds to the predicted mean value of system performance events, i.e. events related to hardware performance counters, over a horizon window z, i.e., $\hat{S} =< \mu((f_1)_{t}^{t+z}), \mu((f_2)_{t}^{t+z}), ..., \mu((f_7)_{t}^{t+z}) >$. Since, we focus on dis-



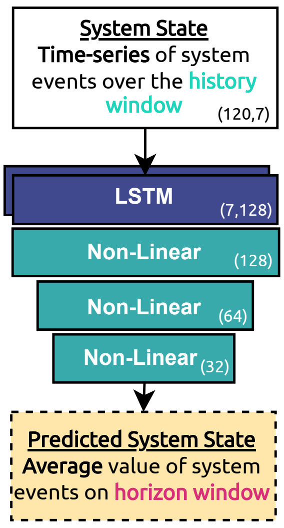
(a) Redis (b) Memcached

Figure 5.12.. Distribution of total execution time to serve 10.000 requests (left) and of 99*th* and 99.9*th* percentile of response time per request (right) for Redis and Memcached over difference execution scenarios.

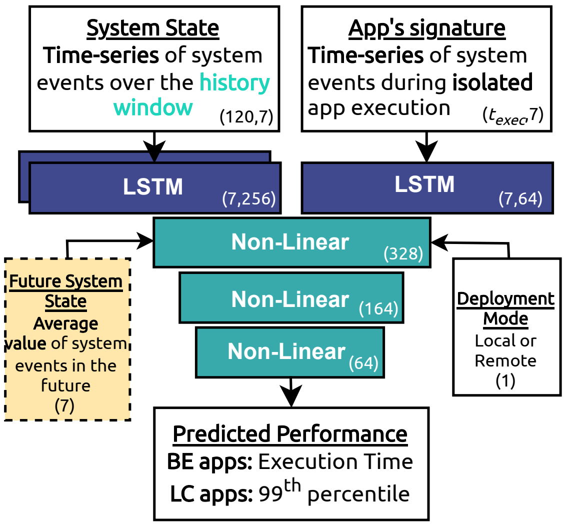
aggregated memory allocation orchestration decisions, we define system state considering the system metrics monitored by the Watcher component (5.4.1). After evaluating different values for r and z , we have determined that a value of 120 seconds provides useful insights both regarding the history and the horizon windows.

Due to the sequential nature of the input feature vector, we utilize Long Short-Term Memory (LSTM) [302] layers as the backbone of the model, which has been proven to be extremely accurate in forecasting system level hardware events in both under interference and for deep horizons [61, 313]. Specifically, the input feature vector is first processed by 2 LSTM layers that identify dependencies between the time-series data and the results are passed to a triplet of non-linear blocks, that combine fully-connected layers with Rectified Linear Unit (ReLU) activation functions, batch normalization and dropout layers to expose non-linearity and avoid overfit.

Performance Prediction Model: This model is responsible for forecasting the performance of incoming applications, if deployed on local and remote memory allocation modes. Within Adrias, we adopt a universal modeling approach, i.e., we build a unique model for all the BE and one for all the LC applications, where the former predicts the expected execution time and the latter the 99^{th} response time percentile respectively. Although prior research works typically follow a per-application modeling approach [6, 41, 164], we argue that this is not efficient (yet could be more accurate), since building a performance model per application is too time-consuming and requires simulating scenarios for each new application and maintaining a single model per workload imposes serious scalability issues.


System State
Time-series of system events over the history window
(120,7)
LSTM (7,128)
Non-Linear (128)
Non-Linear (64)
Non-Linear (32)
Predicted System State
Average value of system events on horizon window

(a) System's state model architecture



(b) Performance prediction model architecture

Figure 5.13.. DNN architecture of (a) system's state and (b) application's performance prediction models. Parentheses shows the #features and size of each layer.

Modeling and predicting the performance of applications is a non-trivial task and requires awareness regarding: *i)* the dynamics and sources of interference on the underlying system and *ii)* the inherent characteristics of the application itself and how these characteristics get affected from the current and future status of the system. To uncover this information, the performance models receive as inputs four parameters: The past and predicted system state feature vectors S and $\hat{S}$, the deployment mode (local/remote) and a feature vector $A_k =< (f_1)_0^{t_{exec}}, (f_2)_0^{t_{exec}}, ..., (f_7)_0^{t_{exec}} >$. We call this vector the *application's signature* and it is a unique identifier per application, that contains the sequences of monitored metrics during application's execution in isolation.

The time-series inputs, i.e., system state history and application's signature, are individually processed by two LSTM layers that identify important features in the sequential data. The output results are then concatenated with the deployment mode and the future system state vector $\hat{S}$ to form the hidden representation, which is then processed again by a triplet of non-linear blocks to provide the final performance prediction.

Online phase: Deployment

During deployment, the Predictor exposes a server network interface, that continuously listens for incoming prediction requests from the Orchestrator. In case of such a request, it forecasts the performance of the desired application and replies with the predicted performance for both local and remote memory allocation modes.

5.4.3. Orchestrator

The orchestrator component leverages the predicted metrics generated by the predictor to proactively assess the overall state of the system and choose the disaggregated memory policy for deployed applications accordingly. The design of the scheduling logic of the Orchestrator component is driven by two fundamentals: *i)* Typical cloud systems policies demand for QoS guarantees for latency critical workloads and best-effort execution for batch applications [41,63] and *ii)* Disaggregated memory systems hide dangerous pitfalls (characterization process of Section 5.3), i.e., disaggregated memory imposes significant performance overhead if utilized recklessly, especially when multiple applications compete over the available resources and, thus, should be leveraged wisely, targeting applications that benefit the most out of it. We tackle the first aspect directly, by introducing a straightforward, yet effective, scheduling logic for BE and LC applications, while the second one is addressed indirectly, through the automatic feature extraction by the Adrias prediction models.

When a new workload is deployed on the disaggregated system, the orchestrator first examines whether it owns any prior information regarding its *application signature*. If not, it schedules the application on the remote memory allocation mode, captures and stores its signature to be used at a later stage. Otherwise, the Orchestrator communicates with the Predictor and receives the estimated execution time (for BE) or 99^{th} percentile (for LC) for local and remote memory modes. In the case of BE applications, we utilize the following discrete function to decide between the two modes:

$$mode_{BE} = \begin{cases} local, & \text{if } \hat{t}_{local} < \beta * \hat{t}_{remote}. \\ remote, & \text{otherwise.} \end{cases} \tag{5.1}$$

where $\hat{t}$ is the predicted execution time retrieved from the Predictor and β is a slack parameter that depicts the maximum performance loss margin that we are willing to sacrifice in order to make use of the remote memory. The choice of β depends on two factors, closely related to Section 5.3.3. First, the application itself (as described later in Sec. 5.5.2), since different applications present dissimilar performance characteristics when deployed on remote memory and, second, the underlying interference, since overwhelming the remote memory requires greater performance degradation tolerance.

For LC workloads, the employed logic aims to utilize remote memory whenever and as much as possible, without violating the QoS constraints. Specifically, the memory mode is chosen based on the following statement:

$$mode_{LC} = \begin{cases} remote, & \text{if } \hat{p}^{99th}_{remote} \leq QoS. \\ local, & \text{otherwise.} \end{cases} \tag{5.2}$$

where $\hat{p}^{99th}$ is the predicted 99^{th} response time percentile retrieved from the Predictor. Particularly for LC applications, achieving QoS requirements solely through performance assessment during deployment can turn out to be infeasible, due to the unpredictability of the system's future load. In such cases, Adrias can be utilized complementary with other runtime control frameworks, e.g., [40,165], to dynamically adjust the resource requirements at runtime.

5.4.4. Implementation

We have implemented Adrias using the Python (`v3.7.0`) programming language. Each component is implemented as a separate class and is instantiated during the initialization of the framework as a daemon thread running in the background. The communication between the *orchestrator* and the *predictor* components is done using the ZeroMQ messaging library [356]. Moreover, all the models have been developed on top of the PyTorch library [142]. The source-code as well as all model's configuration parameters are publicly available under an open-source license [4].

Deployment: Adrias design allows two deployment modes, *integrated* and *segregated*. In the former, all the components of Adrias (Watcher, Predictor and Orchestrator) are deployed as a single entity on the same physical server. In this mode, the role of the orchestrator is to decide between allocating memory from the local DRAM or from a remote donor node. In the latter, a Watcher and a Predictor agent is deployed on each borrower node of the cluster, while the Orchestrator is deployed on "master" node receiving deployment requests. Whenever a new application is deployed on the cluster, the Orchestrator communicates with the Predictor agent per node and receives the predicted performance for the respective application. Based on the aggregated information from all

[4] Adrias Github Repository

Event	$\mathbf{R^2}$
LLC_{mis}	0.9969
LLC_{ld}	0.9995
MEM_{ld}	0.9641
MEM_{st}	0.9983
RMT_{lat}	0.9977
RMT_{rx}	0.9871
RMT_{tx}	0.9876
Avg.	0.9932

Table 5.3.. R^2 score per performance event

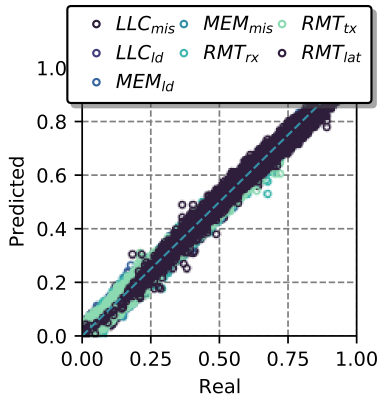


Figure 5.14.. Performance events predictions' regression residuals

the nodes, it decides *i)* to which node should the application be deployed and *ii)* whether it should be deployed on local or remote memory. For the purposes of this study and due to the limited hardware infrastructure available, we have used the integrated Adrias deployment mode.

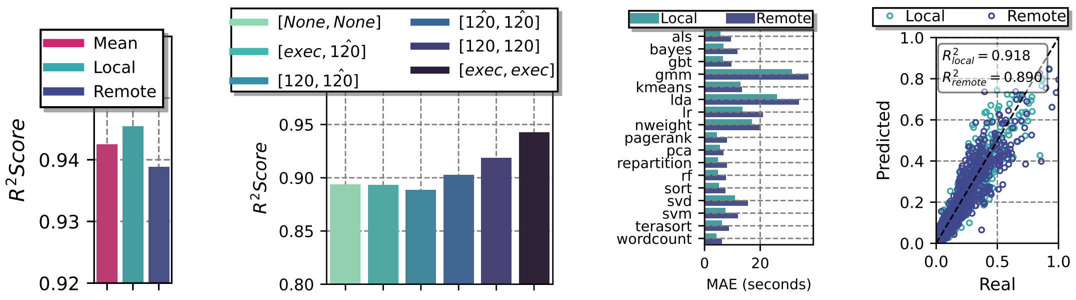
5.5. Evaluation

We evaluate Adrias' effectiveness in terms of *i)* prediction accuracy w.r.t future system state and applications' performance, *ii)* utilization efficiency of the disaggregated memory and *iii)* resources overhead.

5.5.1. Accuracy Evaluation

System state prediction model

We partition the datasets produced during simulation (Section 5.4.2) in two subsets of 60% (training set) and 40% (test set) of the samples respectively and we examine the ability of the system state prediction model to accurately predict the mean value of monitored metrics over the horizon window. Table 5.3 shows the respective results per monitored metric, by evaluating the coefficient of determination - R^2 for a horizon window equal to 120 seconds. Adrias achieves pretty high accuracy, ranging from 0.964



(a) Overall prediction accuracy

(b) Impact of real vs. predicted system state on perf. accuracy

(c) MAE per Spark benchmark

(d) Prediction residuals

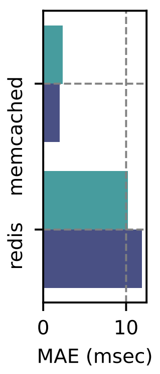
Figure 5.15.. Evaluation of performance prediction model (execution latency) for Best-Effort (BE) applications

up to 0.999 R^2 score for all the examined metrics and with an average of 0.993 R^2 overall, illustrating the ability of Adrias to proactively assess the tendency of the metrics of the system in the future. To further assess the robustness of our model, we also present a residual analysis of the predictions, as there exist cases where a high R^2 score could be counter-intuitive [357]. Figure 5.14 presents the results, where the x axis shows the actual values of the metrics and the y axis the predicted ones. This plot verifies the strong prediction capabilities regarding the system's state, since the majority of the points lie on the 45^o residual line.

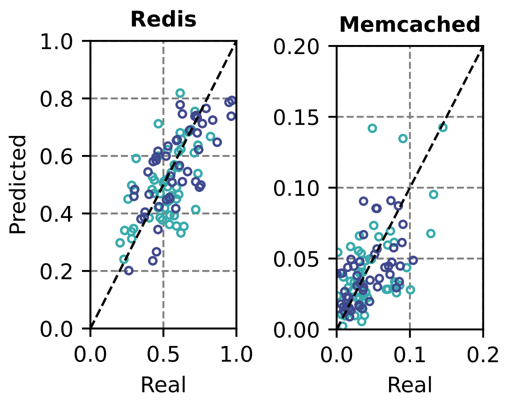
Application performance prediction models

Similar to Section 5.5.1, we partition the dataset in two subsets of 60% (training set) and 40% (test set). As a first step, we train and test the performance models for BE applications, using as the future system state ($\hat{S}$) the actual monitored metrics, gathered during the trace collection process. Figure 5.15a depicts the respective results, showing that Adrias is able achieve a $0.942R^2$ score on average, with a slightly higher accuracy for predicting the execution time on local mode ($R^2 = 0.945$) compared to the remote ($R^2 = 0.939$).

Impact of stacked models to overall accuracy: An issue with the proposed stacked predictor approach is that the actual monitored metrics will not be available during a realistic inference step. Thus, an open question that rises is the following: *Should we train the performance models using as "future system state" the actual system metrics,*



(a) MAE

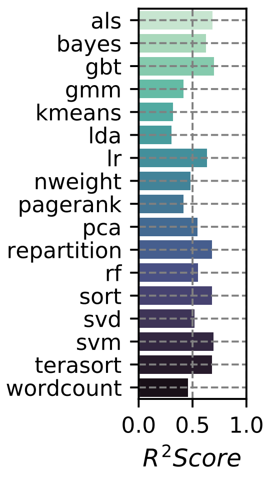


(b) Prediction residuals for LC applications

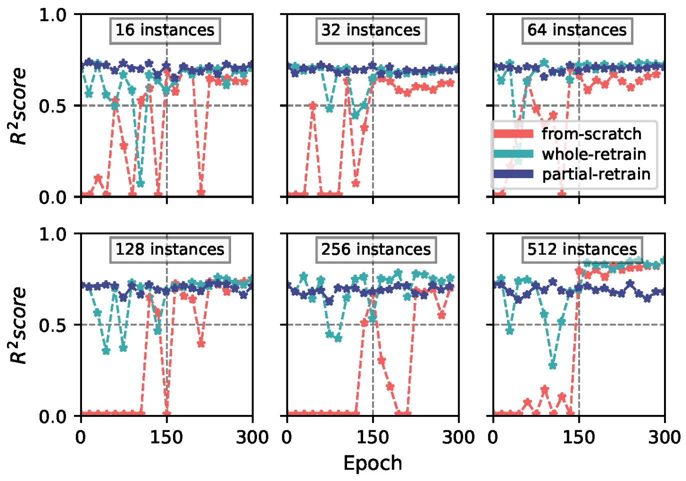
Figure 5.16.. Evaluation of prediction model (99^{th} percentile) for LC apps

or train using propagated predictions from the system state model? To answer this question, we examine the prediction accuracy for different input vectors ($\hat{S}$) during training and testing. Figure 5.15b shows the results, where the pair $[x_1,x_2]$ maps to (train,test) and denotes the type of vector $\hat{S}$ used in each phase. Specifically, $None$ implies that $\hat{S}$ was not fed to the model, while 120, $\hat{120}$ and $exec$ indicate that $\hat{S}$ is calculated from the actual metrics or propagated from the system state model for a window of 120 seconds or for the full duration of the application respectively. The pairs $(120, 120)$ and $(exec, exec)$ give the theoretical maximum accuracy of the model, which, however, is not achievable in practice, since we cannot know a priori the exact values of the monitored metrics in the future. The plot reveals that overall best approach is to feed to the performance model the predicted vector $\hat{S}$ also in the training phase ($[\hat{120}, \hat{120}]$). As we see, even though the system state model provided extremely accurate predictions ($0.99R^2$), we still experience an accuracy drop of approximately 3% compared to the theoretical maximum presented above ($[exec, exec]$). Moreover, we also observe that leveraging the predicted future system yields a 2% better accuracy compared to only considering historical data ($[None, None]$), thus, verifying the advantage of predictive over conventional monitoring.

Runtime accuracy: Last, by employing the $[\hat{120}, \hat{120}]$ approach, we also show the Mean Absolute Error for BE (Fig. 5.15c) and LC (Fig. 5.16a) applications, as well as the respective regression residuals (Figures 5.15d and 5.16b), scaled to $[0, 1]$ using min-max normalization. Comparing the mean absolute error with the median performance presented in Fig. 5.11, the employed DNN models are able to provide accurate performance predictions for both BE and LC applications. Even in cases where we observe high MAEs (e.g., `gmm,lda`), we see that these errors correspond to approximately 10% variation compared



(a) Application-granular leave-one-out accuracy



(b) Model (re)training accuracy over time (epochs) for various number of samples for `gbt` benchmark

Figure 5.17.. Performance prediction accuracy for unseen applications (a) and different model retraining approaches (b).

to the median values of their performance distribution. Overall, we are able to achieve an R^2 score equal to 0.905 for BE and 0.874 for LC respectively.

Generalization on unseen applications: Last, we test Adrias' universal modeling approach ability, by evaluating accuracy using an application-granular leave-one-out validation. Figure5.17a shows the R^2 score per benchmark, when excluded during training phase. The model is able to generalize adequately for certain benchmarks (e.g., `gbt`) whereas it fails for others (e.g., `lda`) yielding 0.72 and 0.30 R^2 scores respectively. This suggests that a continuous collection of representative *application signatures* and retraining is crucial for unknown applications. We explore the effectiveness of three retraining approaches to improve the accuracy on newly collected data of unseen applications, i.e., *i) from-scratch:* train the whole model from the beginning, *ii) whole-retrain:* retrain all the layers of the model using data from the unseen application and *iii) partial-retrain:* update only the weights of the model that correspond to the application's signature. To avoid bias on the new data during retraining, we set a one-order of magnitude lower learning rate and we feed batches of mixed samples from seen and unseen applications. Figure5.17b shows the accuracy achieved and time needed (0.8sec per epoch on a V100 GPU) for different number of training samples for `gbt` application. For lower number of available samples (16-64), whole-retrain results to better accuracy compared to a ground-up training, whereas for higher ones (128-512) it achieves comparable results with less time (epochs) needed. Partial retraining does not improve accuracy over the unseen application, showing the close interrelationship between the application's characteristics (signature) with the dynamics of the system (system state).

5.5.2. Orchestration Evaluation

In this section, we examine the efficiency of Adrias' orchestration mechanism described in Section 5.4.3. To schedule applications, Adrias utilizes the pretrained predictive models presented and evaluated in the previous section. In order to orthogonalize training and evaluation process, we avoid duplication between the traces collected and used to train our models and the ones occurring during evaluation, thus we generate a set of additional execution scenarios (as described in Section 5.4.2) with arrival rates ranging from {5,20} up to {5,60}, which are utilized to evaluate the orchestration logic. For BE applications, we evaluate the ability of Adrias to utilize the remote memory as much as possible, without violating the performance threshold introduced through the slack parameter β. For LC applications, we explore Adrias' capability to successfully predict and schedule Redis and Memcached on the remote memory (when possible), without violating a pre-established Quality-of-Service (QoS) constraint. We compare Adrias with three other scheduling schemes, i.e., *Random* and *Round-Robin*, where the memory allocation mode is chosen randomly and in turn between local and remote respectively and *All-Local*, where all applications are allocating memory from the local DRAM.

Best-Effort applications

Figure5.18 (top) shows the execution time distribution of all the examined BE benchmarks and Figure5.18 (bottom) the number of times each application got scheduled on the local and remote memory when using the *Random*, *Round-Robin* and *All-Local* schedulers, as well as Adrias under different β slack parameter values.

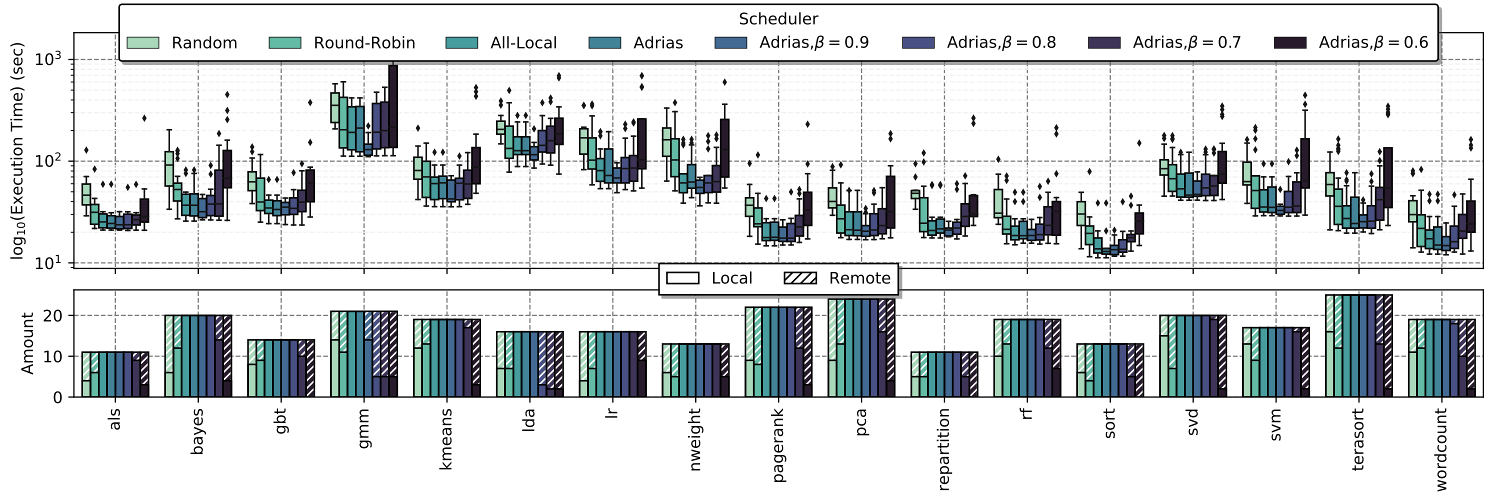


Figure 5.18.. Execution latency distribution of BE applications (top) and number of times that the application was deployed on local and remote memory (bottom) for different scheduling logics.

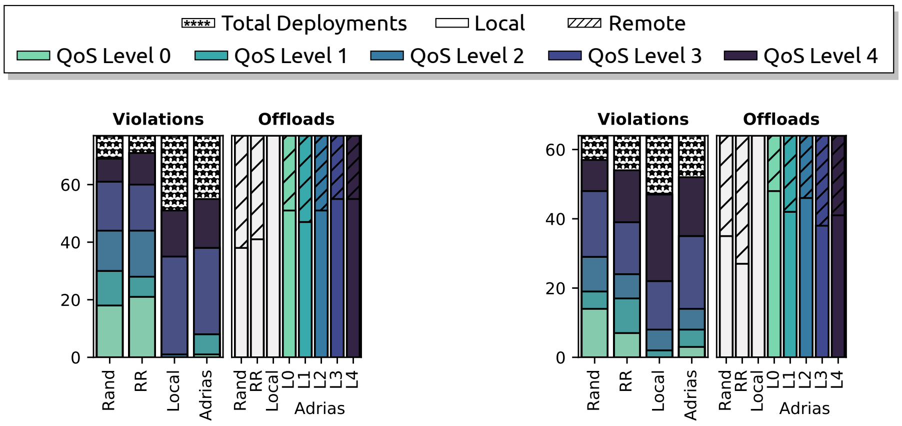
For the majority of the workloads, the *Random* and *Round-Robin* schedulers provide the worst performance distributions, confirming the need for intelligent orchestration mechanisms. For higher slack parameter values (β = 1 and β = 0.9), Adrias provides identical scheduling decisions with the *All-Local* scheduling logic, due to the explicit performance deterioration of the remote memory combined with the implicit accuracy errors of the prediction models, which render the orchestrator incapable of utilizing the remote memory. For β values equal to 0.8 and 0.7 Adrias achieves to effectively utilize the remote memory, managing to offload approximately 10% and 35% of deployed applications with an average drop of 0.5% and 15% in the median performance over all apliccations respectively.

While the values 0.8 and 0.7 for β would imply an equivalent degradation on the performance of applications, we observe that this is not the case, which is attributed to the accuracy error of the performance prediction model.

Moreover, Adrias' scheduler favors offloading certain applications to the remote memory (e.g., `gmm`, `lda`), which as was shown in Sec. 5.4.2 present overlapping performance distributions between local and remote modes, whereas it avoids offloading the ones presenting "non-overlapping" distributions (e.g., `nweight`). This observation verifies that Adrias is able to properly model and expose the inherent characteristics of the examined applications. Finally, for lower slack values (i.e., β=0.6) Adrias offloads the majority of deployed applications to remote memory, which, however, induces significant performance degradation.

Latency-Critical applications

We evaluate the ability of Adrias to efficiently allocate LC applications on the remote memory without violating QoS constraints. We identify and examine QoS constraints of various strictness. Based on Figure5.12, we define five different QoS (tail latency) levels, i.e. 0 up to 4 per LC application, that correspond to the 87.5^{th}, 75^{th}, 50^{th}, 25^{th}, 12.5^{th} distribution percentiles, where Level 0 denotes the most relaxed and Level 4 the strictest QoS constraints respectively. Figure5.19 depicts total number of violations (left) and offloads (right) for Redis (a) and Memcached (b) applications for the respective QoS levels and all examined schedulers. Similar to the case of BE applications, we observe that *Random* and *Round-Robin* schedulers provide the worst possible results, since they introduce the highest numbers of QoS violations both for Redis and Memcached, whereas *All-Local* outperforms the others by introducing almost no violations for looser QoS constraints and minimum number for stricter ones. Regarding lower QoS levels (0-2), Adrias achieves almost identical results to the *All-Local* approach, by almost eliminating the



(a) Redis (b) Memcached

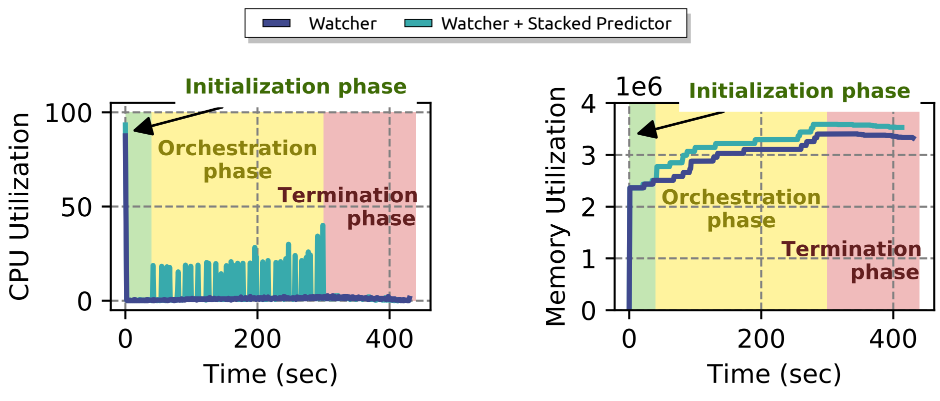
Figure 5.19.. Number of QoS Violations (left) and number of times the application was deployed on local and remote memory (right) for Redis (a) and Memcached (b) benchmarks.

majority of QoS violations, while offloading almost 1/3 of the LC applications to the remote memory. For stricter QoS requirements, Adrias provides comparable results to the *All-Local* scheduler by introducing an average of 5% and 20% more QoS violations for Redis and Memcached respectively.

Adrias' impact on data traffic: Last, we quantify the amount of transmitted data over the FPGAs' network interconnection. Among all the examined scenarios, Adrias reduces the amount of transmitted data by 45% ($\beta = 0.8$) and 23% ($\beta = 0.7$) on average compared to Random and Round-Robin schedulers respectively. We note that in cases where Adrias offloads similar number of applications with the other schedulers, it generates up to 55% less traffic on the channel, revealing its tendency to favor less memory-intensive applications to be allocated on the disaggregated memory.

5.5.3. Overhead Analysis

Finally, we evaluate Adrias in terms of its computational requirements and imposed overhead when deployed on a borrower node. Specifically, we measure the CPU and RAM utilization for the Watcher and the Predictor components using Linux's `top` command for a generated mixed-workload scenario with duration equal to 300 seconds. In addition, based on prior research [313] we set the number of PyTorch's OpenMP threads equal to 1. Figure 5.20 shows the respective overheads, where the Initialization phase refers to



(a) CPU (%) (b) Memory (Bytes)

Figure 5.20.. Overhead of Adrias in terms of CPU and Memory

the grace period of 120 seconds, required to gather the history window used as input to the predictions models; the Orchestration phase refers to the real execution part, where Adrias utilizes the Predictor's inference engine to evaluate the performance of incoming applications on the system; and the Termination phase refers to the waiting period until all benchmarks have finished their execution. During initialization, we notice a spike both in CPU and memory due to the instantiation of Adrias' components (as described in Section 5.4.4). In terms of memory, Adrias imposes minimal overhead on the system, with an average of 3MB RAM utilization. Moreover, this overhead is accumulated during the orchestration phase, due to the fact that Adrias monitors and stores information regarding all the running containers on the system. Last, we observe that Adrias has almost no impact in terms of CPU utilization, with occasional CPU usage spikes of approximately 15% [5]. For the shake of completeness, we also report the average inference time of the Predictor, which requires 0.63 seconds on average, for the system state and performance predictions.

5.6. Further Discussion Points

Why Deep Learning? Modern cloud data centers suffer from extensive and non-deterministic performance variability due to interference, workload diversity and HW heterogeneity [30], thus mechanistic or model predictive control approaches form highly expensive solutions due to the extensive simulations required to capture all the possible deployment scenarios [358]. To this end, ML-centric cloud platforms are attracting a lot

[5]`top` expresses CPU utilization as a percentage of total CPU time (12800% in our system)

of attention [28, 55, 359]. The rationale behind the employment of DL techniques is the automatic pattern discovery of neural networks; notably such patterns, able to discriminate performance indicators, could not be effectively set by typical human modeling or would require extensive scenario analysis. In particular, LSTMs have been proven to be extremely efficient on interpreting temporal patterns, i.e., interpreting system monitor time-series to actual performance metrics [61, 313].

Ability of Adrias to unveil human-driven remarks. During evaluation (Figure5.18) we noticed that Adrias favors offloading certain applications to the remote memory (e.g., `gmm`, `lda`). While these applications present overlapping performance distributions between local and remote modes (Sec. 5.4.2), Adrias avoids offloading "non-overlapping" ones (e.g., `nweight`), which also suffer from stacked interference effects (R7). This verifies that Adrias properly models the inherent characteristics of the examined applications.

Adrias & HW heterogeneity. Adrias assumes no prior knowledge on the HW infrastructure, as any performance variability due to heterogeneity will directly affect the monitored metrics. For example, in case a system avails of both remote DRAM and NVMe, these would be considered by Adrias as two different memory tiers, with different latency characteristics. There is no requirement for Adrias to be aware of the actual medium backing each tier.

Adrias scalability. Due to the inherent HW limitations of the ThymesisFlow prototype, Adrias was evaluated on a single-node cluster. However, by design, Adrias is able to scale on multiple nodes, where the monitoring (Watcher) and performance prediction (Predictor) components are distributed across the nodes of the cluster. The orchestration logic could be centralized (e.g., be integrated directly in the control plane of Kubernetes [118]), however, it should be adjusted in a straightforward manner to account for cluster-level efficiency in case of iso-QoS predictions between different nodes.

5.7. Conclusion

Hardware disaggregation is the next big step for efficient and fine-grained management of cloud infrastructures. We presented Adrias, a monitoring and orchestration framework for memory disaggregated cloud systems. We performed an extensive, interference-aware characterization process for a set of well-known cloud applications and highlighted the hidden pitfalls on a real memory disaggregated testbed. Driven by this analysis, we designed Adrias, a framework leveraging deep learning techniques to decide the memory mode of deployed applications and showed that it can efficiently utilize remote memory with minimal performance overheads.

Chapter 6.

Deep-Learning Driven Autotuning for Taming High-Dimensionality of Spark Deployments

The unprecedented amount of data uploaded and manipulated in the Cloud has pinpointed the need for more efficient data processing. In-Memory Computing (IMC) frameworks (e.g., Spark) offer enhanced efficiency for large-scale data analytics, however, they also provide a plethora of configuration parameters that affect the resource consumption and performance of applications deployed on top of them. Manually configuring all these parameters can be very time-consuming, due to i) the high-dimensional configuration space, ii) the complex inter-relationship between different parameters, iii) the diverse nature of workloads and iv) the inherent data heterogeneity.
In this chapter, we present Sparkle, an end-to-end framework for performance modeling and autotuning of Spark applications based on deep learning techniques. Sparkle leverages a modular DNN architecture that expands to the entire Spark parameter configuration space, while also provides a universal performance modeling approach, thus, completely eliminating the need for human or statistical reasoning in the loop. By employing a genetic optimization process, Sparkle quickly traverses the design space and identifies highly optimized Spark configurations. Through an extensive experimentation campaign on the HiBench benchmark suite, we show that Sparkle's DNN delivers an average prediction accuracy of 93%, with very high generalization capabilities, i.e., $\approx 80\%$ accuracy for unseen workloads, dataset sizes and configurations, clearly outperforming state-of-art. Regarding the end-to-end optimization, we show that Sparkle is able to explore very efficiently Spark's high-dimensional parameter space, delivering new dominant Spark configurations, which correspond to 65% Pareto coverage w.r.t its Spark native optimization counterpart.

6.1. Introduction

Nowadays, the unprecedented amount of data produced every day is truly overwhelming. In fact, according to a recent research [360], over than 2.5 quintillion bytes of data are produced every day. To process and make value out of this information, novel frameworks [92, 141, 245] have emerged that enable the parallel processing of huge data volumes in a distributed manner. Spark [245] is today the de-facto processing engine for large scale data analytics. Spark proposed a novel distributed model, called resilient distributed dataset (RDD), that is stored in memory while being computed upon, thus eliminating expensive intermediate disk writes, as in other data analytics frameworks, e.g., Hadoop [141]. Although initially offered as a cluster-based solution, nowadays Spark has been extended to support modern cloud- and/or cloud-native deployment models [361].

Even though Spark's architecture provides inherent performance efficiency, it also offers a wide variety of configuration parameters which can be tuned to alter several aspects of its runtime engine, for further increasing performance. In fact, the latest Spark releases expose more than 150 configuration parameters [362]. Although most of them are well documented, official Spark performance tuning guidelines refer only to a very primitive subset of Spark parameters [247], thus leaving the burden of analyzing their impact and tuning the final deployment, solely on the developer. Analyzing and exploring the impact of various configurations on the performance of Spark applications and also examining the inter-correlation between different parameters is a painful procedure for developers, due to *i)* the high-dimensional configuration space, *ii)* the huge, cumulative, running time required and *iii)* the time required to comprehend in depth the purpose of each parameter. On top of that, the inter-relationship between different parameters further introduces an extra level of complexity. As a typical example, while the amount of memory per executor usually improves performance, it also leads to increased garbage collection times [363]. Consequently, application developers tend to tune empirically only the most obvious performance-related parameters, such as the number of executors or the RAM per executor, while, also, neglecting the performance variability for different dataset sizes. Nevertheless, such an approach not only does not provide optimal results, but also augments the cost dramatically, due to the unnecessary usage of memory and CPU resources and long duration of working processes.

However, Spark applications can be bottle-necked by any resource in the cluster with CPU, memory and network bandwidth being the most common congestion points [249]. As shown later in the manuscript (sec. 6.3.1), proper tuning of Spark configuration can provide tremendous performance gains reaching up to ×7 faster execution for typical in-memory applications. While Spark itself offers a tuning guideline [247], it mainly focuses on data serialization and memory tuning and, most importantly, it mostly relies on an application- and datasize-specific trial and error approach, thus being far away from delivering optimal configuration results. Moreover, the high-dimensional nature of the parameter space exposed from Spark engine challenges both the modelling and optimization strategies. As mentioned in [364], in high dimensional spaces, i.e. #parameters > 100, even scalable search strategies break down, requiring usually to either adopt local optimization strategies [364] or retain aggressive parameter pruning during modelling [6, 7, 365]. From the above discussion, it is evident that there is a need of automated tuning frameworks, to ease the exploration of this high-dimensional search space.

As recently discussed in [269], major principles such as *versioning adaptivity*, *data resiliency*, *workload heterogeneity* and *fast convergence* should be considered as first class design concerns for Spark auto-tuning. These standards tackle the provision of accurate performance predictions for different workloads and volume sizes as well as enable near-optimal configuration in an instant manner, to amortize optimization costs through the resulting savings. Even though prior research has proposed tuning frameworks [4,6,7,365], which automatically configure Spark parameters to optimize execution time, they typically neglect one or more of the aforementioned principles, by building workload-aware, dataset-driven or even version-specific performance models.

To address the aforementioned challenges, we propose *Sparkle*, an end-to-end auto-tuning framework for high-dimensional Spark in-memory analytics. *Sparkle* relies on deep learning techniques and low-level performance monitoring time-series to model performance of Spark deployments in an application agnostic manner. By employing genetic optimization, the framework efficiently traverses the search space online and recommends optimized deployment configurations. *Sparkle* advances over state-of-the-art approaches by providing a universal performance modeling approach, rather than application- and/or dataset-specific ones, whereas it also extends over the complete configuration space, thus, completely eliminating the need for human-in-the-loop or statistical approaches to identify the importance per parameter. The contributions of this work are:

- We provide an in-depth analysis on the impact of parameter tuning on the performance of Spark applications. We showcase how the importance of each parameter is affected w.r.t. *i)* the deployed application and *ii)* its input dataset size.

- We propose a hybrid DNN architecture that leverages convolution, Long Short-Term Memory (LSTM) and non-linear blocks to exploit both application- and configuration parameter-related characteristics, thus, being able to provide accurate performance predictions of Spark deployments using a single, universal model.

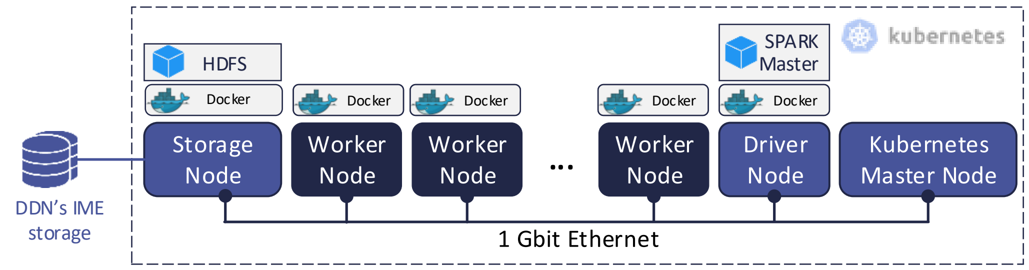


Figure 6.1.. Target HW & SW infrastructure

- Based on the above, we design *Sparkle*, an online autotuning framework for Spark analytics. *Sparkle* seamlessly adapts to changes in the Spark engine, by considering the entire Spark configuration space, hence, not relying to statistical or human reasoning for identifying the importance of possible new parameters introduced in the loop. Moreover, through its universal modeling approach, it manages to extrapolate to unseen workloads and dataset sizes, tackling the inherent workload heterogeneity found in modern deployments.

We evaluate *Sparkle* against a set of 13 in-memory applications from the HiBench [2] suite. Experimental results show that *Sparkle* provides an average accuracy of 93% for determining whether a SPARK configuration will provide speedup compared to default execution or not, and an average MAPE of 7% for estimating the actual value of the speedup. Moreover, through its universal modeling approach, *Sparkle* is able to provide accurate predictions for unseen applications, dataset sizes and configurations, with an average of $\approx$ 80% classification accuracy and $\approx$ 10% prediction error. Overall, *Sparkle* improves the performance and costs expenses of the examined applications by up to a factor of $\times$6.8 and by up to 54% respectively, compared to the naive, default execution.

6.2. Examined Testbed & Spark Background

6.2.1. Experimental Methodology

This section describes our experimental infrastructure and the benchmarks used to evaluate *Sparkle*. Figure 6.1 shows an overview of our hardware and software technology stack.

Hardware cluster: All of our experiments have been conducted on a 13-node homogeneous cluster of high end machines. Table 4.1 shows the respective HW specification, where 10 servers are exclusively reserved for deploying Spark executors (worker nodes) and the rest are used for hosting necessary software stack components. The servers are interconnected over a 1 Gbit/s ethernet communication channel. In addition, our cluster has access to a DataDirect Network's (DDN) Infinite Memory Engine (IME) [366] storage system, that enables fast buffering of large data chunks.

Software layer: We consider Spark deployments managed by a Kubernetes [175] resource manager, where the executors are dynamically deployed as Docker containers [286] on top of the worker nodes. All of our nodes have Docker (`v19.03.2`) installed and are registered in a Kubernetes (`v1.19.7`) cluster, where, one node acts as the master of our Kubernetes cluster. The Spark driver is deployed as an isolated container in one of the nodes of the cluster (Driver Node). Finally, we employ Hadoop HDFS [141] (`v2.7`) as our distributed file system with 1 namenode and 1 datanode, also deployed as containers on an isolated node (Storage Node). The remaining 10 HW nodes are exclusively reserved for deploying Spark executors (worker nodes).

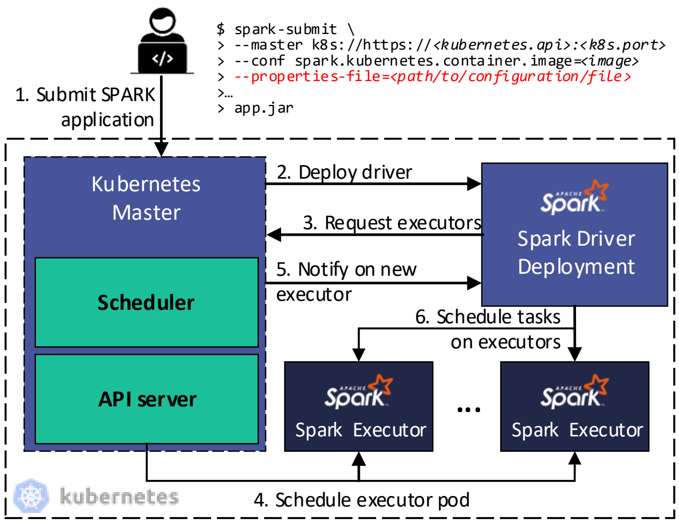
Examined SPARK applications: We study 13 SPARK applications derived from the HiBench benchmark suite [2], which is widely used to evaluate Spark, listed in Table 4.3. These applications form representative examples out of four different workload categories, i.e., graph analytics, micro-operations, machine learning and web searching. Moreover, we examine a diverse set of input dataset sizes per application.

6.2.2. Importance of Spark parameters

Next, we provide an overview of Spark [245], showing Spark works with a Kubernetes resource manager and providing some further insights on the significance of Spark parameters over diverse workloads and different dataset sizes.

Spark over Kubernetes Architecture:

Figure 6.2 shows an overview of Spark's over Kubernetes architecture. Spark applications are deployed directly over Kubernetes by specifying the respective API endpoint using the `-master` parameter. Thus, when users submit a Spark application (1), they communicate directly with Kubernetes, through the API server. Once the API server receives the request to deploy a new Spark application, it automatically deploys a Spark



```
$ spark-submit \
> --master k8s://https://<kubernetes.api>:<k8s.port>
> --conf spark.kubernetes.container.image=<image>
> --properties-file=<path/to/configuration/file>
>…
> app.jar
```

Figure 6.2.. Spark over Kubernetes deployment flow

Driver container on the cluster (2), that is responsible for requesting Spark executors from the API server (3). After the Driver's request for new executors, Kubernetes' API server automatically deploys the respective number of executor containers on the cluster (4), which can communicate directly with the Spark Driver, and notifies back once the executors are up and running (5). Finally, the Driver deployment assigns tasks on the executor containers and performs Directed Acyclic Graph (DAG) scheduling on them.

Tuning over the entire parameter space

Spark v3 provides over 150 parameters that control both performance and management related (e.g., `spark.app.name`) settings and can be configured separately per application. This process guarantees that the engine has a flawless performance and also prevents bottle-necking of resources within Spark. As expected, not all parameters have the same impact on performance, however, determining the most important ones is not a trivial task. While previous approaches either choose important parameters empirically [6,149] or through statistical reasoning techniques [7], we argue that such approaches are not efficient and one has to consider all the parameters to achieve an optimal performance. To showcase our statement, we assess the importance of Spark parameters through a univariate statistical test, namely the ANOVA F-Test [367]. Specifically, we execute our examined applications for different input configurations and different dataset sizes and evaluate the importance per Spark parameter over all the executed scenarios.

The boxplot of Fig. 6.3 shows the respective results, where the y-axis shows the score of the respective parameter derived from the F-test and x-axis shows all our examined Spark parameters sorted in a descending order based on the median value of the score. As expected, it is shown that `executor-instances` forms the most important feature by far, which is also the reason that previous efforts [146,148] focus exclusively on optimizing this parameter. Another interesting fact is that `scheduler.minRegisteredResourcesRatio` forms the third most important parameters in our Kubernetes-based cluster, however, prior scientific efforts have not considered these parameters in their examined ones. Overall, we observe a varying behavior regarding the significance of the first 33 parameters, while for the rest the impact appears to be lower.

Significance over applications: We broaden our analysis by exploring the performance impact of each parameter in a per-application manner. The question posed is the following: *Are various applications affected differently from the same parameters?*. The top heatmap of Fig. 6.3 shows the F-test score per application, where brighter cells depict a higher yielded score. As observed, certain parameters (e.g., `executor.instances`) are equally important across all benchmarks. However, there are also cases where performance sensitivity is not alike among all workloads (e.g., `executor.cores`). This is typical even for parameters that appear as less significant overall, such as `default.parallelism` and `executor.pymemory`, which affect only a specific subset of the examined applications (MC4, ML2, PR and NW, ML3, ML4, ML5 respectively).

Significance over dataset size: Last, we perform the same analysis in a per-datasize manner, where the respective results are shown in the bottom heatmap of Fig. 6.3. As before, we notice analogous motifs regarding performance sensitivity. In addition, we observe that less important parameters have very limited impact when examining different sizes, although they do have impact in different applications, as shown in the top heatmap. The first 23 most significant parameters however seem to impact in non trivial patterns applications performance under scaled datasets sizes, which also positively cross-validates the analysis performed in prior dataset-aware research works [6, 7].

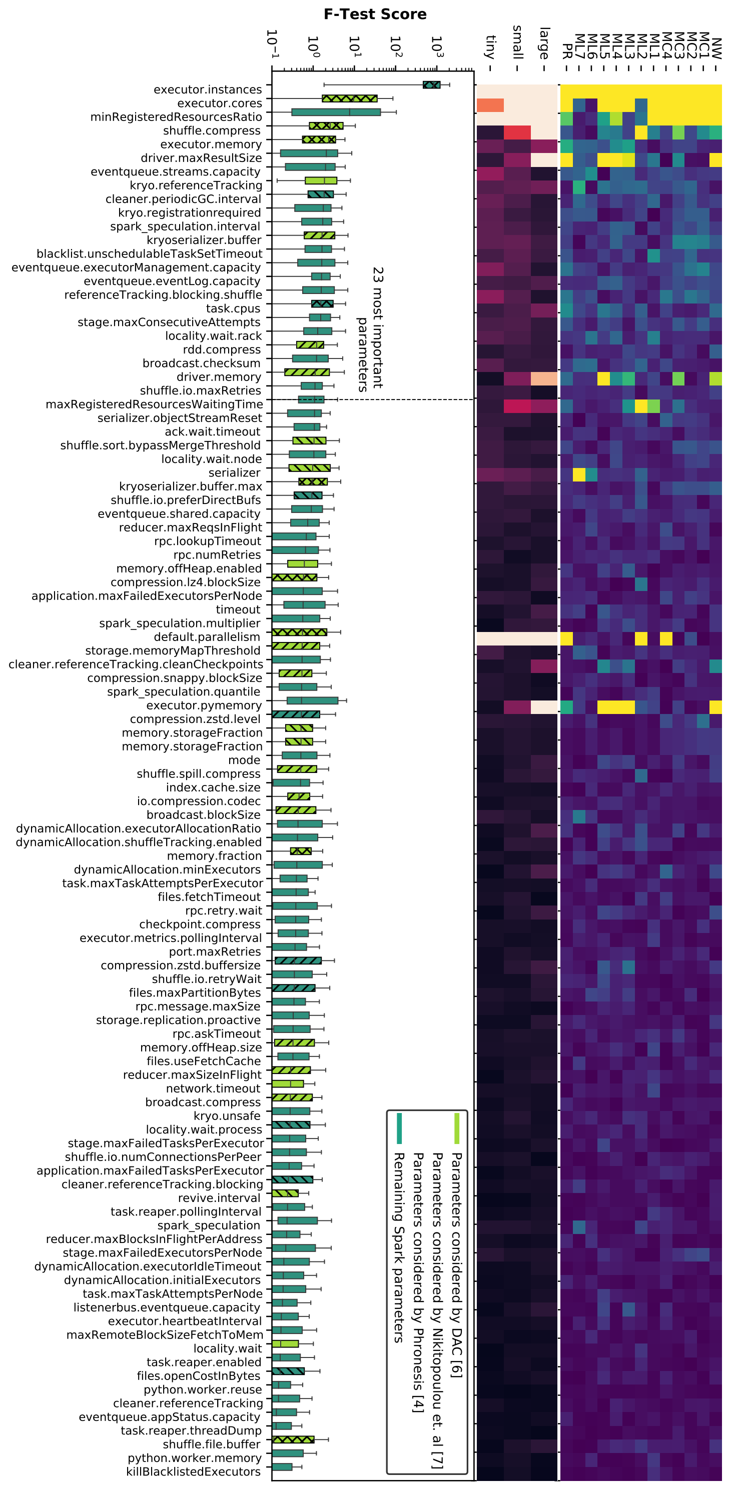


Figure 6.3.: Importance of Spark parameters based on the score derived from ANOVA F-Test. Top: Application-specific importance. Middle: Dataset size-specific importance. Bottom: Overall importance and parameters considered in prior works.

Remarks From the above discussion, we conclude that “cherry picking" Spark parameters should not be the de-facto approach and there is no “golden rule" to determine the ideal configuration, since this process depends, among others, on the nature of the deployed application and its input dataset. To further strengthen our statement, we examine the set of parameters considered in prior Spark autotuners, chosen either empirically [4,6] or through statistical reasoning [7]. The bottom of Fig. 6.3 presents the respective parameter sets, through different decorations inside the boxplots. We see that the parameters chosen by each approach differ considerably from each other. What is of great interest is that only 13 out of our 30 most important parameters are considered in at least one prior work, with the the most important one (i.e., `executor.instances`) it neglected in [6]. Moreover, there are four distinct parameters which are considered solely in [6], [7] and [4] respectively, while, overall, there are only 7 common ones between all the three tuning frameworks. We conclude that empirical selection of parameters can be almost entirely subjective, while also, statistical approaches lead to completely different assumptions depending on the circumstances.

Employing universal modeling

Spark is a general-purpose data analytics framework that enables the development and deployment of new (potentially unseen) applications on different clusters. Previous state-of-the-art Spark autotuners employ performance modeling techniques at the application [6,146,148] or cluster [7] level. While these techniques have shown to improve the performance of individual applications, they are inadequate for generalization to new applications. Thus, conventional autotuning frameworks, when optimizing for a new application, require the entire tuning lifecycle to be repeated, which includes collecting performance data for different Spark configurations, training a new machine learning model from scratch, and tuning its hyperparameters to increase its prediction accuracy. With an increasing number of new applications to tune, the costs associated with this process become prohibitively high due to new data collection, particularly for applications with large input datasets, which can take several days to execute [368].

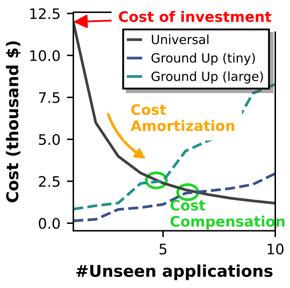


Figure 6.4.. Cost trend for universal and per-application modeling approaches

To address this challenge, we adopt a universal modeling approach that mitigates the overall costs of tuning new applications by distributing the fixed and nonelastic expenses of data collection and training across different, previously unseen applications. Indeed, an organization may deploy a variety of Spark applications each with unique configurations and datasets. Figure 6.4 illustrates this concept, presenting the cost of collecting data and training an ML model for a universal autotuner (without loss of generality, we consider training with 20 different application/dataset pairs) versus a ground-up ML model. Considering AWS's billing model, the y-axis represents the cost of deploying a 10-node cluster of `c4.8xlarge` EC2 instances, while the x-axis depicts the cost changes as more new applications are added. In the ground-up approach, users must pay for each new unseen application, resulting in an ever-increasing cost proportional to the number of unseen applications. On the other hand, a universal autotuner incurs an initial cost (cost of investment) for gathering the initial training dataset and building the ML model. As more unseen applications are deployed, this cost is amortized, leading to a compensation point where the initial cost of investment is reimbursed by the savings due to the universal autotuner's ability to optimize previously unseen applications.

6.3. *Sparkle*: End-to-end autotuning framework for high-dimensional Spark configurations

Sparkle is an end-to-end framework that automatically tunes the configuration parameters of Spark deployments to optimize their performance and operational cost. Compared to previous approaches that consider application-level [6, 146, 148] or cluster-level [7] performance modeling techniques, *Sparkle* provides a more generic approach, providing a universal deep learning architecture that is able to find near-optimal configurations for diverse sets of Spark applications and dataset sizes, using a single trained model. Figure 6.5 shows an overview of the proposed framework. *Sparkle* consists of an offline and an online phase, which we discuss in the following sections.

6.3.1. Offline Phase: Data Colleciton & Performance Modeling

The offline phase concerns the collection of data used to train Sparkle's DNN performance models.

Preliminary parameter pruning: As already mentioned, Spark offers more than 150 configuration parameters [362]. However, a large portion of these parameters refer to Spark configurations not relative to performance, e.g., the name of the spark application (`spark.app.name`). Thus, as a preliminary step, these parameters have been identified and removed from the rest of the exploration process. This procedure resulted in 101 parame-

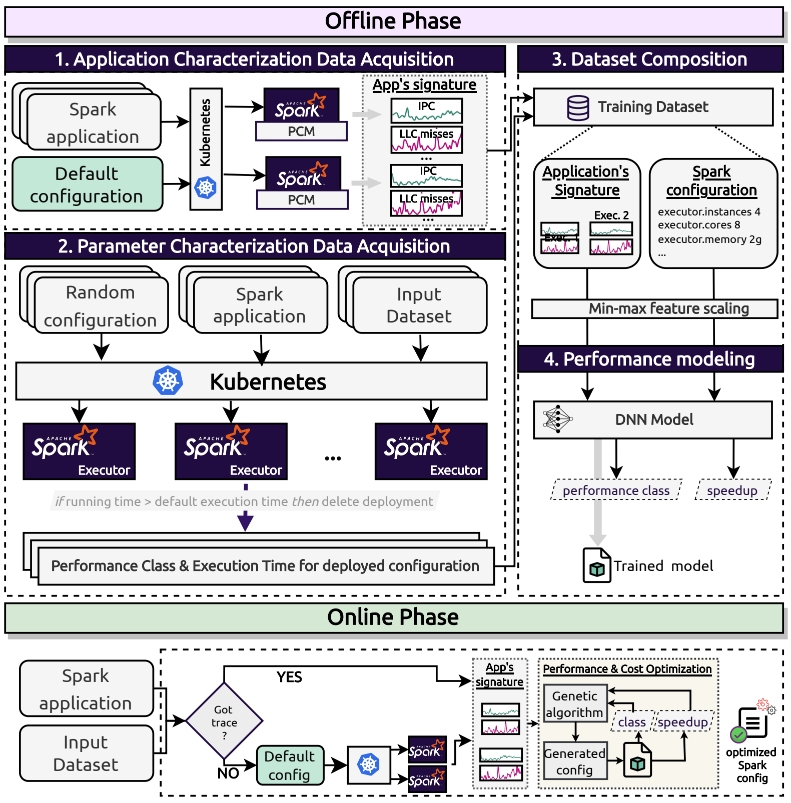


Figure 6.5.. *Sparkle*'s Architecture Overview

ters, which concern a wide range of configurations that can be applied on the spark system. These 101 parameters are either numeric configurations (e.g. `spark.executor.memory` etc.) or boolean variables (e.g. `spark.shuffle.compress`). Similar to previous works [4, 6, 7], we employ a partial factorial design of experiments for each parameter, by sampling from three up to six representative values, ranging from the lowest up to the highest acceptable ones, where the minimum and maximum values have been determined according to [362][1]. We note that for `spark.executor.instances`, we consider values ranging from 1 up to 20, which depict under-subscription, fully-subscription and over-subscription of our 10-node cluster.

[1]The detailed configuration space can be found in Appendix 6.A, Table 6.A.1.

Application Characterization Data Acquisition

As a first step, *Sparkle* collects data related to application-specific characteristics. The rationale of this step is twofold. First, it provides a performance baseline to be used as a ground-truth for assessing the efficiency of different examined configurations, during Steps 2 and 3 of the offline phase. Second, it aims to provide a "unique identifier" per application, which represents the dynamics of the application during execution. Prior scientific works have demonstrated that low-level system metrics can provide deep application-specific insights and be used to characterize running workloads on a system [313, 369]. Motivated by this observation, *Sparkle* creates this unique identifier by exploiting low-level performance counters during application's execution. Specifically, for each Spark application and each examined dataset size the framework deploys the respective combination on the cluster using the default Spark configuration. During execution, *Sparkle* continuously monitors system-wide low-level performance counter events over the allocated worker nodes. In particular, we collect information for 35 different performance events, concerning the performance (e.g., IPC, LLC misses, Memory reads and writes), the state (e.g., Core/Package C-States) and the power consumption (e.g., Processors and DRAM energy consumption) of the system, with a monitoring interval of 1 second. The result is 35 different signal traces per application with a length equal to the total execution latency for the default configuration.

Parameter Characterization Data Acquisition

In this step, *Sparkle* collects data related to the characterization of Spark parameters. The inherent purpose is to identify the execution behavior of Spark applications for different input configurations. To do so, our framework generates various Spark configurations and assess their impact on performance of applications. Each configuration is produced through a generator that randomly picks a value for each parameter (for all the 101 examined parameters) from its respective representative value space. Next, each Spark application is deployed and executed for all the different, generated configurations and for all the different input dataset sizes over the Kubernetes cluster.

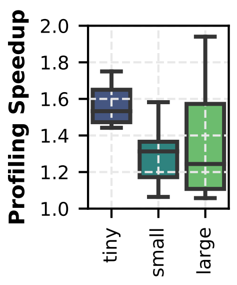
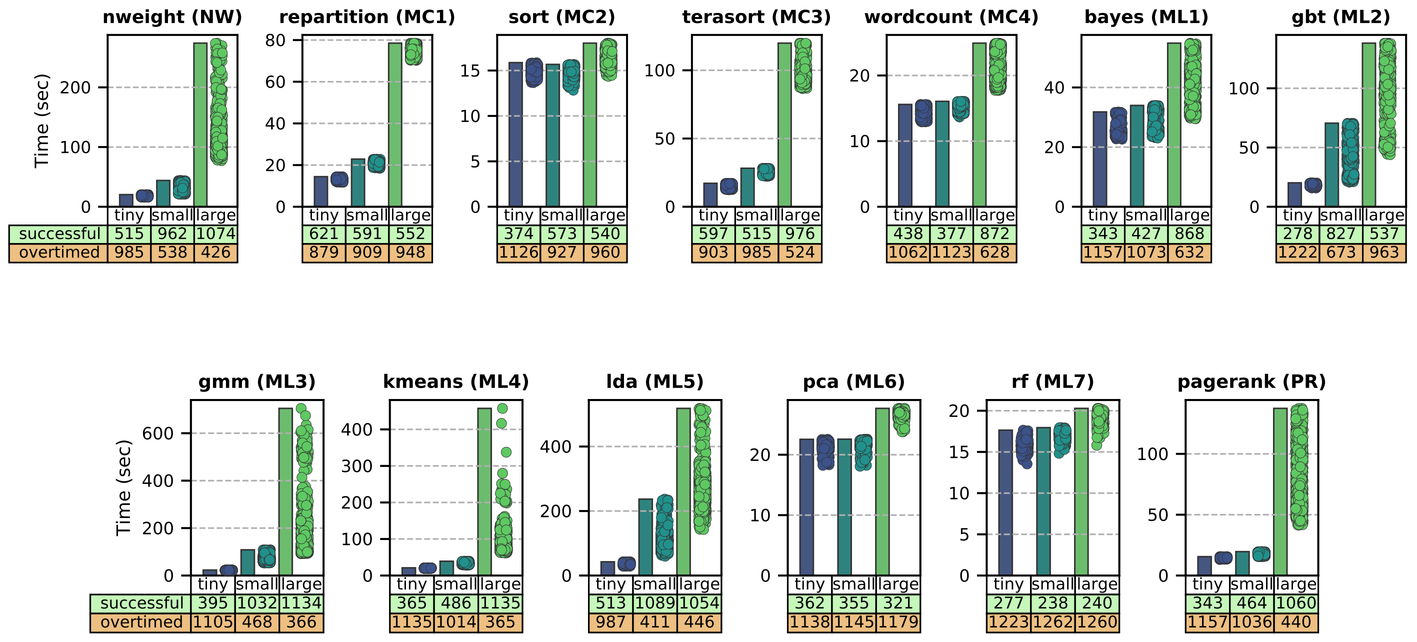


Figure 6.6.. Speedup over naive profiling approach



nweight (NW)	tiny	small	large
successful	515	962	1074
overtimed	985	538	426

repartition (MC1)	tiny	small	large
successful	621	591	552
overtimed	879	909	948

sort (MC2)	tiny	small	large
successful	374	573	540
overtimed	1126	927	960

terasort (MC3)	tiny	small	large
successful	597	515	976
overtimed	903	985	524

wordcount (MC4)	tiny	small	large
successful	438	377	872
overtimed	1062	1123	628

bayes (ML1)	tiny	small	large
successful	343	427	868
overtimed	1157	1073	632

gbt (ML2)	tiny	small	large
successful	278	827	537
overtimed	1222	673	963

gmm (ML3)	tiny	small	large
successful	395	1032	1134
overtimed	1105	468	366

kmeans (ML4)	tiny	small	large
successful	365	486	1135
overtimed	1135	1014	365

lda (ML5)	tiny	small	large
successful	513	1089	1054
overtimed	987	411	446

pca (ML6)	tiny	small	large
successful	362	355	321
overtimed	1138	1145	1179

rf (ML7)	tiny	small	large
successful	277	238	240
overtimed	1223	1262	1260

pagerank (PR)	tiny	small	large
successful	343	464	1060
overtimed	1157	1036	440

Figure 6.7.. Execution time distribution of all the examined benchmarks for different Spark configurations and diverse dataset sizes

To minimize profiling time, *Sparkle* forcefully terminates deployments which exceed the execution time of the default configuration, acquired during Step 1, and labels them as "overtimed". This design decision, even if insignificant in theory, becomes extremely efficient in practice, offering huge savings with regards to the time required to execute all the generated configurations.

In fact, Fig. 6.6 presents the profiling speedup gains over a naive profiling approach for 300 configuration deployments for all the examined benchmarks and dataset sizes. We see that depending on the dataset size and the scalability potentials of each benchmark, we can obtain almost up to two times lower profiling time in certain cases.

Profiling Insights: For the purposes of this work, we have generated and evaluated 1500 different configurations per application, through an experimental campaign of ~ 90 days. Figure 6.7 shows the execution time distribution of all the different configurations per benchmark and dataset size (scattered dots), the execution time of the default configuration (bars) and the number of successful and overtimed configurations (table). It reveals two important insights: (i) First, *parameter tuning does not have the same effect among different applications.* Indeed, we observe that certain applications (e.g., nweight, lda, pagerank) are significantly affected from the tuning process, providing up to five times reduced execution latency compared to the default configuration, whereas there are also cases where parameter tuning provide slight performance gains (e.g., sort, wordcount). (ii) Second, *parameter tuning impact on performance is directly affected by the input dataset size.* This observation is expected, as Spark is intended for big data applications and, therefore, we anticipate to have greater performance gains for larger datasets. However, there are certain cases where parameter regulation has impact on particular dataset sizes, e.g., in the case of kmeans, tuning yields extremely more efficient results for large dataset sizes compared to tiny and small ones. These observations further suggest that an efficient tuning framework should be able to exploit the inter-relationship between application specific features and the input dataset.

Dataset Composition & Pre-processing

The collected execution profiles and performance classes per benchmark/dataset/configuration triplet form the final dataset used for training and testing of *Sparkle's* performance model. Due to the different scales on the assembled data, we perform a min-max normalization, in order to bring the data values between the range [0, 1]. Specifically, regarding the applications' signatures, we normalize each performance event separately, with the minimum and maximum values derived over all the examined benchmarks, rather than in a per-benchmark manner. This allows to have a relative interpretation of system dynamics between different applications. For configuration parameters, first, we convert and map any boolean or string parameters to an identical integer representation. A typical example is `spark.serializer` which is converted from `<JavaSerializer,KryoSerializer>` to `<0,1>` respectively. After the conversion, we normalize each parameter separately, using as minimum and maximum the respective values from the parameter's representative values.

Performance modeling

Sparkle follows a dual performance modeling approach. Given an application's signature and a Spark configuration, it predicts if a speedup is expected in a yes/no manner (classification), aiming to discard overtimed predictions, and estimates the extend of the speedup (regression), if exists. *Sparkle* tackles simultaneously both the classification and regression tasks through an efficient deep temporal architecture, as follows:

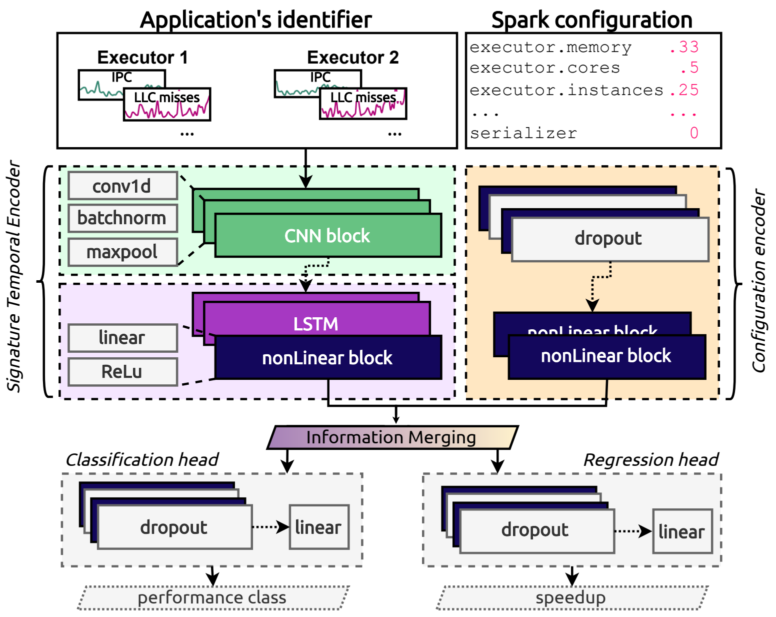
Anatomy of DNN architecture: Figure 6.8 depicts the proposed DNN architecture and its components for *Sparkle*'s performance model. In more details:

• *Signature Temporal Encoder:* This component aims to encode applications' signatures of uneven length into fixed-sized feature vectors using a hybrid CNN-LSTM architecture. First, we apply a 1D CNN backbone of three consecutive 1D convolutional layers, accompanied by ReLU activations, Bath-Normalization [370], which assists convergence, and Max-Pooling operations. The 1D CNN backbone provides a coarse "organization" of the temporal information into deep features, while considerably reduces the length of the sequence with the max-pooling operations (by a factor of 27, overall). On top of the CNN backbone, we apply an one-directional LSTM network of two layers in order to effectively encode the reduced sequence of deep features into a fixed-sized vector.

• *Configuration Encoder:* The spark configuration is encoded through a feed-forward network consisted of four linear layers. Between consecutive linear layers, we add ReLU and Dropout layers [371]. The latter are used to prevent overfitting.

• *Information Merging:* The two extracted encodings are merged via a concatenation function.

• *Classification & Regressions Heads:* Both components have the same architectural structure, consisted of three fully connected linear layers, intervened by ReLU and Dropout



```
executor.memory    .33
executor.cores      .5
executor.instances .25
...                ...
serializer           0
```

Figure 6.8.. DNN architecture for performance modeling

layers. The output is a single value. For the classification head, the output is followed by a sigmoid activation and Binary Cross Entropy (BCE) loss is used for training, while for its regression counterpart, Mean Squared Error (MSE) loss is adopted.

Datasize Awareness: We note that information about the dataset size is not explicitly used as input to *Sparkle*'s model. Dataset-size can not be faithfully encoded into a single value, since "tiny" size, for example, may have very different characteristics for different applications (see Table 4.3). Information about dataset-size is implicitly contained in the application's identifier, thus the temporal encoder of the DNN extracts the relevant and useful features from application's signature.

Training Details: Training is performed using the Adam optimizer [372] and a cosine annealing scheduling tactic with warm restarts, used to assist convergence to better performing optima [373]. We train our system with a multitask loss by adding the individual losses of classification (BCE) and regression (MSE). Such joint training of a shared backbone for both tasks not only requires reduced computational resources, but also slightly outperformed an initial two-stage approach of two independent networks, one for classification and one for regression, with prediction errors being reduced around 0.3% for both tasks.

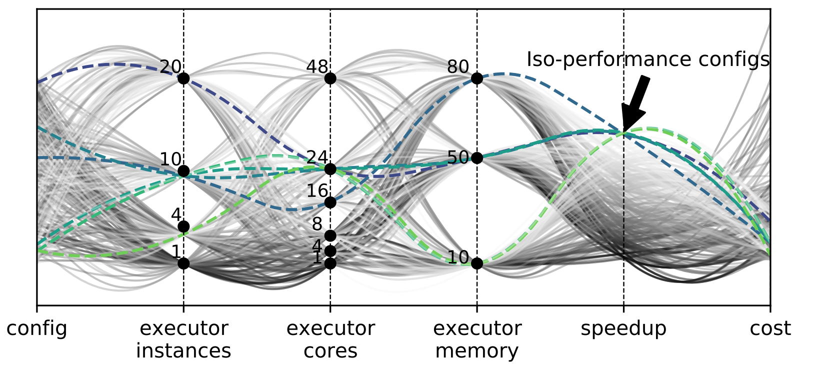


Figure 6.9.. Speedup gains and cost expenditure of 300 different configuration for `gmm` application.

6.3.2. Online Phase

In the online phase, *Sparkle* aims to identify optimized configurations for applications deployed on the cluster. In case of uncharacterized applications deployed, *Sparkle* executes them using the default configuration and captures and stores the respective PCM metrics used as their "unique identifier". Then, the framework iteratively queries the performance models (of Step 4) for different parameter values to determine an optimized configuration.

The case of iso-performance configurations: It is possible that different configurations lead to similar performance gains over default execution, due to the strong conjuction between the various Spark parameters. For example, certain applications may benefit equally from horizontal (increasing executor instances) and vertical (increasing cores/memory per executor) scaling. As a representative example, Fig. 6.9 shows the speedup provided by 300 different configurations for the `gmm` application. Each line represents a different configuration and the intersections with the vertical axes depict the value of the respective parameter. We see that different sized deployments provide identical speedup over default configuration. However, this is not the case in terms of operational cost, as more “fat" deployments typically lead to more expensive deployment expenses.

Optimization approach: *Sparkle* adopts meta-heuristic optimization, to efficiently explore the underlying search space and ensure convergence towards optimal solutions. Based on the iso-performance observations, we differentiate with prior-art that focus solely on performance efficiency [6, 7], targeting to a dual-objective optimization problem, i.e., minimize the deployment cost while maximizing the performance of the applied

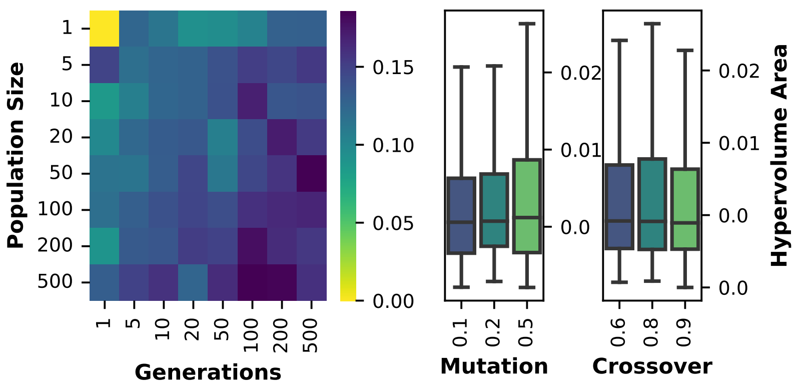


Figure 6.10.. Impact of NSGA-II's parameters on the final Pareto

configuration. Specifically, *Sparkle* follows an open-loop optimization approach, where the optimization process is applied on top of the trained DNN model to evaluate the dynamics of the solution space. This approach leads to a set of Pareto optimal solutions [374], which trade-off performance for cost efficiency and vice-versa. *Sparkle* employs the Non-dominated Sorting Genetic Algorithm (NSGA-II) to traverse the solution space, due to its ability to escape local optima and provide fast convergence to efficient solutions [6, 261, 375]. NSGA-II is an evolutionary algorithm and operates for a number of generations, where at each generation the elites of a population are given the opportunity to be carried on to the next one. Based on the mutation and crossover operators, it creates new offspring populations to be examined in subsequent generations. In order to determine an optimal set of values regarding NSGA-II's hyperparameters (population size, generations, mutation probability and crossover rate), we explore the impact of each NSGA-II hyperparameter in the final pareto front. We examine the effect of each hyper-parameter by evaluating the area of the hypervolume [376] which is dominated by the provided set of solutions with respect to a reference point (default execution) for all benchmark/dataset pairs. Figure 6.10 shows how area is affected for different values of the examined parameters [377]. We observe that, as the number of generations and population size increases, so does the distance of the Pareto front from the default point. However, this also leads to increased exploration time budgets, as these parameters greatly impact the search space of the algorithm. Moreover, we notice that mutation and crossover parameters insignificantly affect the quality of the solutions. Based on the above, we select a population size equal to 10, which provides a good trade-off between the exploration time required and the quality of the final Pareto front. Regarding the mutation probability and the crossover operator, we choose the values of 0.5 and 0.8 respectively.

6.3.3. *Sparkle's* Implementation

We implement Sparkle on top of Spark (`v3.0.1`) using Python (`v3.8.5`). For the generation of random Spark configurations, we utilize the Opentuner framework [378], that provides an automated way of defining variable search spaces, by specifying parameters that should be tuned. Moreover, in order to monitor low-level performance counters, we utilize the Performance Counter Monitoring (PCM) API [189], which provides a plethora of hardware performance counters for each logical core, each socket, as well as the whole server system. Our proposed DNN model has been implemented using the PyTorch framework [142] and scikit-learn libraries [283]. Finally, regarding the implementation of our optimizer, we have utilized pymoo [379], a multi-objective optimization framework in Python. The source-code of *Sparkle* is available under an open-source license[2].

6.4. Evaluation

We implement Sparkle on top of Spark (`v3.0.1`). For the random sampling Spark configurations, we utilize the Opentuner framework [378]. The Performance Counter Monitoring (PCM) API [189] used for low-level performance counters monitoring, whereas our proposed DNN model has been implemented using the PyTorch framework [142]. Finally, regarding the NSGA-II implementation, we have utilized the multi-objective optimization framework pymoo [379]. The source-code of *Sparkle* is publicly available[3].

We evaluate *Sparkle* across four dimensions: *i)* by its ability to accurately model and assess the performance of our examined applications, *ii)* by its prediction accuracy compared to other ML solutions proposed in prior art, *iii)* by its ability to extrapolate to unseen deployments and *iv)* by the quality of the Pareto solutions derived through the optimization phase.

6.4.1. Accuracy evaluation

First, we evaluate the ability of Sparkle to model the performance of our examined benchmarks w.r.t. different Spark configurations. Specifically, we measure the accuracy of our

[2]Omitted for blind review
[3]Omitted for blind review

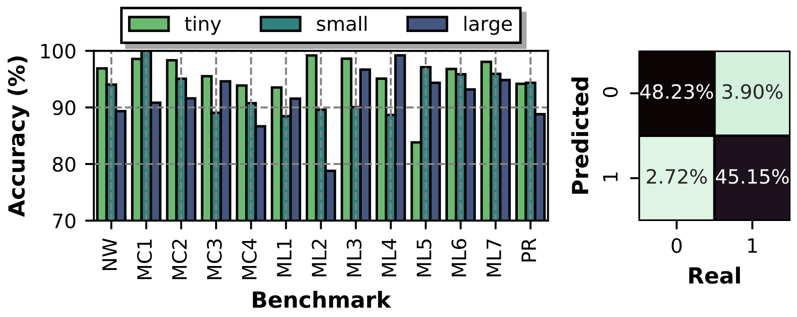


Figure 6.11.. *Left:* Classification accuracy for unseen Spark configurations per benchmark and dataset. *Right:* Confusion Matrix.

DNN models for predicting *i)* the performance class (speedup/slowdown) and *ii)* the actual speedup provided by configurations that belong in the former class. We partition the dataset produced through the data collection phase (Sec. 6.3.1) in two subsets of 90% (training set) and 10% (test set) of the samples, where the test dataset consists of benchmark-dataset-configuration triplets not used during training. To measure accuracy, we follow a 10-fold cross-validation procedure.

Classification Accuracy: Figure 6.11 shows the accuracy results of our model for each examined benchmark and dataset size. For the majority of the benchmarks Sparkle provides extremely high accuracy of more than 90% regardless of the examined dataset size, showing its ability to correctly predict whether a given configuration if more efficient compared to default execution. We also plot the confusion matrix, which depicts the percentage of true/false positive/negative predictions made by the model, where 0 denotes the slowdown and 1 the speedup class. We see that, overall, Sparkle achieves an average accuracy of ≈ 93%. Moreover, the model also reveals high precision, sensitivity and specificity accuracy of 93.35%, 92.62% and 93.43% respectively[4]. Last, the percentage of false negative predictions, i.e., mistaken prediction that a configuration will provide speedup, is kept at low levels, with approximately 2.7% of the total instances belonging to this case, showing that Sparkle is not susceptible to mistaken predictions that will lead to slow-downed executions.

Regression Accuracy: Next, we evaluate the accuracy of Sparkle's speedup prediction model by measuring the Mean Absolute Percentage Error (MAPE). Figure 6.12 shows the respective results, showing that Sparkle provides robust predictions regardless of the benchmark and dataset size, with an average of 7.2% MAPE overall. Also, by examining the actual and predicted speedup values, we observe that this error is distributed evenly

[4]**Precision** is the ratio (correct/all) instances labeled as *speedup*. **Sensitivity** is the ratio of (correct/real) instances labeled as *speedup*. **Specificity** is the ratio of (correct/real) labeled as *slowdown*.

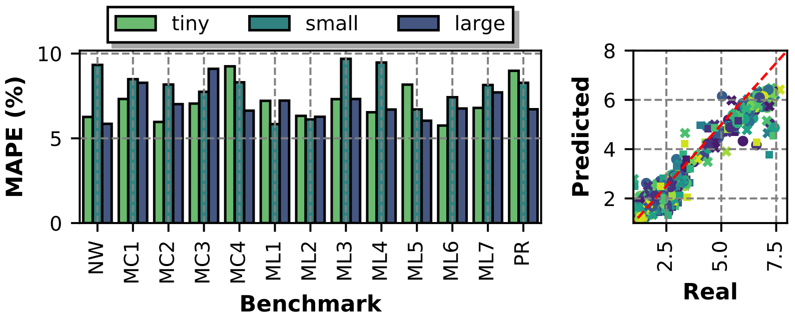


Figure 6.12.. *Left:* Regression accuracy (MAPE) for unseen Spark configurations per benchmark and dataset. *Right:* Predicted vs real speedup over all benchmarks and datasets.

across all applications and datasets, as most of the points reside close to the 45^o regression line. Note that for large speedup values (> 5.0), we observe that our system underestimates the actual speedup. Such behavior is to be expected since 77% of the speedup values are under 2.0, while almost 84% are under 2.5.

Impact of amount of training data on accuracy: Last, we examine how predictions are affected by the number of instances used to train our models, by measuring the accuracy achieved for different portions of the training set. Figure 6.13 shows the respective results, both for classification and regression models. We observe that the overall accuracy of both models steadily increase with the number of training instances used, up to a certain plateau of roughly 50% of the dataset, after which no or minimal improvement is noticed. This percentage corresponds to approximately 650 different configurations examined for each benchmark/dataset pair, which is comparable or even less compared to the number of training configurations required by prior art [4, 6] that follow a per-benchmark modeling approach. This is due to the fact that Sparkle's universal model allows the encoded information to be shared between different benchmarks, thus, requiring less configurations to be examined per application.

6.4.2. Comparative Analysis

We further compare Sparkle's regression model against five different ML approaches for modeling performance of SPARK applications, proposed in prior research. Specifically, we implement from scratch the following models: *i)* Random Forest (RF) [4, 7, 147], *ii)* Multi-Layer Perceptron (MLP) [4], *iii)* Support Vector Machine (SVM) [5,265] and *iv)* hierarchical modeling (xgboost) [6]. For RF, MLP and SVM, we build a performance model

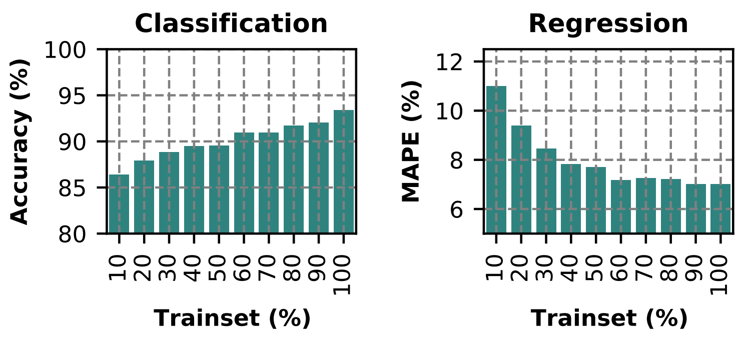


Figure 6.13.. Models' accuracy for different portions of the dataset used during training. *Left:* Classification. *Right:* Regression.

per benchmark and dataset size, which is the methodology followed in prior art [4,5]. In the case of RF, we also examine a cluster-wise performance modeling approach, where the assignment of applications into clusters is derived from [7]. Last, for hierarchical modeling, the developed model is benchmark-specific and dataset-aware, as described in [6]. We should note that each modeling approach differs, both qualitatively and quantitatively, in the number of Spark parameters used as inputs to the model, which we choose based on the tables found in the respective papers. Last, we fine-tune the hyper-parameters of the ML models to increase the prediction accuracy. Again, we use a 10-fold cross-validation to measure the accuracy per approach.

Regression Accuracy: Figure 6.14 presents the MAPE of each modeling approach per benchmark and dataset size examined. We observe that even though Sparkle adopts a universal performance modeling process, it manages to achieve comparable and sometimes even better prediction accuracy compared to the application- and dataset-specific estimators. On average, Sparkle delivers a MAPE increment of 1.3% compared to the RF case, whereas it outperforms MLP and SVM approaches by providing 1.1% and 2.3% lower MAPE respectively. Moreover, Sparkle also prevails over datasize-aware and cluster-wise approaches, by providing 4.7% and 17.5% lower MAPE respectively. What is of great interest is that Sparkle tends to provide better predictions on applications that present more spread performance distributions. This is more evident in the case of large dataset sizes, where applications reveal higher performance variability (as shown in Fig. 6.7) and where Sparkle clearly outperforms the other approaches for the majority of the cases, by providing 1.5% up to 6.3% lower MAPE compared to application and dataset specific performance modeling.

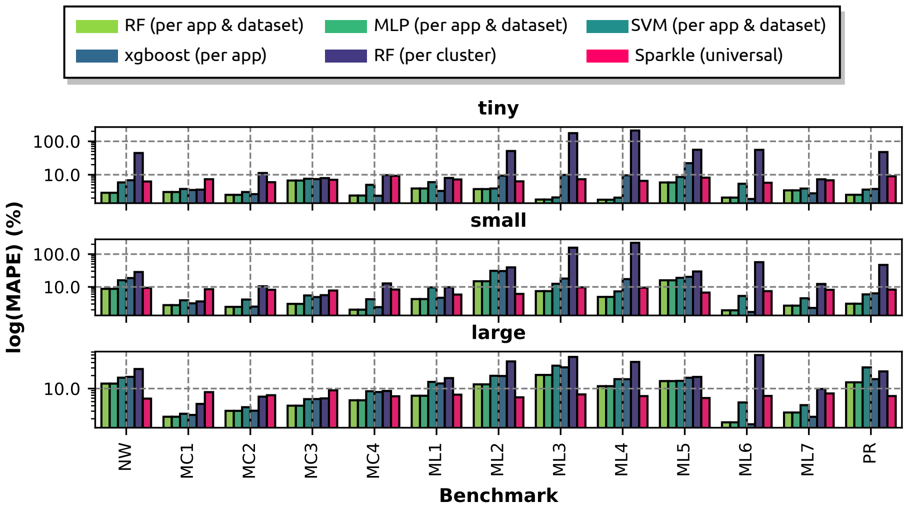


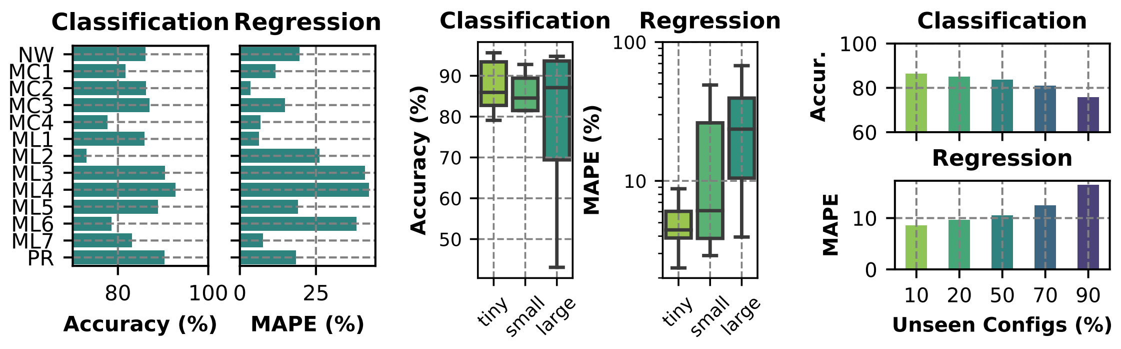
Figure 6.14.. Comparison of Sparkle's regression model accuracy with application- and dataset-specific [4, 5], dataset-aware [6] and cluster-wise [7] performance modeling techniques used in prior art.

6.4.3. Generalization study

Sparkle's universal DNN architecture allows it to be leveraged in a seamless manner to extrapolate to scenarios not encountered during the training data acquisition phase. Thus, next we investigate *Sparkle*'s generalization capabilities, i.e., *"how efficient can we model performance of unknown applications, dataset sizes and configurations?"*. Figure 6.15 shows the respective results per case.

Unseen applications: We evaluate the accuracy for unseen applications using an application-granular leave-one-out validation. Figure 6.15a shows the accuracy and MAPE for predicting the performance class and speedup respectively. We see that *Sparkle* provides high classification accuracy, ranging from 70% up to 90%, which reveals the framework's capability to quickly and efficiently identify profiles that will lead to faster execution compared to the default configuration. Regarding the estimations of the actual speedup, *Sparkle*'s predictions are not so robust, with the MAPE ranging from 4% up to 31% and an average value of 19%.

Unseen dataset sizes: We further explore how *Sparkle* behaves in cases of unseen dataset sizes. This time we perform an application/dataset combination leave-one-out validation, i.e., we train the model with all available data except a specific application/dataset set, which forms the "unseen" case. Figure 6.15b shows the distribution of



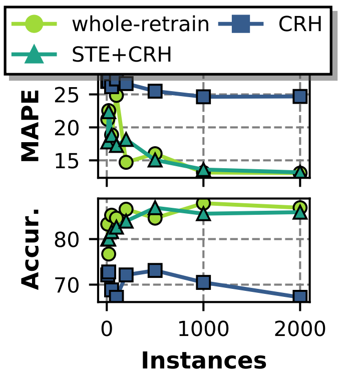
(a) Unseen Applications (b) Unseen Dataset sizes (c) Unseen Configurations

Figure 6.15.. Sparkle's extrapolation ability to a) unseen applications b) unseen dataset sizes and c) unseen configs.

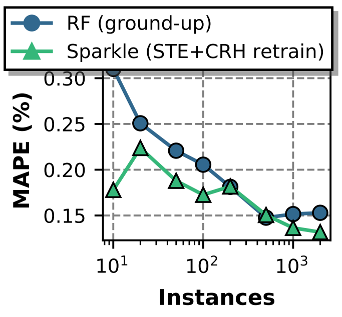
accuracy and MAPE per dataset size over all the examined applications. We see that for both tasks, *Sparkle* is able to efficiently generalize from larger to smaller datasets, whereas it struggles to achieve the opposite. This is expected, since, as discussed in Sec. 6.3.1, the degree of speedup achieved through parameter tuning increases with the size of the input data. This matter is less evident for classification, where even for large unseen datasets, the median accuracy is kept at high levels (≈ 85%), yet the lower whisker resides between 10% and 70%, showing increased variability for certain cases. For regression, generalizing to larger datasets becomes prohibitive, with the median MAPE of the "large" case reaching ≈ 25%. Even when extrapolating to intermediate dataset sizes (i.e., from "tiny" and "large" to "small"), we experience a non-robust behavior, with a low median MAPE of ≈ 4%, but a dispersed overall distribution approaching ≈ 50% error for specific instances.

Unseen configurations: Last, we evaluate *Sparkle*'s performance on completely unseen configurations, i.e., we use as our test set Spark configurations that have not been encountered during training by any application/dataset pair. Figure 6.15c presents the results, where the x-axis shows the percentage of the 1500 configurations considered as unseen during training. In this case, *Sparkle* achieves more robust predictions, with the accuracy and MAPE showing a small deviation up to the 50% coverage, after which a decline of ≈ 5% is observed for both cases. Nevertheless, we see that even for low unseen percentage values, we experience ≈ 10% and ≈ 2% performance loss for accuracy and MAPE respectively compared to our initial trained model, which highlights the need for information sharing across different applications.

Model retraining: The aforementioned generalization results show a level of robust-



(a) Impact of retraining different components of Sparkle's model on overall accuracy



(b) Sparkle's retraining vs ground-up training of an application-specific model

Figure 6.16.. Retraining for unseen apps a) w.r.t. model's components b) vs RF ground-up training

ness to unseen data, but, as expected, performance is non-trivially decreased. To address this issue, a retraining scheme is considered, where the initial data are trained alongside a number of instances from the unseen applications in a fine-tuning rationale; the DNN is initialized as a pre-trained model to the already seen data and then trained with a reduced learning rate using the Adam optimizer, trying to find a neighboring optima where the unseen data is also effectively modeled. Adam optimizer adapts well to sparse information, lending from the AdaGrad optimizer [380] that simulates larger learning rate for infrequent parameters, thus being effective even for a few number of new instances.

We explore the effectiveness of three retraining approaches to improve the accuracy on newly collected data of unseen applications, i.e., *i) whole-retrain:* update all the weights of the model *ii) STE+CRH:* update only the weight of the signature temporal encoder (STE) and the classification and regression heads (CRH) and freeze the ones of the configuration encoder and *iii) CRH:* update only the weights of the classification and regression heads. We perform our evaluation on the `gbt` (ML2) benchmark, which experienced the highest accuracy prediction error in the previous experiments. Figure 6.16a shows the MAPE (top) and classification accuracy (bottom) of the three retraining approaches. We see that retraining the STE along with the CRH provides analogous results with retraining all the weights of the model, which shows that *Sparkle*'s configuration encoder can effectively identify important configuration features for unseen applications. Moreover, training only the CRH of the network does not provide any accuracy increment, revealing the importance of encoding application-related information to the model.

To pinpoint the significance of *Sparkle*'s universal modeling approach compared to application specific ones followed in prior works, we further examine the MAPE achieved by retraining the STE+CRH of *Sparkle* and by an application and dataset specific ground-up training of an ML model, for different number of training instances available. We select Random Forest as the adversary model, since Sec. 6.4.2 showed that it is the most accurate estimator among the ones examined. Figure 6.16b shows the MAPE achieved by the two approaches for different number of training instances. *Sparkle* steadily provides more accurate predictions compared to the application-tailored RF model up to the point of 200 instances, where the two models yield similar prediction scores. Moreover, as the number of instances increases above 500, *Sparkle* achieves to widen the accuracy gap again by providing 2% less MAPE.

• *Generalization Discussion:* Generalization to unseen settings forms a challenging task, especially when there is no guarantee that the new "category" of data shares similar structure with the existing ones. Initial experimentation suggests no indication of overfitting behavior, which leads to the assumption that there are "missing parts" for unseen settings that cannot be extrapolated from existing information. Possible causes are completely new patterns for specific applications or a domain shift gap, connected to the domain adaptation problem [381]. Nevertheless, we see that the universal viewpoint of *Sparkle* assists on bringing the trained DNN model close enough to an optimum that can extrapolate behavioral patterns of new applications even from sparse information. Future directions could include self-supervised ideas such as contrastive learning, along with appropriate data augmentation, in order to extract deep features with increased generalization abilities. Moreover, a more in-depth analysis on the re-training concept could be beneficial for handling a large volume of new unseen applications, datasets and configurations. Specifically, online training schemes and domain adaptation techniques could be leveraged, to quickly adapt to new unseen apps/datasets without sacrificing the existing performance (i.e., avoiding catastrophic forgetting).

6.4.4. Genetic Optimization

Last, we examine *Sparkle's* genetic optimization step, in terms of its ability to deliver performance- and cost-efficient deployments. To calculate cost, we consider the pricing of 15 different AWS instances, with diverse characteristics in terms of virtual cores and memory capacity offered. Specifically, we calculate the cost of each deployment by identifying the least-expensive, "right-sized" instance w.r.t. the value of `executor.cores` and `executor.memory` parameters and multiplying its price with the number of `executor.instances` and the total execution time of the respective configuration.

***Sparkle* vs. default execution:** Fist, we evaluate the speedup and cost gains offered by *Sparkle* compared to the naive default execution. Overall, *Sparkle* offers solutions that maintain deployment expenses between 0.5% higher (by sacrificing cost for performance) up to 54% lower, with an average of 13% cost gains, compared to default execution.

Regarding speedup gains, *Sparkle* achieves to provide speedups ranging from ×1.05, in case of application/dataset pairs that are inherently not affected by parameter tuning (as shown in Sec. 6.3.1), up to ×6.8, with an average of ×1.72 speedup over all the examined cases.

***Sparkle* vs. model-less optimization:** Last, we compare the set of Pareto solutions provided by our framework with a model-less optimization approach, where each individual (Spark configuration) from the population is natively executed on the cluster. We set the termination criterion of the optimization algorithm to 1 hour, allowing the model-less approach to evaluate an adequate number of Spark configurations. Figure 6.17 presents the results per benchmark, where blue-colored points reveal the Pareto solutions proposed by the model-less optimizer and pink-colored the ones proposed by *Sparkle*. For *Sparkle*, we natively execute the final Pareto front proposed, to obtain the real performance per configuration, rather than the model's estimation. With opaque colorings, we indicate the final Pareto fronts per dataset size, which include non-dominant solutions provided by either optimization approach. *Sparkle* prevails over the model-less optimization approach, as the majority of the points relying in the final Pareto front belong to its own set of proposed solutions. Specifically, *Sparkle* covers approximately 65% of the final Pareto front, while the rest 35% belongs to the native approach. Moreover, it expands the hypervolume area formed w.r.t. the point of default execution, by offering 9.55% increased area compared to model-less optimization.

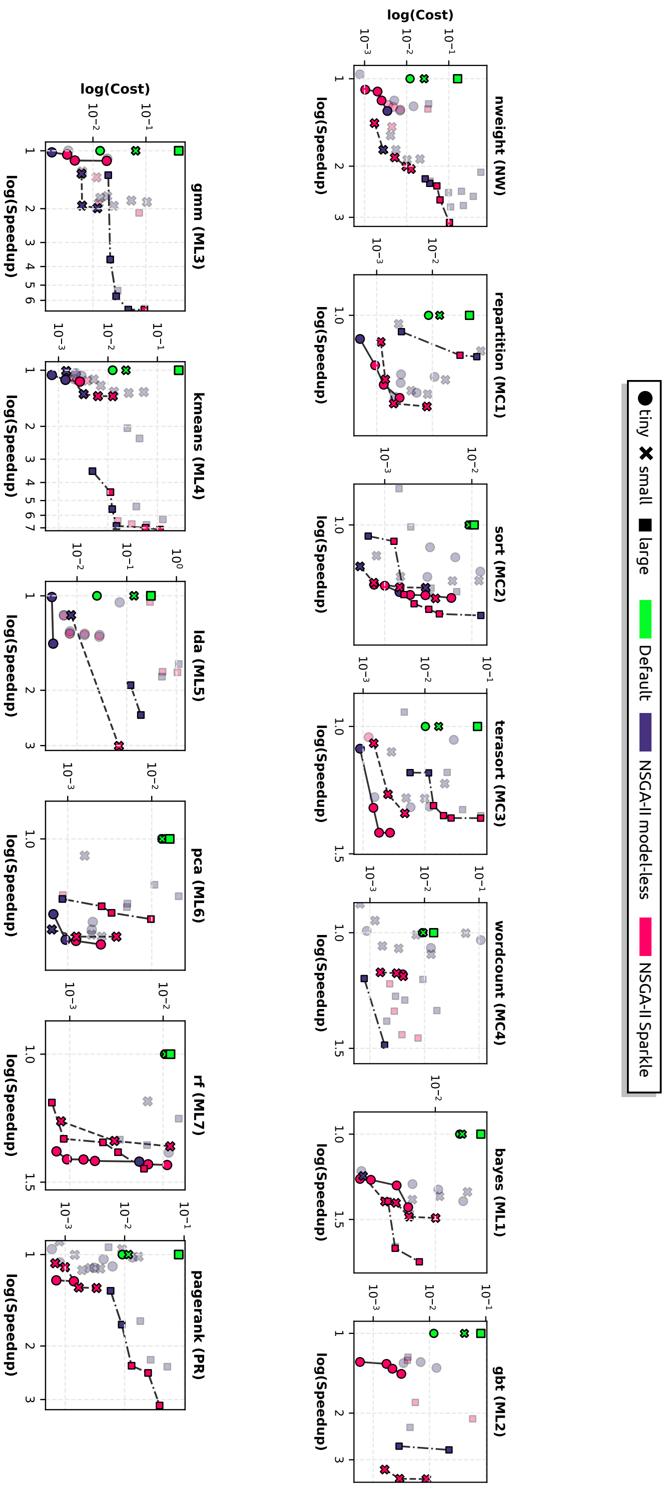


Figure 6.17.. Pareto solutions proposed by *i)* a model-less NSGA-II optimization approach and *ii) Sparkle*. Opaque coloring shows the final Pareto that includes non-dominated points among both approaches. Green points depict default execution.

6.5. Conclusion

This chapter presents *Sparkle*, a deep-learning driven autotuning framework for high-dimensional Spark analytics. *Sparkle* advances over state-of-the-art approaches by providing a universal performance modeling approach, rather than application- and/or dataset-specific ones considered in prior art. Moreover, *Sparkle* expands over the entire configuration space, thus, completely eliminating the need for human-in-the-loop or statistical approaches to identify the importance of each parameter.

Chapter's Appendix

6.A. Spark parameters considered

Table 6.A.1.: Spark parameters considered within *Sparkle* along with their default value and the examined value range investigated in this work.

#	Parameter Name	Description	Default Value	Examined Values
1	`shuffle.file.buffer`	Size of the in-memory buffer for each shuffle file output stream, in KiB unless otherwise specified.	32k	8k, 32k, 128k
2	`shuffle.sort.bypassMergeThreshold`	In the sort-based shuffle manager, avoid merge-sorting data if there is no map-side aggregation and there are at most this many reduce partitions.	200	50, 200, 800
3	`speculation.interval`	How often Spark will check to speculate tasks.	100ms	10ms, 100ms, 500ms
4	`speculation.multiplier`	How many times slower a task is than the median to be considered for speculation.	1.5	1.1, 1.5, 5
5	`speculation.quantile`	Fraction of tasks which must be complete before speculation is enabled for a particular stage.	0.75	0.5, 0.75, 0.85
6	`broadcast.blockSize`	Block size for `TorrentBroadcastFactory`	4m	1m, 4m, 16m
7	`io.compression.codec`	The codec used to compress internal data such as RDD partitions, event log, broadcast variables and shuffle outputs.	lz4	snappy, lzf, lz4
8	`io.compression.lz4.blockSize`	Block size used in LZ4 compression.	32k	16k, 32k, 64k
9	`io.compression.snappy.blockSize`	Block size used in Snappy compression.	32k	16k, 32k, 64k
10	`kryoserializer.buffer.max`	Maximum allowable size of Kryo serialization buffer.	64m	32m, 64m, 128m
11	`kryoserializer.buffer`	Initial size of Kryo's serialization buffer	64k	32k, 64k, 128k

12	`executor.cores`	The number of cores to use on each executor.	6	1, 4, 8, 16, 24, 48
13	`driver.memory`	Amount of memory to use for the driver process.	1g	500m,1g,8g
14	`storage.memoryMapThreshold`	Size of a block above which Spark memory maps when reading a block from disk	2m	1m, 2m, 4m
15	`network.timeout`	Default timeout for all network interactions.	120s	60s, 120s, 240s
16	`locality.wait`	How long to wait to launch a data-local task before giving up and launching it on a less-local node.	3s	1s, 3s, 10s
17	`scheduler.revive.interval`	The interval length for the scheduler to revive the worker resource offers to run tasks.	1s	1s, 3s
18	`shuffle.compress`	Whether to compress map output files.	true	true, false
19	`memory.fraction`	Fraction of (heap space - 300MB) used for execution and storage.	0.6	0.4, 0.6, 0.7
20	`shuffle.spill.compress`	Whether to compress data spilled during shuffles.	true	true, false
21	`speculation`	If set to "true", performs speculative execution of tasks.	false	true, false
22	`broadcast.compress`	Whether to compress broadcast variables before sending them.	true	true, false
23	`rdd.compress`	Whether to compress serialized RDD partitions.	false	true, false
24	`memory.storageFraction`	Amount of storage memory immune to eviction, expressed as a fraction of the size of the region set aside by `spark.memory.fraction`.	0.2, 0.5, 0.8	0.2, 0.5, 0.8
25	`memory.offHeap.enabled`	If true, Spark will attempt to use off-heap memory for certain operations.	false	true, false
26	`memory.offHeap.size`	The absolute amount of memory which can be used for off-heap allocation, in bytes.	0	25m, 50m
27	`driver.maxResultSize`	Limit of total size of serialized results of all partitions for each Spark action in bytes.	1g	500m,1g,4g
28	`reducer.maxReqsInFlight`	Limits the number of remote requests to fetch blocks at any given point.	Int. Max-Value	2147483647, 200000000
29	`reducer.maxBlocksInFlight PerAddress`	Limits the number of remote blocks being fetched per reduce task from a given host port.	Int. Max-Value	2147483647, 200000000

30	`maxRemoteBlockSizeFetchToMem`	Remote block will be fetched to disk when size of the block is above this threshold in bytes.	200m	20m, 1g
31	`shuffle.io.maxRetries`	Number of retries for fetches that fail due to IO-related exceptions.	3	2, 3, 6
32	`shuffle.io.numConnectionsPerPeer`	Connections between hosts are reused in order to reduce connection buildup for large clusters.	1	1, 3
33	`shuffle.io.preferDirectBufs`	Off-heap buffers are used to reduce garbage collection during shuffle and cache block transfer.	true	true,false
34	`shuffle.io.retryWait`	How long to wait between retries of fetches.	5s	3s, 5s, 8s
35	`kryo.unsafe`	Whether to use unsafe based Kryo serializer.	false	true, false
36	`serializer.objectStreamReset`	When serializing using `JavaSerializer`, the serializer caches objects to prevent writing redundant data.	100	-1, 50, 100, 200
37	`storage.replication.proactive`	Enables proactive block replication for RDD blocks.	false	true, false
38	`cleaner.periodicGC.interval`	Controls how often to trigger a garbage collection.	30min	15min, 30min, 60min
39	`cleaner.referenceTracking`	Whether to track references to the same object when serializing data with Kryo.	true	true, false
40	`cleaner.referenceTracking.blocking`	Controls whether the cleaning thread should block on cleanup tasks	true	true, false
41	`cleaner.referenceTracking.blocking .shuffle`	Controls whether the cleaning thread should block on shuffle cleanup tasks.	false	true, false
42	`cleaner.referenceTracking .cleanCheckpoints`	Controls whether to clean checkpoint files if the reference is out of scope.	false	true, false
43	`executor.heartbeatInterval`	Interval between each executor's heartbeats to the driver.	10s	5s, 10s, 20s
44	`files.fetchTimeout`	Communication timeout to use when fetching files added through `SparkContext.addFile()`	60s	30s, 60s, 120s
45	`files.maxPartitionBytes`	The maximum number of bytes to pack into a single partition when reading files.	128MB	70000000', 134217728, 190000000
46	`files.openCostInBytes`	The estimated cost to open a file, measured by the number of bytes could be scanned at the same time.	4MiB	2000000, 4194304, 9000000

47	`rpc.message.maxSize`	Maximum message size (in MiB) to allow in "control plane" communication.	128	16, 128, 512
48	`port.maxRetries`	Maximum number of retries when binding to a port before giving up.	16	8, 16, 32
49	`rpc.numRetries`	Number of times to retry before an RPC task gives up.	3	2, 3, 6
50	`rpc.retry.wait`	Duration for an RPC ask operation to wait before retrying.	3s	2s, 3s, 6s
51	`rpc.askTimeout`	Duration for an RPC ask operation to wait before timing out.	120s	60s, 120s, 240s
52	`rpc.lookupTimeout`	Duration for an RPC remote endpoint lookup operation to wait before timing out.	120s	60s, 120s, 240s
53	`core.connection.ack.wait.timeout`	How long for the connection to wait for ack to occur before timing out and giving up.	120s	60s, 120s, 240s
54	`locality.wait.node`	Customize the locality wait for node locality.	3s	1s, 3s, 10s
55	`locality.wait.process`	Customize the locality wait for process locality.	3s	2s, 3s, 6s
56	`locality.wait.rack`	Customize the locality wait for rack locality.	3s	2s, 3s, 6s
57	`scheduler.maxRegisteredResources WaitingTime`	Maximum amount of time to wait for resources to register before scheduling begins.	30s	15s, 30s, 60s
58	`scheduler.mode`	The scheduling mode between jobs submitted to the same SparkContext.	FIFO	FIFO, FAIR
59	`scheduler.listenerbus.eventqueue .capacity`	Capacity for event queue in Spark listener bus.	10000	1000, 10000, 50000
60	`scheduler.blacklist .unschedulableTaskSetTimeout`	The timeout in seconds to wait to acquire a new executor and schedule a task before aborting a TaskSet which is unschedulable.	120s	60s, 120s, 240s
61	`blacklist.timeout`	How long a node or executor is blacklisted for the entire application.	60min	30min, 60min, 120min
62	`blacklist.task .maxTaskAttemptsPerExecutor`	How many times it can be retried on one executor before the executor is blacklisted for a given task.	1	1, 3
63	`blacklist.task .maxTaskAttemptsPerNode`	How many times it can be retried on one node, before the entire node is blacklisted for a given task.	2	2, 4
64	`blacklist.stage .maxFailedTasksPerExecutor`	How many different tasks must fail on one executor, within one stage, before the executor is blacklisted for that stage.	2	2, 4

65	`blacklist.stage .maxFailedExecutorsPerNode`	How many different executors are marked as blacklisted for a given stage, before the entire node is marked as failed for the stage.	2	2, 4
66	`blacklist.application .maxFailedTasksPerExecutor`	How many different tasks must fail on one executor, in successful task sets, before the executor is blacklisted for the entire application.	2	2, 4
67	`blacklist.application .maxFailedExecutorsPerNode`	How many different executors must be blacklisted for the entire application, before the node is blacklisted for the entire application.	2	2, 4
68	`blacklist.killBlacklistedExecutors`	If "true", allows Spark to automatically kill, and attempt to re-create, blacklisted executors.	false	true, false
69	`task.cpus`	Number of cores to allocate for each task.	1	1, 3
70	`task.reaper.enabled`	Enables monitoring of killed / interrupted tasks.	false	true, false
71	`task.reaper.pollingInterval`	Controls the frequency at which executors will poll the status of killed tasks.	10s	5s, 10s, 20s
72	`task.reaper.threadDump`	Controls whether task thread dumps are logged during periodic polling of killed tasks.	true	true, false
73	`stage.maxConsecutiveAttempts`	Number of consecutive stage attempts allowed before a stage is aborted.	4	2, 4, 8
74	`spark.speculation`	If set to "true", performs speculative execution of tasks.	false	true, false
75	`reducer.maxSizeInFlight`	Maximum size of map outputs to fetch simultaneously from each reduce task.	48m	12m, 48m, 100m
76	`shuffle.service.index.cache.size`	Cache entries limited to the specified memory footprint.	100m	10m, 100m, 500m
77	`checkpoint.compress`	Whether to compress RDD checkpoints.	false	true, false
78	`io.compression.zstd.level`	Compression level for Zstd compression codec.	1	0, 1, 4
79	`io.compression.zstd.buffersize`	Buffer size in bytes used in Zstd compression, in the case when Zstd compression codec is used.	32k	8k, 32k, 128k
80	`kryo.referenceTracking`	Whether to track references to the same object when serializing data with Kryo.	true	true, false
81	`kryo.registrationrequired`	Whether to require registration with Kryo.	false	true, false

82	`serializer`	Class to use for serializing objects that will be sent over the network or need to be cached in serialized form.	Java Serializer	JavaSerializer, KryoSerializer
83	`broadcast.checksum`	Whether to enable checksum for broadcast.	true	true, false
84	`default.parallelism`	Default number of partitions in RDDs returned by transformations like join, reduceByKey, and parallelize when not set by user.	8	1, 8, 24
85	`files.useFetchCache`	If set to true, file fetching will use a local cache that is shared by executors that belong to the same application	true	true, false
86	`executor.metrics.pollingInterval`	How often to collect executor metrics.	0	0, 2s, 5s
87	`python.worker.memory`	Amount of memory to use per python worker process during aggregation.	512m	64m, 512m, 4g
88	`python.worker.reuse`	Reuse Python worker or not.	true, false	true, false
89	`scheduler .minRegisteredResourcesRatio`	The minimum ratio of registered resources to wait for before scheduling begins.	0.8	0.1, 0.4, 0.8
90	`scheduler.listenerbus.eventqueue. shared.capacity`	Capacity for shared event queue in Spark listener bus.	10000	1000, 10000, 50000
91	`scheduler.listenerbus.eventqueue. appStatus.capacity`	Capacity for appStatus event queue.	10000	1000, 10000, 50000
92	`scheduler.listenerbus.eventqueue. executorManagement.capacity`	Capacity for executorManagement event queue.	10000	1000, 10000, 50000
93	`scheduler.listenerbus.eventqueue. streams.capacity`	Capacity for streams queue in Spark listener bus.	10000	1000, 10000, 50000
94	`scheduler.listenerbus.eventqueue. eventLog.capacity`	Capacity for eventLog queue in Spark listener bus.	10000	1000, 10000, 50000
95	`executor.instances`	Number of executor instances to request from resource manager.	2	1,4,10,20
96	`dynamicAllocation. executorIdleTimeout`	If dynamic allocation is enabled and an executor has been idle for more than this duration, the executor will be removed.		10s, 60s, 240s
97	`dynamicAllocation.initialExecutors`	Initial number of executors to run if dynamic allocation is enabled.	0	0, 1, 4
98	`dynamicAllocation.minExecutors`	Lower bound for the number of executors if dynamic allocation is enabled.	0	0, 4, 10
99	`dynamicAllocation. executorAllocationRatio`	Allows to set a ratio that will be used to reduce the number of executors w.r.t. full parallelism.	1	0.1, 0.5, 1

100	`dynamicAllocation.shuffleTracking.enabled`	Enables shuffle file tracking for executors, which allows dynamic allocation without the need for an external shuffle service.	false	true, false
101	`executor.memory`	Amount of memory to use per executor process	1g	10g, 50g, 80g

Chapter 7.

Conclusions

In this chapter, we present the conclusions of this Ph.D. thesis. We summarize how the frameworks presented in this thesis advance beyond the state-of-the-art and we also discuss future extensions of the dissertation.

7.1. Thesis' Summary

Today, we stand at a pivotal moment in the creation and transformation of a digital world. Cloud computing is a vital pillar towards this transformation as it forms the de-facto execution model of modern applications, due to the efficiency, elasticity and cost-effectiveness it offers. From the cloud providers' point of view, minimizing the total cost of ownership (TCO) of cloud infrastructures without sacrificing the quality of services offered to clients is a top priority. At the same time, from the end-users' point of view the ultimate goal is to maximize performance of applications while reducing the operational pricing costs. Towards making out the most for both worlds, efficient management of Cloud resources is essential, in order to derive the possible maximum out of the available computational units. Resource efficiency can be achieved in many ways, i.e., by applying optimization from application-, to cluster-, to system-level. However, given the complexity as well as the huge number of available optimization knobs of modern Cloud datacenters, naive resource management policies are not able to provide optimal resource orchestration decisions. Towards building more efficient Cloud platforms, ML-driven resource management appears as a prominent solution, able to handle and manage the huge complexity of such systems. However, questions like "how", "when" and "where" it is better to integrate ML into cloud resource management are still vague.

In this thesis, we examined the application of deep-learning techniques for optimizing performance and resource efficiency of Cloud systems. We investigated the employment of deep neural networks on different optimization layers, ranging from monitoring, to system, to cluster and up to application level. Specifically, our work can be summarized as follows: ⋄ Regarding monitoring of Cloud systems, we developed *Rusty*, an interference-aware predictive monitoring framework for multi-tenant Cloud systems; ⋄ Regarding cluster-level management, we introduced *Adrias* a resource orchestration framework for memory-disaggregated Cloud systems. ⋄ Regarding application-level optimazation, we introduced *Sparkle*, an end-to-end, deep-learning driven parameter auto-tuning framework for Spark in-memory analytics.

Henceforth, we present a brief summary of our work and how the proposed frameworks tackle open-problems in the field.

Rusty *constitutes a predictive monitoring framework for multi-tenant servers systems.* Given the high unpredictability and dynamicity of modern Cloud infrastructures and applications, resource orchestrators and workload schedulers should be able to predict at runtime fluctuations regarding resource requirements of applications. Rusty forms a step towards this direction by providing a predictive monitoring framework for Cloud systems, able to dynamically forecast per-application performance characteristics under interference. Rusty employs Long Short-Term Memory networks (LSTM) networks, and leverages low-level system events to provide future predictions of per-application performance metrics. Our experimental evaluation has shown that Rusty can obtain extremely accurate predictions regarding performance (IPC and LLC misses) as well as energy consumption, thus, forming a really promising solution towards proactive resource allocators.

Adrias *forms a resource orchestration framework for memory-disaggregated Cloud systems.* With composable infrastructures forming the next big step regarding the design architecture of modern datacenters, novel orchestration frameworks are required able to optimally allocate disaggregated resources over the deployed applications. To the best of our knowledge, Adrias forms the first framework for interfence-aware application placement on memory disaggregated Cloud systems. Adrias leverages monitoring information from the underlying server and network interconnection, and employs deep-learning techniques to model performance of applications deployed on such infrastructures. Based on these predictions, it applies a naive scheduling logic in order to optimally place applications either on local or disaggregated memory, without sacrificing their performance. The experimental results show that Adrias is able to utilize remote memory efficiently, by identifying and placing the appropriate type of applications that are affected the least from disaggregated memory allocations.

Sparkle *forms an end-to-end, deep-learning driven parameter auto-tuning framework for Spark in-memory analytics.* The Spark in-memory analytics framework is gaining a lot of attention lately due to its enhanced performance compared to the traditional Hadoop. Spark offers more than 100 tunable parameters, which can be configured to change the behavior of the Spark engine, thus directly affecting the performance of deployed applications. Sparkle relies on deep learning techniques and low-level performance monitoring time-series to model performance of Spark deployments in an application agnostic manner. By employing genetic optimization, the framework efficiently traverses the search space online and recommends optimized deployment configurations. Experimental results show that *Sparkle* provides high accuracies for predicting the performance of SPARK applications, while it also offers high speedups compared to the default Spark configuration execution.

7.2. Future Extensions

The presented research work has showed that machine learning forms a very promising tool for optimizing resource efficiency inside datacenter infrastructures. However, given the rapid growth and expansion of next generation Clouds and the new requirements, both from a technical as well as a societal perspective, that constantly appear over the years, there are still open and novel issues to be addressed in future efforts. All the major Cloud providers, European networks of researchers (e.g., HiPEAC), as well as Europe itself, have already forged the path for the design of future computing systems. This path builds upon four major pillars, discussed below.

7.2.1. Sustainable and Green computing

Energy efficiency, climate neutrality and sustainability currently form top priorities for the entire globe. To meet these criteria, governments, businesses, and organizations are taking significant steps to comply with the societal requirements and environmental laws set by Europe and others. As we are heading towards digital societies, the total ICT contribution on the pollution and carbon emissions also increases[1], as the need for cloud and data processing infrastructures has multiplied with the advent of technologies like 5G, IoT, smart sensors, big data analysis, satellites, AI, deep learning, media and video streaming, and so on. In fact, from a European perspective, in 2018 the energy consumption of DCs was 76.8 TWh and is expected to rise by 30% in 2030.

As a result, energy-efficient Cloud computing is becoming a high priority both for EU and

[1]The Cloud Is Material: On the Environmental Impacts of Computation and Data Storage

US[2]. DCs have to become more energy efficient, reuse waste energy (e.g., heat), and use more renewable energy sources, with a view to becoming carbon-neutral in the near future. In this direction, proper management of computing resources inside the DC will become even more important to design more energy-proportional computing systems. In such systems, the single objective optimization commonly followed by modern Cloud systems (i.e., maximize performance of running applications) is transformed to a two (or multiple) objectives which have to be satisfied simultaneously (e.g., minimize power consumption while satisfying some minimum performance requirements). This optimization problem becomes even more difficult, if we consider that energy billing is gradually changing into a dynamic priced asset. Nevertheless, over the past few years, performance has largely driven research and development in large-scale Cloud systems, whereas increases in energy consumption have often been disregarded. Consequently, the design and development of novel, sophisticated Cloud orchestrators is required, able to optimize such conflicting requirements efficiently.

The frameworks proposed in this thesis can be extended accordingly to account for more sustainable and green computing. Specifically, *Rusty* framework already provides accurate predictions related to the energy consumed by servers under interference. These predictions can be leveraged by runtime controllers to drive pro-active, energy-efficient (rather than performance-driven) scheduling policies. For example, latest Intel processors allow for per-core DVFS, a technology that can be used along with Rusty's predictions to increase the energy proportionality of the server. Moreover, the monitoring mechanism and the scheduling logic of *Adrias* could be straightforwardly extended to support energy-aware application scheduling. By feeding additional, energy-related monitoring metrics to our DNN models, we could retrieve similar predictions, which can be included to the scheduling logic, to decide between the most energy-efficient memory mode. Last, *Sparkle*'s optimization approach can also be extended accordingly, by developing models for predicting the energy consumption of a Spark deployment for a given configuration and encapsulating the energy consumption parameter into the optimization goals of the genetic algorithm. Additional scheduling algorithms could be incorporated to all the frameworks, to also extend towards considering renewable energy sources and dynamic energy billing, e.g., perform time-shifted and geolocation-aware application deployment, where non-time-critical and geolocation-indifferent deployments will be postponed and/or executed in different locations to take advantage of green energy sources, reduced billing costs and others.

[2]Green cloud and green data centres

7.2.2. Hyperscale computing across the edge/cloud continuum

The huge amount of data generated by today's applications, has led to the inception of *edge computing* [10]. At its core, edge computing aims to bring computation closer to where data are generated, thus, reducing latency and bandwidth issues caused by moving all the data to the Cloud. Over the last years, the edge computing market has increased rapidly and was valued at USD 7.43 billion in 2021, while, it is expected to expand at a compound annual growth rate (CAGR) of 38.9% up until 2030[3]. While edge computing systems offer lower communication latency, their limited computing capability has led to the synergy of edge and Cloud, realizing the edge-cloud computing continuum. Going one step further, hyperscale edge computing systems are starting to appear on the foreground, where computation is distributed across the entire continuum and where computational resources can seamlessly scale up/down to accomodate fluctuating demand. On top of that, we observe an increasing adoption of multi-cloud and hybrid cloud strategies [382]. These strategies involve using a combination of different cloud providers and on-premises systems to create a more flexible and resilient IT infrastructure. This approach allows organizations to take advantage of the best features and services offered by different cloud providers while also maintaining control over their own data and applications. As a result, it is expected that multi-cloud and hybrid cloud solutions will become the norm in the future.

Despite its rapid growth and the tremendous advantages they offer, this combination of hyperscale, edge and multi-cloud computing systems pose a plethora of new challenges with respect to efficient orchestration of computing resources. One of the most important challenges is the extreme device heterogeneity introduced in such environments, where the available scheduling units (i.e., devices across the continuum) have diversified computational/memory power, runtime environments, network interfaces and others. Moreover, in hybrid cloud environments, scrutinizing all the available deployment options, both from performance- as well as cost-efficiency perspective adds an extra level of complication. In such environments the problem of resource orchestration becomes extremely challenging. Efficient scheduling schemes should take into account the different dynamics of the available computing resources, the relationship between where computation takes place and where data are stored (to minimize data movement overheads), while also application-level optimizations become even more important, to fully exploit the available computing power of low-end embedded devices.

Future extensions of our proposed frameworks to adapt to this new paradigm could be the following. Regarding *Rusty*, the core idea of predictive monitoring could be applied as is. Although, the monitored metrics considered by Rusty should be extended to also

[3]Edge Computing Market Size, Share & Trends Analysis Report

consider network traffic related events, whereas, for the training of the models we could also explore federated learning approaches across the continuum. For low-end devices, the overhead introduced by Rusty's LSTM model could be prohibitive. In such cases, Rusty can be extended to support remote API calls that communicate the monitored data, thus, offering the ability to offload the forecasting process to more powerful devices. *Adrias'* design already accounts for resource heterogeneity, since, local and remote memory can be considered as two heterogeneous resources with diverse computing capabilities. Thus, the problem of deploying applications in distributed computing resources can be directly mapped to the one examined in Chapter 5. Of course, Adrias' orchestration logic can be modified accordingly to also consider trade-offs between performance/cost between different Cloud offerings. With respect to scalability, Adrias is able to scale on multiple nodes by design, where the monitoring (and possibly prediction components) are distributed across the nodes of the cluster, whereas the orchestration logic could be easily centralized (e.g., integrated as a custom scheduler with Kubernetes). Last, *Sparkle*'s models should be adapted accordingly to account for heterogeneous computing nodes, as well as the presence of interference on the performance of applications.

7.2.3. More complex AI, use of specialized architectures and tighter HW&SW co-design

From a broader perspective, given the roadmaps described above, we foresee that the use of AI for resource management in the cloud is expected to become even more important in the future [383]. Regarding sustainable computing, AI-based resource management systems can play a key role by optimizing the use of resources to reduce energy consumption and carbon emissions, by analyzing patterns of usage and automatically adjust the power consumption of cloud resources based on demand. This can lead to significant reductions in energy consumption and costs, as well as a reduction in the environmental impact of cloud systems. Moreover, in the context of hyperscale edge computing, AI-based resource management systems can analyze patterns of usage and automatically allocate resources to meet the specific needs of edge devices and users, which can improve performance and reduce latency and cost savings through reduced resource waste. On top of the above, the potential end of Moore's law will pave the way for the use of more specialized architectures in the computing continuum [384]. Examples of this trend include field-programmable gate arrays (FPGAs), e.g., Microsoft Cloud uses FPGAs to accelerate Bing searches and other applications[4], smartphone technologies with dozens of specialized accelerators co-located on the same chip, hardware used in large data centers, such as Google's Tensor Processing Unit (TPU), and a vast array of other deep learning

[4]Microsoft's Bing search engine uses FPGA chips to provide more intelligent answers

accelerators both in the edge and the Cloud.

Taking these into account, we expect future Cloud resource orchestrators to rely on tighter software and hardware co-design for improving their performance and efficiency. By designing the software and hardware components together, it will be possible to optimize the overall system for specific use cases and to improve the performance of specific tasks, by analyzing usage time-series patterns and adjusting the configuration of both the HW (e.g., DVFS, CAT) and SW (e.g., SW thread scaling, sampling frequency) to meet the specific needs per application. However, there exist many challenges to realize the above. From a software standpoint, more expressive application descriptors and monitoring solutions must be developed (e.g., open standards for application monitoring and logging), which can be leveraged in a systematic way from AI-driven approaches. From a hardware point of view, we expect server systems and specialized accelerators to provide more detailed monitoring information, that can be leveraged by ML models to accomplish resource-management related tasks. Moreover, the rise of serverless architectures [33] and latest advancements towards microsecond scale application scheduling [385] demands for more granular monitoring hardware that can keep up with such requirements.

Overall, the frameworks proposed in this thesis, can be directly extended to analyze more expressive HW and SW monitoring events. First, the proposed deep learning models can become deeper, to be able to analyze more complex patterns within the provided input features. Moreover, ensembling techniques [386] and/or multi-branch DNN architectures can be considered that combine system-wise related monitoring information along with applications' runtime information (e.g., logging and application's metadata) and source code characteristics (e.g., using NLP techniques [387]).

Appendices

Appendix A.

Co-design Implications of Cost-effective On-demand Acceleration for Cloud Healthcare Analytics: The AEGLE Approach

Nowadays, big data and machine learning are transforming the way we realize and manage our data. Even though the healthcare domain has recognized big data analytics as a prominent candidate, it has not yet fully grasped their promising benefits that allow medical information to be converted to useful knowledge. In this chapter, we introduce AEGLE's H2020 EU project's big data infrastructure provided as a Platform as a Service. Utilizing the suite of genomic analytics from the Chronic Lymphocytic Leukaemia (CLL) use case, we show that on-demand acceleration is profitable w.r.t a pure software cloud-based solution. However, we further show that on-demand acceleration is not offered as a "free-lunch" and we provide an in-depth analysis and lessons learnt on the co-design implications to be carefully considered for enabling cost-effective acceleration at the cloud-level.

A.1. Introduction

At the centre of health debates there are open questions on how to manipulate, share and produce value out of data [388]. Even though the term Big-Data has become a buzzword in the field of information technology, its applicability on biological data is still limited. Modeling biological phenomena is typically very complex and has always been understood to be a computationally intensive process. Thus, In order to draw meaning from the exponentially increasing quantity of healthcare data, technologies capable of processing massive amounts of data efficiently and securely have to be adopted.

Collecting and aggregating anonymous data from geographically dispersed locations makes it possible to construct statistically meaningful databases, based on which, macroscopic reasoning can be made, rather than solely focusing on the individual and associated pathology. Answers to these questions will create opportunities to predict long-term health conditions and identify non-traditional intervention points, as well as to design better diagnostics tools, prevent diseases, increase access to and reduce the costs of healthcare [389]. As discussed in [390], effective use of data in the US health sector could generate USD 300 billion in value per year. The implementation of big data analytics in the healthcare sector has the potential to boost the integration of user-generated data with official medical data, leading to healthcare that is more integrated and personalised [391]. Several European initiatives [1] have already pinpointed the importance and usefulness of health-care big data, e.g. to predict the outbreak of an epidemic. To that end, business interest is growing, like in the case of Open Data initiative, where big data health providers, research institutes and industry aim to develop a vendor-neutral Big-Data platform [2].

The use of available medical data can allow clinicians to simulate potential outcomes and thus prevent patients from undergoing ineffective treatments or provide better treatment plans. In other words, accumulating data to develop a greater understanding of pathophysiological processes will result in significant healthcare improvements. However, the strategic advantage brought by Big-Data in healthcare still materializes at slow paces, as only some large-scale organizations have established few pilot or proof-of-concept projects.

Data-driven services are still needed to cater for the data versatility, volume, velocity and veracity within the whole data value chain of healthcare analytics. Currently, none of the

[1]Big Data Analytics: What it is and why it matters

[2]Top Healthcare Data Startups

existing Big-Data EU projects are completely dedicated to healthcare and the provision of corresponding services, or the management of diseases. The AEGLE project aspired to bridge this gap, by implementing a full data value chain to create new value out of rich, multi-diverse health data with the goal to revolutionize integrated and personalized healthcare services.

The project builds upon the synergy of cloud technologies together with heterogeneous high performance reconfigurable acceleration for delivering optimized analytic services on Big-Bio Data applications. At local level, the data are anonymized and uploaded to the cloud. At cloud level, the framework consists of the frontend and the backend part. On the frontend, an advanced visualization service and a friendly user interface simplify the data visualization and the execution of complex analytics workflows. On the other hand, the cloud-backend is the core of AEGLE's big data framework, as it is responsible for data storage and efficient execution of conventional and accelerated workflows. AEGLE's modular deployment envisions to support and enable a healthcare oriented analytic design framework whose functionalities and services are not provided solely by a unique user interface. Users of the framework can develop their own user interfaces at the local level and interconnect them with the services and tools provided at the Cloud level.

The rest of the chapter is organized as follows. In section A.2, we present in brief AEGLE's targeted use-cases and we give detailed information regarding the overall architecture of AEGLE framework and its innovative on-demand acceleration mechanism. In section A.3 we analyze the essential costs for the operation of the platform. Finally, section A.4 explores the acceleration capabilities of CLL workloads as well as the co-design implications for cost-effective acceleration at the cloud, while section A.5 concludes the chapter.

A.2. The AEGLE cloud infrastructure

AEGLE cloud infrastructure hosts data and analytic tools from three distinct use cases, *i*) Chronic Lymphocytic Leukemia (CLL), *ii*) Type II diabetes (T2D) and *iii*) Intensive Care Unit (ICU). We utilize CLL as our driver case to highlight the technical contribution, due to the high workload divergence of CLL workflows, i.e. large scale descriptive statistics, machine learning and genomic pipelines, that make a strong claim on the utilization of the on-demand acceleration features.

Fig. A.2.1 depicts an overview of the architecture of the AEGLE platform. On the bottom

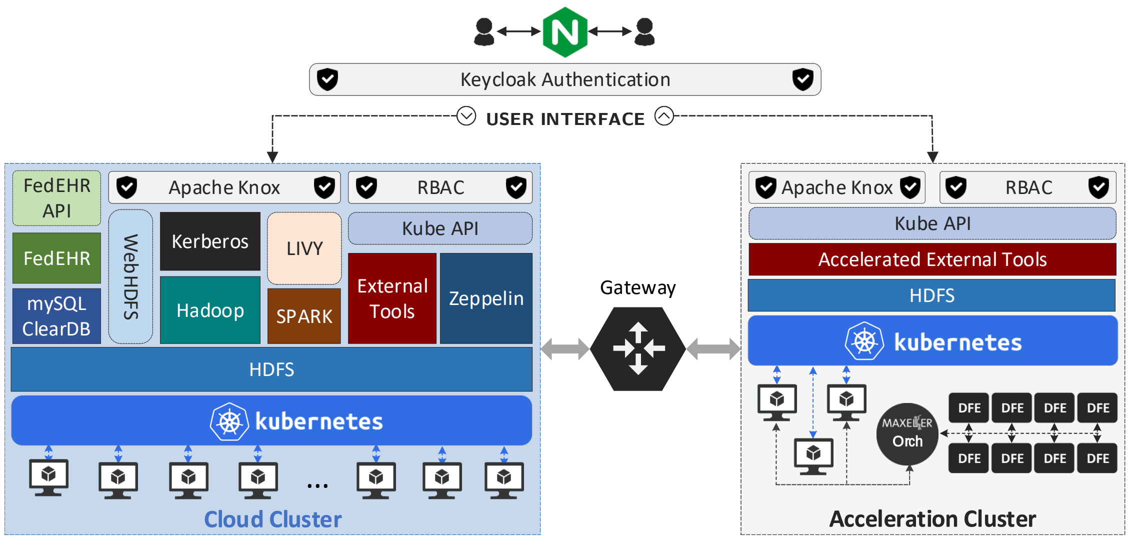


Figure A.2.1.. AEGLE's Big Data Framework hierarchy.

of the pyramid, lies the hardware stack. The hierarchical design of AEGLE allows the accommodation of any type of devices, i.e., bare-metal servers, virtual machines and accelerator devices. In addition, it allows the extension of its hardware stack, enabling new accelerators and/or entire acceleration clusters to be transparently federated in the existing infrastructure. Currently, AEGLE's hardware stack consists of two clusters, a *conventional* cluster comprised of ordinary virtual machines and an *accelerated* cluster that pairs virtual machines with Maxeler MPC-X servers providing dataflow acceleration of specific applications.

AEGLE platform exposes two kinds of services, *static* and *dynamic* ones. Static services are always up and running and constitute the backbone of the framework, as they are responsible for authentication of users, persistent storage of data, and mainly for the interaction of users with the framework through the user interface. On the other hand, dynamic services are operations that the users of AEGLE perform on the platform, such as workflows execution or dataset visualization.

Resource management: To efficiently manage the deployment of applications on this large and heterogeneous pool of hardware resources, Kubernetes [175] is adopted as the core resource manager, providing automated deployment of containerized applications. Kubernetes is responsible for scheduling workloads on the appropriate cluster (conventional or accelerated), as well as managing resources among VMs. The accelerated cluster supports multiple MPC-X nodes, which implicates a large pool of resources that could potentially be accessed by multiple independent applications at the same time. This requires some form of cluster level resource management for the acceleration engines, i.e. FPGAs, which is provided by the Maxeler Orchestrator that maintains the availability status of each FPGA. Applications are able to request Data Flow Engine (DFE) resources from a centralized management facility (i.e. the orchestrator) that can allocate, schedule and manage resources in the system.

Security: At the core of the identity management solution lies Keycloak [392]. Keycloak is utilized for modelling the overall security information structure including user roles, allowed actions and group memberships. Due to the heterogeneous security requirements of the different high level components and technologies adopted by the AEGLE platform, Keycloak is framed with Apache Knox [393], which serves as an authorization gateway to the Hadoop services ecosystem. Knox is used as a single point of access for the Hadoop services and also acts as a secondary permissions layer, thus, extending the reach of Apache Hadoop services to users outside of a Hadoop cluster without compromising Hadoop's secure mode. Moreover, Kerberos lies security wise in between the Hadoop cluster and the Knox gateway. When a user needs to perform an action with a Hadoop service, Knox receives an access token from the incoming requests, validates their authenticity and then authenticates itself to Hadoop via Kerberos. Finally, execution of the external tools and visualizations are managed by Kubernetes itself. Role based Access Control mechanism (RBAC) is configured and activated on Kubernetes, so that assignment of user roles and check for permissions can be performed; users can deploy their own external tool pods based on their Keycloak tokens, as RBAC is integrated with Keycloak using an OpenID Connect (OIDC) client.

Workloads: AEGLE provides a plethora of tools for analyzing and examining healthcare data. Within AEGLE, more than 100 analytics have been developed, including analytics for healthcare data and predictive models [394]. The *de-novo* analytics are developed over state-of-the-art Big Data platforms, e.g. Hadoop [141], Spark [93], allowing fast and general-purpose computations, interactive queries and stream processing. Additionally, AEGLE supports a variety of existing analytics, tools, and execution engines, that are widely used in the bio-medical research domain, e.g. SeqMule [395], TopHat [396], which provide genomics analyses, such as DNA or RNA sequencing. These tools are deployed and managed as containerized applications by Kubernetes. Last, to simplify the data representation produced by the execution of complex workflows, the platform provides advanced visualization techniques by utilizing the Apache Zeppelin framework [397].

A.3. Cost model of AEGLE Cloud Infrastructure

AEGLE deployment is amenable to the complex cloud cost models. In order to efficiently exploit the advanced on-demand acceleration features of AEGLE platform, a detailed and realistic assessment of cost w.r.t to performance is needed.

A.3.1. Cost model of AEGLE's PaaS

AEGLE's software components are based on open source solutions. Apache licensing 2.0 conditions enable the free commercial use of this open source software without further costs. AEGLE software platform is only amenable to licensing conditions for the GNUBILA's FedEHR Capsule software that provides data anonymization, upload and repository services

The big data platform of AEGLE, has been deployed on Microsoft's Azure Cloud platform [177], utilizing the Azure Kubernetes Service (AKS). The total platform cost (TC) per month can be calculated as follows:

$$TC = C_M + C_{SS} + C_S + C_{LIC} + C_{DS} + C_{HW} + C_{BW} \tag{A.1}$$

where C_M is the cost for the Virtual Machines used for the deployment of the master nodes of Kubernetes cluster, C_{SS} is the cost for the Virtual Machines hosting the required static services, C_{DS} is the cost for the Virtual Machines hosting dynamic services, C_S is the cost for storage purposes, C_{BW} is the cost for bandwidth transfer inside and outside the cloud cluster, C_{HW} is the cost for hardware accelerators and C_{LIC} is the cost for the software licenses (FedEHR capsule).

Kubernetes master nodes: The master nodes of the cluster should operate indefinitely (24/7), as they are the backbone of Kubernetes ensuring the fluid operation of the cluster. For reliability and fault tolerance reasons, 3 Master nodes were deployed, however the number of master nodes may vary. As a managed Kubernetes service, AKS is free, so the master nodes of the cluster are free of charge ($C_M = 0$).

Static services: The static services of AEGLE comprise of the user interface, the security services, the FedEHR anonymizer and the HDFS cluster, which should be al-

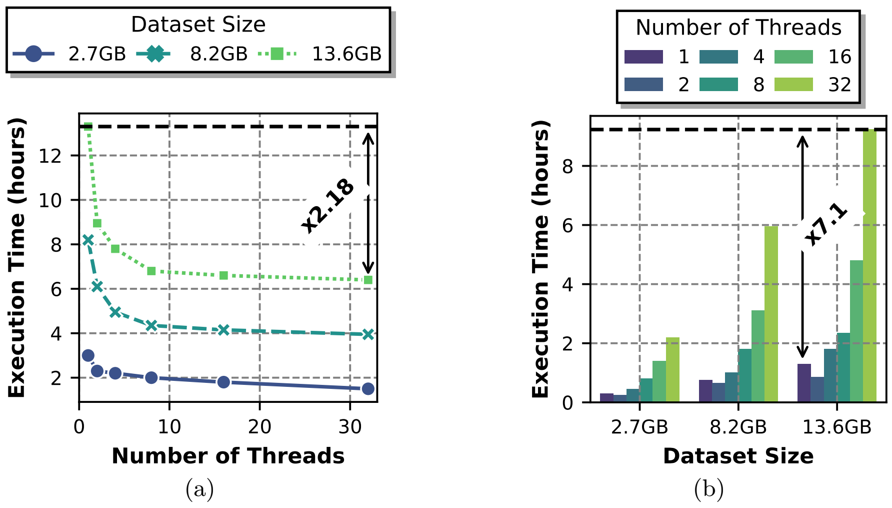


Figure A.3.1.. Execution time and total cost for different number of threads and input datasets for the SeqMule workload.

ways up and running. In order to provide a high availability and fault tolerant cluster, Kubernetes is configured to schedule HDFS nodes to different VMs. Under normal traffic conditions on the platform, 3 datanodes and 1 namenode are utilized. For the static services, we have chosen the L4 machines with a cost of \$0.372/hour (C_{SS} = \$1071.36/month).

Storage and Traffic: Storage costs depend on the amount of data hosted on the platform. Azure charges \$60 per 1TB of File Disk per month. Within AEGLE we host approximately 1TB of anonymized medical data and provide a replication factor of 3 on the HDFS side. Therefore, we utilize 1TB of persistent disk for each HDFS node (C_S = \$240/month). Moreover, the outbound traffic is charged for \$0.07 per GB.

In total, the expenses for the basic operation of the platform are estimated to TC = \$2900.34. This cost might seem high at a first glance, but the platform can provide services to tens of patients, lowering the individual costs to lower levels.

A.3.2. Cost-aware acceleration services

Although acceleration services are available on demand, they should be used cautiously. The tradeoff between speedup gains and cost increment should always be taken into account, even when deciding whether to scale on pure software (i.e., number of threads) or not. For example, Fig. A.3.1 shows the scaling of cost and execution time, w.r.t. the number of threads and input dataset size for the SeqMule pipeline implementing full exome sequencing. As depicted, cost growth escalates quickly compared to savings in execution time. Even for the largest dataset sizes, where we observe better scaling in terms of threads (x2.18 for 32 threads), the increase in cost is greater than time gains by a factor of three (x7.1). This shows that utilizing resources recklessly should be avoided, especially when aiming to keep costs at low levels.

The aforementioned scenario is becoming even more complex, when making use of the acceleration cluster. Acceleration value depends on whether the gained speed-up makes up for traffic charges and the increased cost of occupying an acceleration machine rather than a conventional one. Equation A.2 provides a lower bound on the acceleration speedup,x, required in order to have cost-effective acceleration services:

$$\frac{t_{CPU}}{x} * C_{HW} + C_{BW} \leq t_{CPU} * C_{VM} \tag{A.2}$$

where t_{CPU} is the execution time (hours) of a workload on a conventional VM, i.e. VM without having access to acceleration cluster, and C_{VM} is the cost of the VM. The above equation implies a fundamental principle for cost effective on-demand acceleration, i.e. the gained acceleration speedup should encompass the increased costs of dedicated HW resources, e.g. FPGAs, as well as the data transfer costs from/to the cloud storage. As shown in the next section, this fundamental principle directly questions the currently dominating model of kernel-specific cloud acceleration libraries/stores and suggests for more holistic HW/SW co-designed solutions.

A.4. Cost-effective acceleration of AEGLE's CLL Use Case

Predictive and descriptive CLL analytics are exhibiting low execution runtimes, violating Eq. A.2 bounds, e.g. transferring the input data to the accelerated cluster would take up as much time as pure software execution. On the contrary, SeqMule forms a good candidate for acceleration due to its intense time requirements (8-12 hours for realistic inputs). SeqMule performs automated human exome/genome variants detection, where short fragments of DNA reads are first aligned to a genome reference and then used for variant calling. AEGLE includes two major types of genome analysis in its workflows. The first one includes a single aligner (BWAMEM/Bowtie2) operating on one input dataset, followed by multiple variant callers (GATKLite, SamTools, Freebayes, Varscan). The second type performs alignment on two input datasets (e.g pre- and after-treatment read sequences) and a single variant caller (Varscan) is utilized. Fig. A.4.2

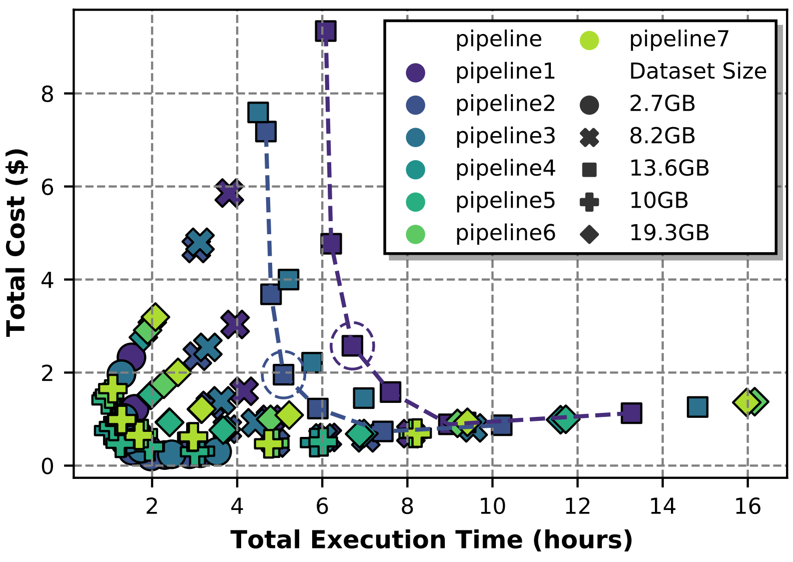


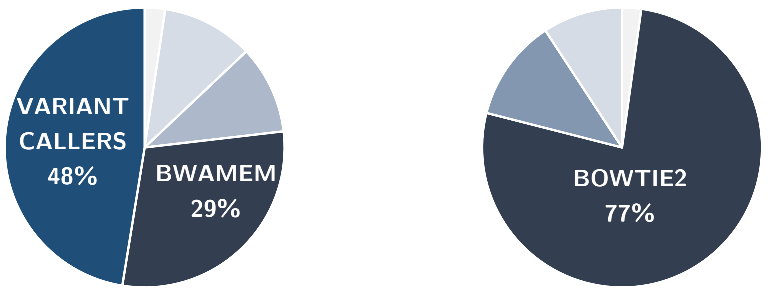
Figure A.4.1.. Total cost and time for software execution of Seqmule pipelines for different input datasets and number of threads.

illustrates the distribution of time per stage for each of the examined Seqmule analyses. Aligners and variant callers take up most of the computation time and thus are usually the target for acceleration. Still, the overall time of minor stages is not negligible either.

A.4.1. Co-design implications of genomic workflows acceleration

Careful consideration is required to decide if it is cost-efficient for users to execute a SeqMule pipeline on the accelerated cluster. For that purpose, a small exploration of the available options is presented. Each pipeline is executed for different input sizes and different number of threads (1-2-4-8-16-32) and the resulting set of configurations is depicted in Fig. A.4.1. As illustrated with dotted lines, configurations of the same pipeline and input size lie on the same frontier, where lower execution times correspond to higher number of threads. Executing the specific configuration on the accelerated cluster should be more efficient than the point of the frontier with the best trade-off between cost and latency (highlighted points).

After identifying the best cost-latency trade-off points, we apply Eq.A.2 to acquire the lower acceleration factor x for each pipeline. Given this value and the profiling results of Fig. A.4.2, we investigate how this speedup can be actually acquired. We form our



(a) Normal Analysis (b) Somatic Analysis

Figure A.4.2.. Execution's time distribution for SeqMule pipelines.

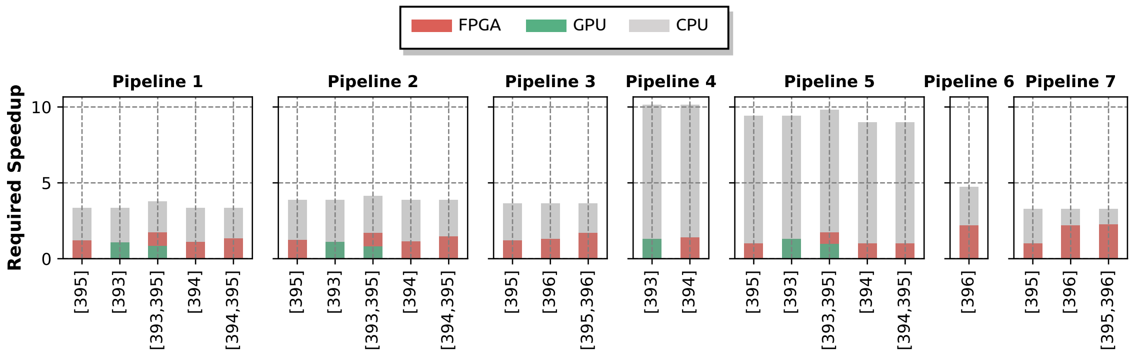


Figure A.4.3.. Total cost and time for software execution of Seqmule pipelines for different input datasets and number of threads.

acceleration library adopting state-of-the-art hardware accelerators for SeqMule aligners and variant callers utilized in these pipelines. To broaden the analysis, we considered accelerators targetting both GPUs and Xilinx FPGAs, as well as Maxeler DFEs [398–401]. For each pipeline, we consider all combinations of hardware accelerators from our library and measure the resulting speedup and cost.

Figure A.4.3 presents the results for all pipelines for a single dataset input size (8.2GB). Each bar stands for a threshold speedup, from which value and onwards hardware acceleration is more cost efficient. Figure A.4.3 shows that the need for an holistic co-design solution is evident in the case of cloud level acceleration, since none of the examined accelerated pipelines achieves the cost-effectiveness threshold. Configurations for pipelines 1,2,3,6,7 partly achieve the target speedup value by relying on FPGAs, GPUs or both. The remaining speedup can be acquired by allocating more software resources in order to boost the performance of the pipeline stages that are still executed on software. That is not the case for pipelines 4 to 5, whose hardware-acquired speedup is far below the threshold, making it potentially hard for software resources to compensate for the difference.

The difficulty to acquire the target speedup can be attributed to the fact that the accelerated solutions are applied only to some stages of the pipeline, directly originated from the current cloud model of kernel-specific acceleration libraries. To add to that, although there are many hardware accelerators for the bottlenecks of these tools, there are only a few integrated co-designed implementations that manage to deliver a maximum of only ×2 speedup, i.e. below the cost-effectiveness acceleration threshold.

A.5. Conclusion

The scope of this work is to present H2020 AEGLE's Platform as a Service, highlighting the on-demand acceleration services that are offered through a framework that seamlessly combines cloud and big data technologies. A detailed description of the cost model of AEGLE platform is provided, along with an exploration that aims to export preliminary guidelines for enabling cost-effective acceleration on demand. Utilizing CLL use case as a driver application on this exploration, we present the challenges of effective co-design and highlight the importance of providing an elegant and effective solution for Big Data Health analytics.

Appendix B.

Resource Aware GPU Scheduling in Kubernetes Infrastructure

Nowadays, there is an ever-increasing number of artificial intelligence inference workloads pushed and executed on the cloud. To effectively serve and manage the computational demands, data center operators have provisioned their infrastructures with accelerators. Specifically for GPUs, support for efficient management lacks, as state-of-the-art schedulers and orchestrators, threat GPUs only as typical compute resources ignoring their unique characteristics and application properties. This phenomenon combined with the GPU over-provisioning problem leads to severe resource under-utilization. Even though prior work has addressed this problem by colocating applications into a single accelerator device, its resource agnostic nature does not manage to face the resource under-utilization and quality of service violations especially for latency critical applications. In this chapter, we design a resource aware GPU scheduling framework, able to efficiently colocate applications on the same GPU accelerator card. We integrate our solution with Kubernetes, one of the most widely used cloud orchestration frameworks. We show that our scheduler can achieve ***58.8%*** *lower end-to-end job execution time 99%-ile, while delivering* ***52.5%*** *higher GPU memory usage,* ***105.9%*** *higher GPU utilization percentage on average and* ***44.4%*** *lower energy consumption on average, compared to the state-of-the-art schedulers, for a variety of ML representative workloads.*

B.1. Introduction

In recent years, the adoption of artificial intelligence (AI) and machine learning (ML) applications is increasing rapidly. Several major Internet service companies including Google, Microsoft, Apple and Baidu have observed this trend and released their own intelligent personal assistant (IPA) services, e.g. Siri, Cortana etc., providing a wide range of features. Compared to traditional cloud applications such as web-search, IPA applications are significantly more computationally demanding [402]. Accelerators, such as GPUs, FPGAs, TPUs and ASICs, have been shown to be particularly suitable for these applications from both performance and total cost of ownership (TCO) perspectives [402]. With the increase in ML training and inference workloads [64, 402], cloud providers begin to leverage accelerators in their infrastructures, to catch up with the workload performance demands. This trend is also evident as Amazon AWS and Microsoft Azure have started offering GPU and FPGA based infrastructure solutions.

In particular, for the case of ML inference oriented tasks, public clouds have provisioned GPU resources at the scale of thousands of nodes in data-centers [403]. Since GPUs are relatively new to the cloud stack, support for efficient management lacks. State-of-the-art cluster resource orchestrators, like Kubernetes [404], treat GPUs only as a typical compute resource, thus ignoring their unique characteristics and application properties. In addition, it is observed that users tend to request more GPU resources than needed [405]. This tendency is also evident in state-of-the-art frameworks like Tensorflow which by default binds the whole card memory to an application. This problem, also known as *over-provisioning*, combined with the resource agnostic scheduling frameworks lead to under-utilization of the GPU-acceleration infrastructure and, thus, quality of service (QoS) violations for latency critical applications such as ML inference engines. To overcome the aforementioned issues, real-time monitoring, dynamic resource provisioning and prediction of the future status of the system is required, to enable the efficient utilization of the underlying hardware infrastructure by guiding the GPU scheduling mechanisms.

In this chapter, we propose a novel GPU resource orchestration framework that utilizes real-time GPU metrics monitoring to assess the real GPU resource needs of applications at runtime and based on the current state of a specified card decide whether two or more application can be colocated. We analyze the inherent inefficiencies of state-of-the-art Kubernetes GPU schedulers concerning the QoS and resource utilization. The proposed framework estimates the real memory usage of a specified card and predicts the future memory usage, enabling better inference engine colocation decisions. We show that our scheduler can achieve **58.8%** lower end-to-end job execution time 99%-ile for the majority of used inference engine workloads, while also providing **52.5%**

higher GPU memory usage, **105.9%** GPU utilization percentage average and **44.4%** lower energy consumption compared with the Alibaba GPU sharing scheduler extension.

B.2. Related Work

The continuous increment in the amount of containerized workloads uploaded and executed on the cloud, has revealed challenges concerning the container orchestration. Workload co-location and multi-tenancy exposed the interference agnostic nature of the state-of-the-art schedulers [166] while the integration of accelerator resources for ML applications revealed their resource unawareness [403]. To enable better scheduling decisions, real-time [114] or even predictive [313] monitoring is required to drive the orchestration mechanisms. Extending to the case of GPU accelerators, real-time GPU monitoring can allow the colocation of containers on the accelerator in a conservative manner to avoid out-of-memory issues [403].

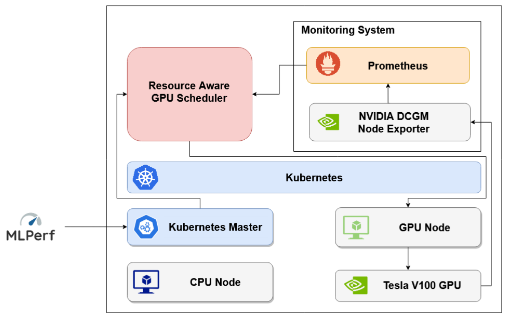
Container orchestration on GPU resources has been in the center of attention of both academia and industry. Throughout the years, various GPU scheduling approaches have been proposed. Ukidave et al. [406] and Chen et al. [407] have proposed GPU runtime mechanisms to enable better scheduling of GPU tasks either by predicting task behavior or reordering queued tasks. More recent works [408, 409] have introduced docker-level container sharing solutions by allowing multiple containers to fit in the same GPU, as long as the active working set size of all the containers is within the GPU physical memory capacity. As distributed deep neural network (DNN) training based applications have started taking advantage of multiple GPUs in a cluster, the research community proposed application specific schedulers [410] that focus on prioritizing the GPU tasks that are critical for the DNN model accuracy. Hardware support for GPU virtualization and preemption were also introduced. Gupta et al. [411] implemented a task queue in the hypervisor to allow virtualization and preemption of GPU tasks while Tanasic et al. [412] proposed a technique that improves the performance of high priority processes by enabling GPU preemptive scheduling. The integration of GPU sharing schemes on GPU provisioned cloud infrastructures managed by Kubernetes is a trend that is also observed. Yeh et al. proposed KubeShare [413], a framework that extends Kubernetes to enable GPU sharing with fine-grained allocation, while Wang et al. [414] introduced a scheduling scheme that leverages training job progress information to determine the most efficient allocation and reallocation of GPUs for incoming and running jobs at any time.

Regarding container orchestration within GPU environments, Kubernetes itself includes experimental support for managing AMD and Nvidia GPUs across several nodes. Kubernetes GPU scheduler extension [415] exposes a card as a whole meaning that a container can request one or more GPUs. Even though this implementation does not provide fractional GPU usage, it allows better isolation and ensures that applications using a GPU are not affected by others. To overcome this problem, the authors in [416] proposed a GPU sharing scheduling solution which relies on the existing working mechanism of Kubernetes. Alibaba GPU sharing extension aims to improve the utilization of GPU resources by exposing the memory of a card as a custom Kubernetes resource, thus, allowing containers to specify their required amount of memory. Even though this approach allows the concurrent execution of multiple containers, its resource agnostic nature makes it dependable on the credibility of the memory requests. Kube-Knots [403] overcomes this limitation by providing a GPU-aware resource orchestration layer that addresses the GPU orchestration problem. Kube-Knots dynamically harvests spare compute cycles by enabling the co-location of latency-critical and batch workloads, thus, improving the overall resource utilization. This way, it manages to reduce QoS violations of latency critical workloads, while also improving the energy consumption of the cluster. However, its predictive nature fails to face the problem of container failures due to incorrect memory usage predictions and thus GPU memory starvation.

B.3. Experimental Setup & Specifications

We target high-end server systems equipped with GPU acceleration capabilities found under today's data-center environments. Specifically, our work targets an ML-inference cluster, where a GPU-equipped node is responsible for serving the computational demands of inference queries effectively. In the proposed framework, whenever an inference engine arrives on the cluster, the Kubernetes master redirects it to our custom resource aware GPU scheduler. By leveraging real-time GPU monitoring and prediction, our scheduler decides whether to schedule it on the GPU, or enqueue the task on a priority queue and delay the execution until there are enough GPU resources available. Figure B.3.1 shows an overview of our experimental setup.

Hardware Infrastructure Characterization: All of our experiments have been performed on a dual-socketed Intel® Xeon® Gold 6138 server equipped with an NVIDIA V100 GPU accelerator card, the specifications of which are shown in Table B.3.1. On top of the physical machine we have deployed three virtual machines, which serve as the nodes of our cluster, using KVM as our hypervisor. The V100 accelerator is exposed on the inference-server VM (24 vCPUs, 32GB RAM) using the IOMMU kernel config-



(a) Proposed Scheduling Framework

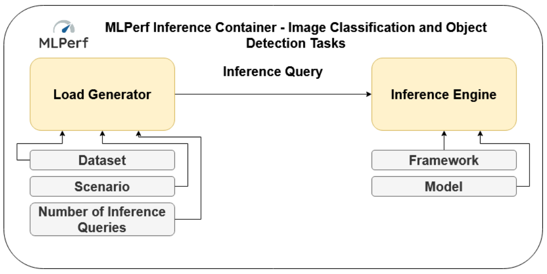

MLPerf
MLPerf Inference Container - Image Classification and Object Detection Tasks
Load Generator
Inference Query
Inference Engine
Dataset
Scenario
Number of Inference Queries
Framework
Model
(b) MLPerf Inference Engine Architecture

Figure B.3.1.. Proposed scheduling framework and MLPerf inference engine architecture

uration, while the rest of the VMs (8 vCPUs, 8GB RAM each) are utilized to deploy critical components of our system, such as the master of our Kubernetes cluster and our monitoring infrastructure.

Software & Monitoring Infrastructure Characterization: On top of the VMs, we deploy Kubernetes container orchestrator (v1.18) combined with Docker (v19.03) which is nowadays the most common way of deploying cloud clusters at scale [417]. Our monitoring system consists of two major components, NVIDIA's Data-Center GPU Manager exporter (DCGM) [418] along with Prometheus [183] monitoring toolkit. DCGM exports GPU metrics related to the frame buffer (FB) memory usage (in MiB), the GPU utilization (%) and the power draw (in Watts). In particular, a DCGM exporter container is deployed on top of each node of the cluster through Kubernetes. This container is responsible for capturing and storing the aforementioned metrics into our Prometheus time-series database every specified interval. We set the monitoring interval equal to 1 second to be able to capture the state of our underlying system at run-time. Finally, metrics stored in the Prometheus time-series are accessed from our custom Kubernetes

Intel® Xeon® Gold 6138	
Cores/Threads	20/40
Sockets	2
Base Frequency	2.0 GHz
Memory (MHz)	132 GB (2666)
Hard Drive	1 TB SSD
OS (kernel)	Ubuntu 18 (4.15)

NVIDIA V100	
Architecture	Volta
Comp. Cap.	7.0
CUDA Cores	5120
Memory Size	32 GB HBM2
Interface	PCIe 3.0 x16
Sched. Policy	Preemptive

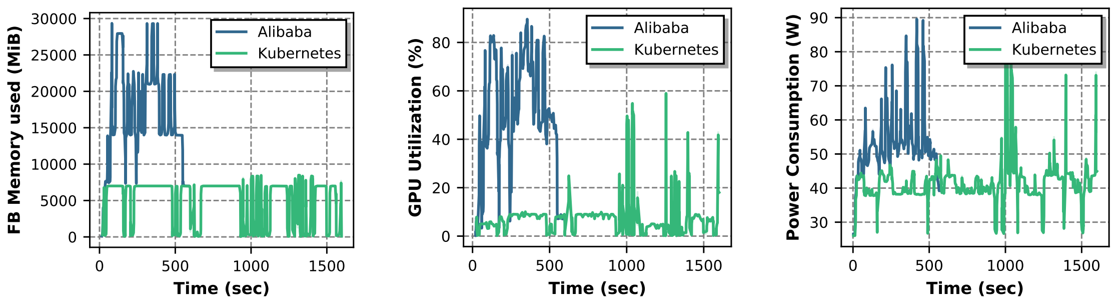
Table B.3.1.: CPU & GPU Specifications

scheduler by performing Prometheus-specific PromQL queries, as described in section B.5.

Inference Engine Workloads: For the rest of the chapter, we utilize MLPerf Inference [419] benchmark suite for all of our experiments, which is a set of deep learning workloads performing object detection and image classification tasks. As shown in Figure B.3.1b, each MLPerf Inference container instance consists of two main components, *i)* the Inference Engine and *ii)* the Load Generator. The Inference Engine component is responsible for performing the detection and classification tasks. It receives as input the pre-trained DNN model used during inference (e.g. ResNet, Mobilenet etc.) as well as the corresponding backend framework (e.g. PyTorch, Tensorflow etc.). The Load Generator module is responsible for producing traffic on the Inference Engine and measure its performance. It receives as input the validation dataset (e.g. Imagenet, Coco) as well as the examined scenario and the number of inference queries to be performed. The scenario can be either Single stream (Load Generator sends the next query as soon as the previous is completed), Multiple stream (Load Generator sends a new query after a specified amount of time if the prior query has been completed, otherwise the new query is dropped and is counted as an overtime query), Server (Load Generator sends new queries according to a Poisson distribution) and Offline (Load Generator sends all the queries at start). Considering the above inputs, the Load Generator performs streaming queries to the Inference Engine and waits for the results. For the rest of this work, we utilize the Single Stream scenario and evaluate our inference engine through the 99%-ile of the measured latency.

B.4. Motivational Observations and Analysis

Latest advancements in the micro-architecture of NVIDIA's GPUs allow the transparent, cooperative execution of CUDA applications on the underlying accelerator, either through CUDA's streams [420] or through CUDA's Multi-Process Service (MPS) [421] capabilities. These functionalities increase the utilization of GPU accelerators, thus, offering increased computing capacity, yet, state-of-the-art frameworks, such as Kubernetes do not provide mechanisms that expose them to end-users. In fact, Kubernetes default GPU scheduler [415] mechanism provides exclusive access to applications requesting GPU accelerators. Even though, this approach allows isolation and ensures that applications using a GPU do not interfere with each other, it can cause high resource under-utilization or QoS violations, especially in deep-learning inference scenarios on high-end GPUs, which have low requirements in terms of CUDA cores and memory. In order to allow more prediction services to share the same GPU and, thus, improve their QoS and the utilization



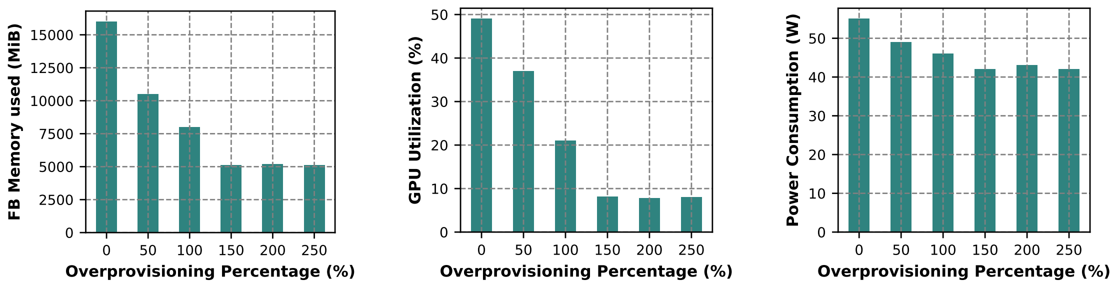
(a) GPU Memory Usage (b) GPU Utilization Percentage (c) GPU Power Usage

Figure B.4.1.. GPU memory usage, utilization percentage and power usage signals for Kubernetes GPU scheduler extension and Alibaba GPU sharing extension.

of the card, partitioning of the GPU memory resource is required. Towards this direction, Alibaba offers a GPU sharing extension [416], which allows the partitioning of the GPU memory. This scheduler allows end-users to define the requirements of their workloads in terms of GPU memory and combines this information with the total available memory of the GPU to decide whether two or more inference engines can be colocated or not.

To demonstrate the inefficiency of the Kubernetes GPU scheduler extension compared with Alibaba GPU sharing extension, we perform a straight comparison between them for the scheduling of a workload that consists of 6 inference engines from the MLPerf suite.

Figure B.4.1 shows the GPU memory utilization (MB), the CUDA cores utilization (%) and the power usage signals of the inference engine workload for the above-mentioned schedulers. As shown, the Kubernetes GPU scheduler extension has an average memory utilization of 5GB, which can be considered relatively low compared to the available 32GB memory of the underlying GPU card. The same observation can be made for the GPU utilization signal (7.22% on average) and the power consumption (41.5 Watts on average), as the GPU binding per inference engine leads to resource under-utilization. On the other hand, the average GPU memory usage for the Alibaba GPU sharing extension is 16GB, which is x3.24 higher. Similarly, we see an average of 49% utilization improvement (x6.8 increment) and an average of 52.9 Watts higher power consumption (x1.28 increase). It also leads to a 52.8% decrease of the average energy consumption as Kubernetes GPU scheduler extension consumption is 66.4 kJ and Alibaba GPU sharing extension consumption is 31.3 kJ. Finally, we observe that the overall inference engine workload duration using the Alibaba GPU sharing extension is x2.67 faster than the Ku-



(a) Memory Usage Average (b) GPU Utilization Average (c) Power Usage Average

Figure B.4.2.. Memory usage, GPU utilization and power consumption averages vs over-provisioning percentage for Alibaba GPU sharing scheduler extension.

bernetes GPU scheduler extension, meaning that the card sharing improves the overall workload duration.

Even though Alibaba's scheduler outperforms the default one, it highly depends on the provisioning degree of the inference engine memory request. For example, if an inference engine requests more memory than it actually needs, this may affect future GPU requests of other inference engines, which will not be colocated, even though their memory request can be satisfied. To better understand the impact of the resource over-provisioning problem within Alibaba's scheduler, we perform 6 different experiments, where we schedule the same inference-engine task, each time with a different memory over-provisioning percentage, ranging from 0% to 250%. Figure B.4.2 depicts the memory usage, the utilization percentage and the power usage averages. For low over-provisioning percentages, Alibaba GPU sharing extension leads to high resource utilization due to the inference engine colocation. However, as shown, it is not able to efficiently sense and handle user-guided over-provisioning scenarios.

B.5. Resource-aware GPU Sharing Kubernetes scheduler

Figure B.5.1 shows an overview of our proposed resource-aware GPU-sharing scheduler. Whenever an inference engine is scheduled from the custom scheduler, the corresponding workload enters a priority queue which defines their scheduling order. The inference engine assigned priority is proportional to the corresponding GPU memory request. As a result the scheduler always tries to schedule the inference engines with the bigger memory requests. If a workload is chosen to be scheduled, the following three co-location mechanisms are successively executed:

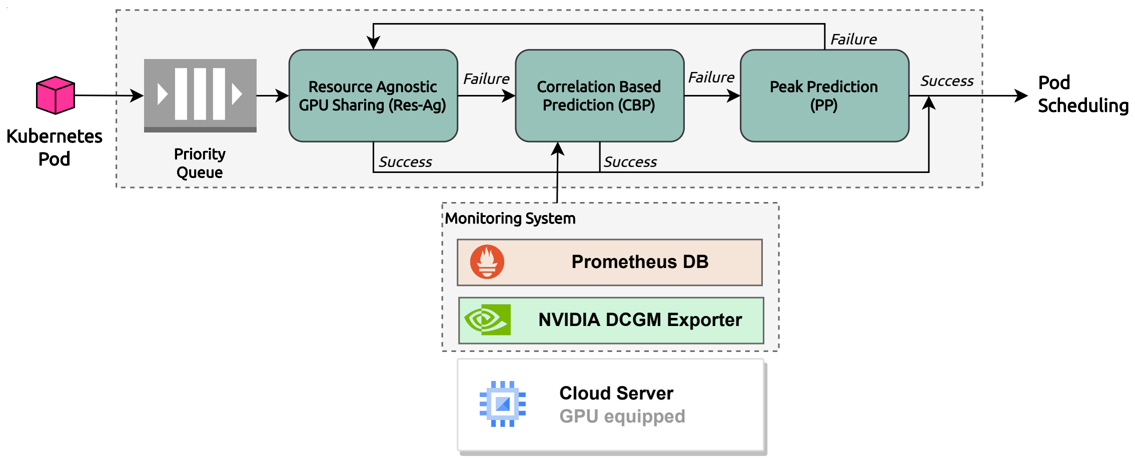


Figure B.5.1.. Resource Aware GPU Colocation Algorithm

Resource Agnostic GPU Sharing: Our custom scheduler holds a variable that is used as an indicator of the available GPU memory. This variable is initialized to the maximum available memory of the used card in the GPU node. If the inference engine memory request is smaller than the value of this variable, the request can be satisfied and the workload can be scheduled. Whenever an inference engine is scheduled, the value of the indicator variable is decreased by the amount of the memory request. Resource agnostic GPU sharing does not face the memory over-provisioning problem as it is not possible to know a priory that the amount of requested memory is actually the amount that the workload needs to run properly. In our proposed scheduler, we overcome this problem by using real-time memory usage data by our GPU monitoring sub-system. The monitoring system data are collected by performing range queries to Prometheus time series database.

Correlation Based Prediction: Correlation Based Prediction (CBP) [403] provides an estimation for the real memory consumption on a GPU node. The estimation is defined from the 80%-ile of the GPU memory usage rather than the maximum usage. The basic idea of this algorithm is that GPU applications, on an average, have stable resource usage for most of their execution, except for the times when the resource demand surges. In addition, the whole allocated capacity is used for a small portion of the execution time while the applications are provisioned for the peak utilization. CBP scheduler virtually resizes the running workloads for a common case, letting more pending inference engines to be colocated.

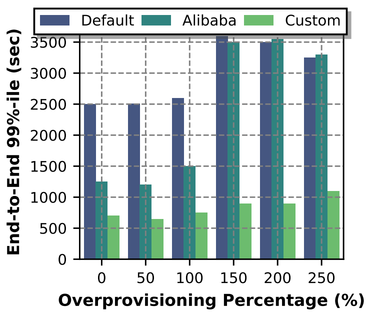
In order to have an accurate estimation, low signal variability is required. The signal variability is calculated using the coefficient of variation (CV) metric [422]. If CV is lower than a defined threshold, the memory usage is defined by calculating the 80%-ile of the signal. The free GPU memory estimation is equal to the difference of the maximum available GPU memory and the memory usage estimation. Finally, if the memory request can be satisfied the workload is scheduled. Otherwise the Peak Prediction algorithm is used.

Peak Prediction: Peak Prediction (PP) [403] relies on the temporal nature of peak resource consumption within an application. For example, a workload that requires GPU resources will not allocate all the memory it needs at once. So, although the GPU memory request cannot be satisfied at the scheduling time, it may be satisfied in the near future. The memory usage prediction is based on an auto regressive model (AR) [423]. For an accurate prediction the auto correlation value of order *k* is calculated. If the auto correlation [424] value is larger than a defined threshold, auto regression of order 1 is performed using linear regression (LR) [425]. If the predicted GPU memory request can be satisfied from PP, then the workload is scheduled. Otherwise, the workload is put into the priority queue and our algorithm attempts to schedule the next available workload from the queue. As we see, PP scheduling decisions depend on the accuracy of the used auto-regressive model and thus linear regression. Even though linear regression is a simplistic approach for predicting the unoccupied memory of the GPU, it can accurately follow the memory utilization pattern (as we further analyze in section B.6). In addition, its low computing and memory requirements, allows the PP mechanism to provide fast predictions at runtime with minimal resource interference.

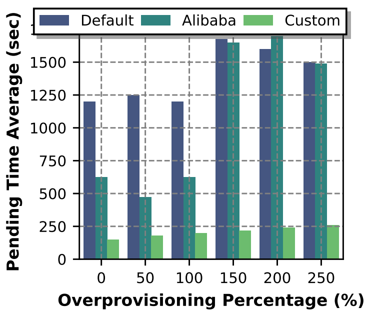
B.6. Experimental Evaluation

We evaluate our custom scheduler through a rich set of various comparative experiments. We consider inference engine workloads for differing intervals between consecutive inference engine arrivals. In each comparative analysis the exact same workload is fed to the Kubernetes GPU scheduler extension [415], the Alibaba GPU sharing extension [416] and the custom scheduler multiple times. Each time a different memory over-provisioning percentage is used.

We provide analysis for homogeneous, i.e., scaling out a single inference service, and heterogeneous workload scenarios. Each workload creates a different inference engine by using the MLPerf inference container we described in section B.3. An inference engine is fully defined from the used backend (e.g. Tensorflow, PyTorch etc.), the pre-trained model, the dataset, the scenario, the GPU memory request and the number of inference queries that are going to be executed. The interval between two consecutive inference



(a) End-to-End Job Execution 99%-ile



(b) Pending Time Average

Figure B.6.1.. End-to-end job execution 99%-ile and pending time average vs over-provisioning percentage homogeneous workload with MIN=5 and MAX=10

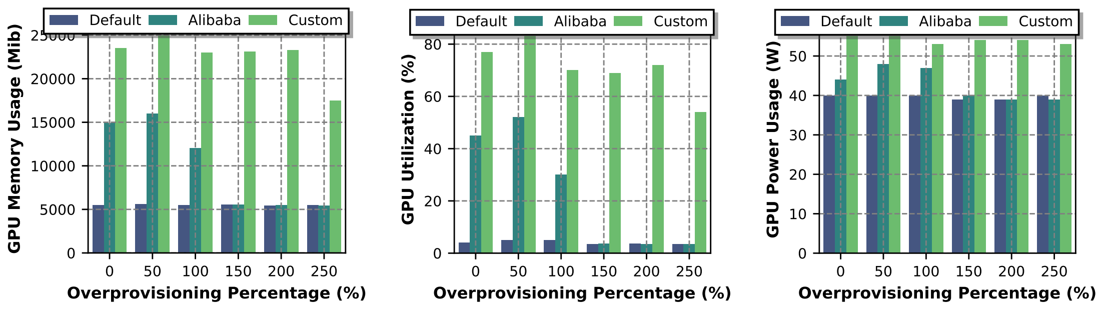
engine arrivals is defined by the values MIN and MAX (random number in [MIN, MAX] interval in seconds).

B.6.1. Homogeneous Workload Evaluation

For homogeneous workload, we consider the Tensorflow ssd-mobilenet engine which uses the Coco (resized 300x300) dataset while each inferences engine executes 1024 queries. Each inference engine requires approximately 7 GB of GPU memory meaning that in a card with 32 GB memory, only 4 can be safely colocated.

Figure B.6.1 shows the end-to-end 99%-ile and the pending time average for all the available schedulers, for different over-provisioning percentages. Custom scheduler offers x6.6 (on average) lower pending time average and x3.6 (on average) lower end-to-end 99%-ile from Kubernetes default GPU scheduler extension. It also offers x5.2 (on average) lower pending time average and x2.8 (on average) lower end-to-end 99%-ile from Alibaba GPU sharing scheduler extension. However, due to the colocation of multiple inference engines, custom scheduler's decisions lead to higher inference engine 99%-ile average.

To understand the above mentioned results, the way each mechanism schedules workloads to the GPU should be analyzed. Kubernetes default GPU scheduler extension allocates the whole GPU resource for each inference engine, leading to severe increase of the pending time average (the average time an inference engine waits in the priority queue). The Alibaba GPU sharing scheduler extension uses a resource agnostic colocation mechanism



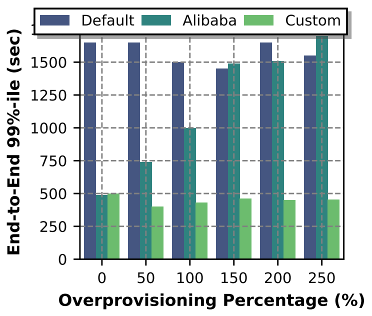
(a) Memory Usage Average (b) GPU Utilization Average (c) Power Usage Average

Figure B.6.2.. Memory usage average, GPU utilization average and power consumption averages vs over-provisioning percentage for homogeneous workload with MIN=5 and MAX=10

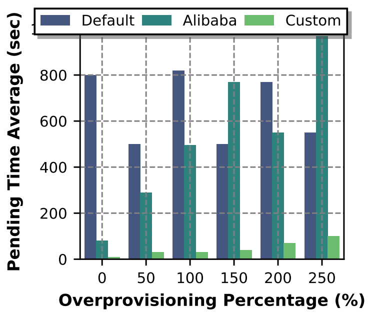
to schedule workloads in the same card. In particular, for over-provisioning percentage equal to 0 % (7 GB memory request) 4 inference engines can be collocated, for 50 % (10 GB memory request) 3 inference engines can be colocated, for 100 % (14 GB memory request) 2 inference engines can be colocated and for 150 %, 200 % and 250 % each inference engine allocates the whole GPU resource. As a result, Alibaba GPU share scheduler extension has similar results with our custom scheduler for over-provisioning percentages equal to 0 % and 50 %. Custom scheduler handles the memory over-provisioning problem in a better way because of its resource aware nature. Figure B.6.1 shows that the proposed scheduler has similar behavior concerning the quality of service metrics for all the different over-provisioning percentages.

Figure B.6.2 shows the memory usage, the utilization percentage and the power consumption averages for all the available schedulers for different over-provisioning percentages. Custom scheduler leads to x3.7 higher memory usage, x16 higher GPU utilization and x1.3 higher power consumption from Kubernetes default GPU extension. It also leads to x2.2 higher memory usage, x2.9 higher GPU utilization and x1.2 higher power consumption from Alibaba GPU sharing scheduler extension. Although we observe an increase in the power usage average, it should be clear that due to the lower overall workload duration the average energy consumption is x2.6 lower from the Kubernetes GPU scheduler extension and x2.2 lower from the Alibaba GPU sharing extension.

In particular, Kubernetes default GPU extension has the lower resource utilization because each inference engine allocates the whole GPU resource. Alibaba GPU share scheduler extension has similar results with our custom scheduler only for 0 % and 50 % over-provisioning percentages. This is expected, since for these over-provisioning percentages



(a) End-to-End Job Execution 99%-ile



(b) Pending Time Average

Figure B.6.3.. End-to-end job execution 99%-ile and pending time average vs over-provisioning percentage heterogeneous workload with MIN=5 and MAX=10

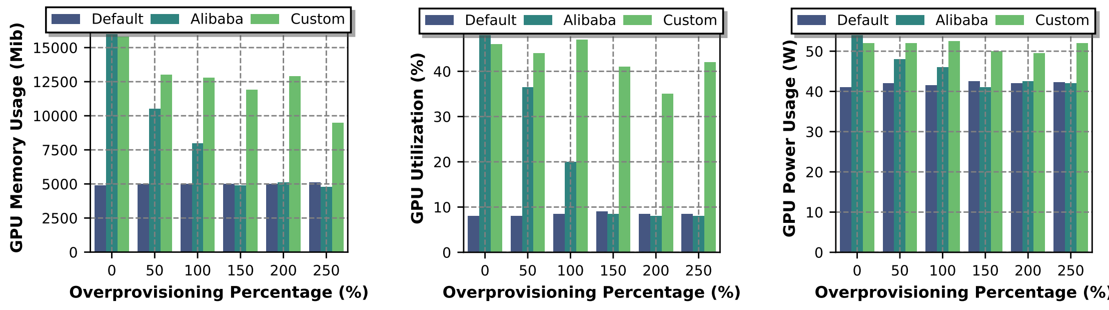
Model	**Dataset**	**Queries/Engine (#Engines)**
mobilenet	Imagenet	1024 (2), 2048 (2)
mobilenet quantized	Imagenet	256 (2), 512 (2)
resnet50	Imagenet	4096 (2), 8192 (2)
sd-mobilenet	Coco (resized 300x300)	128 (3), 1024 (2)
ssd-mobilenet quantized finetuned	Coco (resized 300x300)	64 (2), 1024 (2)
ssd-mobilenet symmetrically quantized finetuned	Coco (resized 300x300)	512 (2), 4096 (2)

Table B.6.1.: Inference engines used for heterogeneous workload evaluation

the scheduler can effectively colocate workloads. The higher the over-provisioning percentage is, the closer the resource utilization is to Kubernetes default GPU extension. Finally, we observe that our custom scheduler has similar behavior concerning the resource utilization for all the different over-provisioning percentages.

B.6.2. Heterogeneous Workload Evaluation

For heterogeneous workload, we consider different inference engines, where each one of them performs a different number of inference queries, as shown in Table B.6.1. Figure B.6.3 shows the quality of service metrics for the heterogeneous inference engine workload. Our proposed scheduler offers x11 lower pending time average and x3.2 lower end-to-end 99%-ile and x8.6 lower pending time average and x2.4 lower end-to-end 99%-ile on



(a) Memory Usage Average (b) GPU Utilization Average (c) Power Usage Average

Figure B.6.4.. Memory usage average, GPU utilization average and power consumption averages vs over-provisioning percentage for heterogeneous workload with MIN=5 and MAX=10

average compared to the Kubernetes default and Alibaba's GPU schedulers respectively. Moreover, Figure B.6.4 shows the respective GPU metrics. We see that, our scheduler leads to x2.5 higher memory usage, x6.1 higher GPU utilization and x1.2 higher power consumption compared to Kubernetes default GPU extension and x1.5 higher memory usage, x2.1 higher GPU utilization and x1.1 higher power consumption compared to Alibaba's GPU sharing scheduler extension.

Container Restarts Analysis: As it was mentioned in section B.5, CBP involves the risk of incorrect scheduling decisions and thus inference engine failures. CBP's prediction accuracy depends on how representative is the free memory signal it receives as input. Since accelerated applications do not always request GPU resources at the beginning of their execution, it is possible that the used signal does not depict the real load of the node. Although several container restarts occured in the previous experiment, we observe that our proposed scheduler still offers better QoS and GPU resource utilization from the baseline state-of-the-art GPU schedulers.

B.7. Conclusion

In this chapter, we presented a resource aware GPU colocation framework for Kubernetes inference clusters. We evaluate the colocation algorithm using workloads that consist of inference engines using different scenarios. We identify and explain the disadvantages of the correlation based prediction (CBP) and peak prediction (PP) scheduling schemes. Finally, we show that our custom scheduling framework improves the defined quality of service metrics while also increases the GPU resource utilization.

Appendix C.

Towards making the most of NLP-based device mapping optimization for OpenCL kernels

Nowadays, we are living in an era of extreme device heterogeneity. Despite the high variety of conventional CPU architectures, accelerator devices, such as GPUs and FPGAs, also appear in the foreground exploding the pool of available solutions to execute applications. However, choosing the appropriate device per application needs is an extremely challenging task due to the abstract relationship between hardware and software. Automatic optimization algorithms that are accurate are required to cope with the complexity and variety of current hardware and software. Optimal execution has always relied on time-consuming trial and error approaches. Machine learning (ML) and Natural Language Processing (NLP) has flourished over the last decade with research focusing on deep architectures. In this context, the use of natural language processing techniques to source code in order to conduct autotuning tasks is an emerging field of study.
In this chapter, we extend the work of Cummins et al., namely Deeptune, that tackles the problem of optimal device selection (CPU or GPU) for accelerated OpenCL kernels. We identify three major limitations of Deeptune and, based on these, we propose four different DNN models that provide enhanced contextual information of source codes. Experimental results show that our proposed methodology surpasses that of Cummins et al. work, providing up to 4% improvement in prediction accuracy.

C.1. Introduction

The ever-increasing amount of data generated and shared by enterprises, industrial and non-profit sectors, and scientific research has resulted in an unprecedented increase in the size and volume of data-intensive jobs [426]. In order to meet the new computational demands of the big data, both hardware and software undergo major changes. Additionally, new latency and power constraints require approaches to solve number of different application and compiler optimization problems. However, their large design decision space to explore, makes tuning applications an even more difficult and time consuming procedure, making the development of tuning heuristics more urgent than ever. Furthermore, contemporary compilers and runtime environments, featuring already hand-coded heuristics, performing this decision making, make their program's performances contingent upon their quality.

Therefore, in order to make heuristic construction more efficient and inexpensive, automation tools have to be imported in this procedure. In this regard, machine learning techniques have been deployed in order to automate the process of selecting the best optimizations [427]. Prediction models using different classes of machine learning are trained through applications' representative features in order to correlate them with their optimal versions. Features can be any important quantifiable properties of applications, static or dynamic, and the choice of the most appropriate features constitutes an extra optimization problem for the designers as well.

Even though machine learning has a proven contribution in automated tuning heuristics [428, 429], its success is contingent on the quality of the extracted features, which is frequently achieved through a combination of domain expertise and trial and error. In order to avoid the extraction of non appropriate features and finally inefficient tuning models, humans are needed to be removed from the loop. Latest publications [8, 430] focus their work on characterizing and tuning applications without using any code feature. Natural Language Processing (NLP) methods are deployed to extract automatically and internally code's text features and feed them to the predictive model. Therefore, NLP models are now able to extract feature representations from source codes automatically and afterwards, other learning systems could employ these learnt feature representations as inputs to predict various down stream tasks.

Among the dozens of optimizations, lately, most of them are focused on the accelerators devices (such as GPUs and FPGAs) deployed to satisfy the even more demanding performance and power constraints from the edge to cloud continuum. One of the most effective and popular one, concerns the optimal heterogeneous device mapping optimization for OpenCL written applications. More specifically, this optimizations refers to the selection of either CPU or GPU device for the most efficient execution of OpenCL kernels in terms of performance.

In this work, we extend DeepTune [8], one of the most influential works on machine-learning based auto-tuning methodologies without any hand engineered feature extractors, attempting to solve the heterogeneous device mapping optimization. We achieve to further improve its effectiveness and finally provide more efficient auto-tuning methods without any need for feature engineering. Our proposed work outperforms DeepTune by providing up to 4.12%, with average 2.65% higher prediction accuracy for the optimal device mapping selection. The rest of this chapter is structured as follows: Section C.2 introduces the related work already published concerning the automation of applications tuning methods. Section C.3 describes the overview of DeepTune's baseline implementation while in Section C.4 we present our proposed improved methodology. The experiments and the findings are described in Section C.5 and finally, in Section C.6 we draw our conclusions.

C.2. Related Work

There have been numerous works conducting with the goal of optimizing source code using machine learning models in order to automate tuning procedure [431]. Back in the early 2008, Agakov et al. [432] were the first to use a machine learning based predictive model to speed up iterative optimization while, in the same year, Cavazos et al. [433] applied logistic regression models to automatically select the best optimization for user's applications. Later, Liu et al. [434] turning to accelerators, used regression trees to optimize CUDA kernels, while Cummins et al. [435] applied classifiers to select the optimal workgroup size of OpenCL compute kernels. Finally, Magni et al. [429] were the first to address the coarsening optimizations through Neural Networks cascade models.

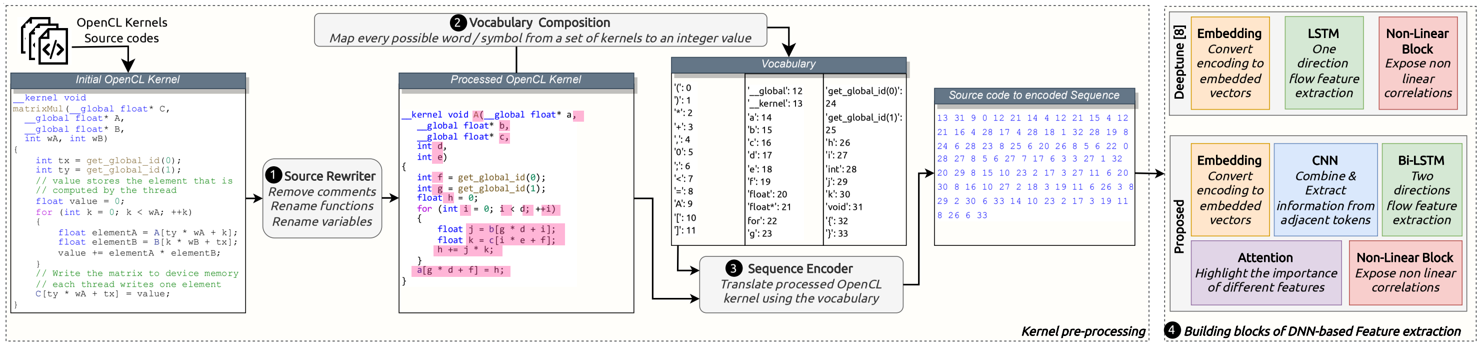
Even though the above works managed to provide sufficient results based on hand crafted program features by developers, heuristics needed to take humans out of the loop. Building auto-tuning models without feature engineering can provide faster, cheaper and more independent heuristics achieving to discover more optimal tuning solutions without any human guidance. Cummins et al. [8] where the first to introduce their work in that direction, with deep neural networks to replace any hand-picked or even compiler IR based automatically extracted features. Surprisingly, their approach matched or surpassed the predictive models using hand-crafted features, proving that deep learning can select more representative and sufficient features than even experts engineers. In the same direction, Ben-Nun et al. [430] managed to apply NLP techniques on the Intermediate Representations (IR) of applications in order to take the feature extraction out of the loop to support a wider range of programming languages. Their experimental results were encouraging, disclosing that the hand-coded features, are not, but an obstacle in the building of the auto-tuning heuristics.

C.3. Deeptune Overview

In this work, we base our research in the work of [8]. Cummins et. al present an end to end architecture that accepts OpenCL kernels as input and decides on the optimal device that those kernels should execute, CPU or GPU. Their proposed methodology consists of two phases: *i)* a kernel pre-processing stage, that transforms the OpenCL kernels to a machine-learning friendly interprentetation and *ii)* a DNN-based feature extraction phase, that receives the transformed code and based on its features identifies the more performance efficient device mapping for the respective kernel. Figure C.3.1 shows an overview of the kernel transformation flow, as well as the basic layers, used as building blocks for feature extraction used in [8]. Next, we briefly describe the basic steps of the work of Cummins et. al, henceforth referred to as Deeptune.

C.3.1. Source Rewriter

The first stage of Deeptune consists of a source rewriter ❶. Its purpose is to reconstruct a given OpenCL kernel to a refined version that eliminates semantically irrelevant information. This refined version can then be more easily processed in an automated manner. Specifically, this component accepts hand written code and it performs the following three actions *i)* removes comments and erase unnecessary spacing *ii)* rewrites function names using an increasing alphabetical order of capital letters [A-Z] and *iii)* rewrites the codes variable using an increasing alphabetical order of lowercase letters [a-z].


OpenCL Kernels
Source codes
Initial OpenCL Kernel
1 Source Rewriter
Remove comments
Rename functions
Rename variables
2 Vocabulary Composition
Map every possible word / symbol from a set of kernels to an integer value
Processed OpenCL Kernel
Vocabulary
3 Sequence Encoder
Translate processed OpenCL kernel using the vocabulary
Source code to encoded Sequence
Kernel pre-processing
Deeptune [8]
Embedding
Convert encoding to embedded vectors
LSTM
One direction flow feature extraction
Non-Linear Block
Expose non linear correlations
Proposed
Embedding
Convert encoding to embedded vectors
CNN
Combine & Extract information from adjacent tokens
Bi-LSTM
Two directions flow feature extraction
Attention
Highlight the importance of different features
Non-Linear Block
Expose non linear correlations
4 Building blocks of DNN-based Feature extraction

```
__kernel void
matrixMul(__global float* C,
  __global float* A,
  __global float* B,
  int wA, int wB)
{
    int tx = get_global_id(0);
    int ty = get_global_id(1);
    // value stores the element that is
    // computed by the thread
    float value = 0;
    for (int k = 0; k < wA; ++k)
    {
        float elementA = A[ty * wA + k];
        float elementB = B[k * wB + tx];
        value += elementA * elementB;
    }
    // Write the matrix to device memory
    // each thread writes one element
    C[ty * wA + tx] = value;
}
```

```
__kernel void A(__global float* a,
   __global float* b,
   __global float* c,
   int d,
   int e)
{
   int f = get_global_id(0);
   int g = get_global_id(1);
   float h = 0;
   for (int i = 0; i < d; ++i)
   {
      float j = b[g * d + i];
      float k = c[i * e + f];
      h += j * k;
   }
   a[g * d + f] = h;
}
```

```
'(': 0      '__global': 12   'get_global_id(0)':
')': 1      '__kernel': 13   24
'*': 2      'a': 14          'get_global_id(1)':
'+': 3      'b': 15          25
',': 4      'c': 16          'h': 26
'0': 5      'd': 17          'i': 27
';': 6      'e': 18          'int': 28
'<': 7      'f': 19          'j': 29
'=': 8      'float': 20      'k': 30
'A': 9      'float*': 21     'void': 31
'[': 10     for': 22         '{': 32
']': 11     'g': 23          '}': 33
```

```
13 31 9 0 12 21 14 4 12 21 15 4 12
21 16 4 28 17 4 28 18 1 32 28 19 8
24 6 28 23 8 25 6 20 26 8 5 6 22 0
28 27 8 5 6 27 7 17 6 3 3 27 1 32
20 29 8 15 10 23 2 17 3 27 11 6 20
30 8 16 10 27 2 18 3 19 11 6 26 3 8
29 2 30 6 33 14 10 23 2 17 3 19 11
8 26 6 33
```

Figure C.3.1.. High-level overview of the end-to-end optimal device mapping methodology for OpenCL kernels. The *kernel-preprocessing* phase transforms source codes to machine learning interpretable format and the *feature extraction*, based on the optimization problem, it generates highly descriptive characteristics for each kernel. ❹-top shows the *feature extraction* approach of Deeptune [8] and ❹-bottom our proposed building blocks.

C.3.2. Vocabulary Composition

The second step of Deeptune's methodology is the vocabulary composition ➋. This step concerns the construction of a vocabulary that maps code related definitions (e.g., `__kernel`, `int`) and punctuation marks (e.g., parentheses, semicolons) to a set of integer representations.

This vocabulary is mandatory, since machine-learning models receive as input numeric values and are not able to process source code directly. To build the corpus vocabulary, this step considers all the possible words and symbols appeared over all the examined OpenCL kernels and performs word level tokenization, which maps each token to a unique integer identifier, called *token id*.

C.3.3. Sequence Encoder

The last step of the pre-processing phase is the sequence encoder ➌. This component combines the processed OpenCL kernel (output of step ➊) with the corpus vocabulary (output of step ➋) and provides an integer sequence that corresponds to the respective input code.

C.3.4. DNN-based feature extraction

As mentioned before, Deeptune utilizes a DNN-based approach to automate the process of feature extraction, thus eliminating completely the need for hand-crafted solutions. The proposed DNN architecture consists of three major layers, that form the building blocks of their model (➍–top). First, an Embedding layer acts as a language model that receives as input the tokenized source code sequence and converts them to embedding vectors of dimension $D = 64$. Next, the embedded vector is passed to a block of two Long Short-Term Memory [302] layers, that is responsible for exposing one-way sequential information on the input vector. Last, DeepTune's model final component is a non-linear block, that is responsible for exposing non-linearities in the data. It consists of two fully connected artificial neural network layers, where the initial layer is composed of 32 neurons. Each possible heuristic decision is represented by a single neuron in the second layer. The model's confidence that the associated choice is right is represented by the activation of each neuron in the output layer. Taking the output layer's argmax yields the decision with the highest activation.

C.3.5. Deeptune's Limitations & Motivation

While the work of [8] is revolutionary in automating the feature extraction process of GPU code, we argue that it neglects important aspects and inherent characteristics of the structure and nature of modern programming languages, which, if approached properly, could provide useful insights. Moreover, novel advancements in the domain of machine learning and NLP [430,436,437], can provide more representative and/or diverse features, unveiling additional hidden patterns in the underlying data. Specifically, we pinpoint three important remarks that Deeptune disregards and which also form the motivation of our work:

R1) Relation between adjacent tokens: The structure of a programming language is very well defined, with syntactical rules. The syntax of a programming language is a collection of rules defining the combinations of symbols that constitute appropriately organized statements or expressions in that language. Deeptune neglects that structure, by treating tokens sequentially, thus it makes no consideration for the premise that adjacent tokens provide additional context for code comprehension. In the example below, the model will assign a high probability to the right token, which is `int`, based on the structure of the variable's `a` declaration.

```
int main(){
    int a = 10;
    ___ b = a + 1;
}
```

Listing C.1: Example code that highlights the dependence of adjacent tokens

R2) Preserve past and future information: While Deeptune examines hidden relationships in the sequential data through the LSTM layers, it only acquires knowledge exclusively from the input's subsequent pass, since unidirectional LSTMs only preserve inputs that has already passed through it using the hidden state. However, the typical flow of imperative programming reveals bidirectional inter-dependencies, from the beginning of the code to its end and vice-versa. A typical example is the following:

```
float sqrt(int a);
int main(){
    int x = sqrt(2);
    // several lines of code
}
float sqrt(int a){
    // implementation
}
```

Listing C.2: Example code that depicts future dependencies

where the call of a function is several lines afterwards than it's implementation.

R3) Significance-aware feature processing: Last, Deeptune treats all the input features equally, even if part of it is less significant. However, it is apparent that not all source code semantics provide proportionate contextual information. For example, a variable declaration (e.g., `int`) is not as critical as the definition of a loop (e.g., `for`). Therefore, it is of great importance to be able to distinguish more valuable features from the less significant ones.

C.4. Proposed Methodology

This section describes the core concepts of our methodology for optimal device mapping of OpenCL kernels between CPU and GPU devices based on NLP techniques. The rationale behind the proposed processing flow is driven by the remarks (R1-R3) drawn in Section C.3.5. Specifically, similar to [8], our methodology consists of two phases, the *kernel pre-processing* and the *feature extraction*, as depicted in Fig. C.3.1. However, it extends the latter to account for additional contextual information on the examined kernels.

C.4.1. Kernel pre-processing

The kernel pre-processing phase of our methodology is identical to the one proposed in Deeptune. We replicate the pre-processing components described in Sections C.3.1 to C.3.3, which correspond to steps ❶-❸ of Fig. C.3.1.

C.4.2. Building blocks for feature extraction

We give special emphasis on the building blocks used to design the DNN model that performs the device mapping classification task (❹). Compared to Deeptune, we incorporate three additional processing layers, that aim to tackle Deeptune's limitations presented above.

First, we extend DeepTune by adding a CNN layer, that receives as input the output of the embedding layer. This method takes advantage of the inherent structure of a textual label by identifying common attributes shared by multiple words and dividing them into three equal parts called trigrams [438]. Trigrams, we believe, are an effective representation of programming syntax structure and helps mitigating the gap of remark **(R1)**. It is composed of 2 layers: one 1-d convolutional layer, with 64 filters, kernel size of 3 and striding step 1, and one max pooling layer of size 4. The embedding layer transmits the words to the convolutional layers in the form of sentences (in our case code instructions). Convolution layer convolve the input using pooling layers; pooling layers aid in reducing the representation of input phrases, input parameters, computation, and overfitting in the network.

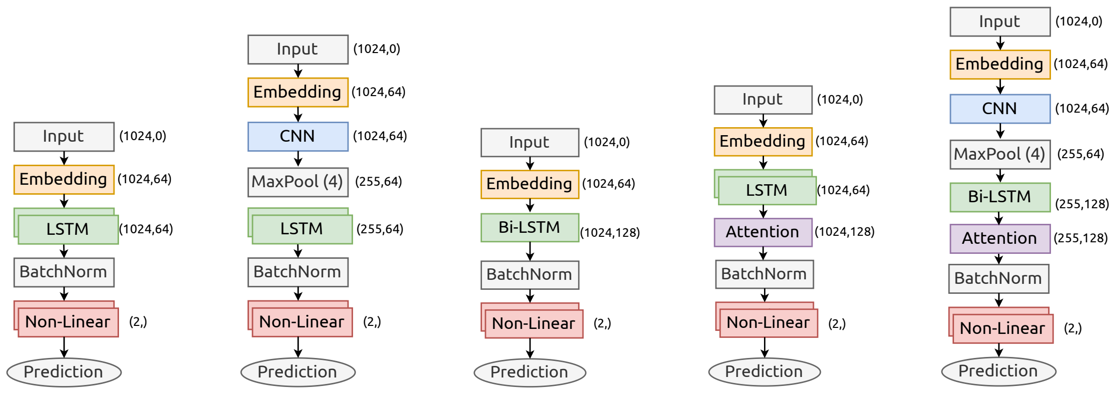
For preserving both past and future information on the input data (remark **R2**), we employ bidirectional LSTM layers. Specifically, Bi-LSTMs are able to obtain knowledge about the sequence in both directions, backwards (future to past) and forwards (past to future) and recognize long-term dependencies [439]. The core idea, is to split the state neurons of a regular LSTM in a part that is responsible for the positive time direction (forward states) and a part for the negative time direction (backward states). Outputs from forward states are not connected to inputs of backward states, and vice versa. Then the hidden states of the two LSTMs are combined to find the hidden state for each time point. It is proven to be more accurate than the traditional LSTM networks [440], but in trade off significant training time. In our case, bidirectional keeps information from both directions, making it easier for the network to understand long-term code dependencies, such as function declarations.

Last, we tackle the problem of significance-aware feature processing (remark **R3**) by exploiting a novel solution in the field of NLP. namely Attention layers [436]. The attention block highlights the importance of different features that are highly correlated with classification, by assigning weights to features, extracting the contextual information. Attention can be proven to be useful in the case of programming code modeling, emphasizing more critical points, such as branches, loops etc., against less important, such as variable declarations, increments etc.

C.4.3. Examined DNN architectures

To understand the impact of each additional layer on the overall accuracy of the device mapping problem, we design different DNN architectures with diverse combinations of the aforementioned building blocks, as shown in Fig. C.4.1. These architectures extend the baseline model of Deeptune (Fig. C.4.1a) in the following directions:

- **Deeptune-CNN**: Includes a CNN layer right after the Embedding, thus, introducing the aspect of trigrams (Fig. C.4.1b). This model examines the impact of



(a) Deeptune [8] (b) Deeptune-CNN (c) Deeptune-BiLSTM (d) Deeptune-Attention (e) Hybrid Architecture

Figure C.4.1.. Examined DNN architectures

feature extraction from adjacent tokens in the source code (R1).

- **Deeptune-BiLSTM**: We replace the two unidirectional LSTM layers of Deeptune, with a bidirectional one (Fig. C.4.1c). Through this model, we explore the effect of preserving past and future information in the code (R2).

- **Deeptune-Attention**: This architecture introduces an Attention layer right after the LSTM, which identifies important features in the input. With this model, we explore whether a significance analysis on the features of the input affect the efficiency of the prediction (R3).

- **Hybrid Architecture**: Final, the hybrid architecture combines all the aforementioned techniques and examines their aggregated impact on the overall accuracy of the model.

C.5. Evaluation

We evaluate our proposed methodology and compare it directly with Deeptune by examining the accuracy of each one of the DNN architectures presented in Section C.4.3.

Table C.5.1.: Technical characteristics of heterogeneous Edge nodes and Cloud Server

Platform	**Frequency**	**Memory**	**Driver**
Intel Core i7-3820	3.6 GHz	8 GB	AMD 1526.3
AMD Tahiti 7970	1000 MHz	3 GB	AMD 1526.3
NVIDIA GTX 970	1050 MHz	4 GB	NVIDIA 361.42

C.5.1. Examined Dataset

We base our evaluation on the dataset of [8] in order to have accurate comparison between the different model architectures. The dataset contains preprocessed and tokenized OpenCL kernels, from 7 different benchmark suites. Additionally, it contains the execution times for each kernel on two GPU devices, the AMD Tahiti 7970 and the NVIDIA GTX 970, as well as on an Intel Core i7-3820 CPU. Table C.5.1 contains details about the CPU-GPU platforms.

The prediction target is the platform in which the execution time is lower. More precisely, when we examine the AMD GPU and Intel CPU cases, the target is $[1, 0]$ if the kernel runs faster on the GPU and $[0, 1]$ if the kernel runs faster on the CPU. This is also referred to as one-hot encoding. Likewise, for the NVIDIA GPU and the Intel CPU case.

C.5.2. Experimental Setup

We use stratified 10-fold cross-validation to evaluate the predictive quality of each model. Each program is randomly assigned to one of ten equal-sized sets; the sets are balanced to ensure a consistent proportion of samples from each class across the whole set. A model is trained on all but one of the sets' programs and then tested on the programs from the unseen set. This procedure is done for each of the ten sets in order to provide a comprehensive prediction for the entire dataset.

All models were implemented in Python using Tensorflow[1] and Keras[2] backends. To ensure the most accurate comparison, we seed our layers with the same number, in order to be initialized with the same weights. The maximum sequence length was set to 1024 and the learning rate at 10^{-3}. We used categorical cross entropy as loss function, a batch size of 64 and trained for 50 epochs. As optimizer, we used the adaptive learning rate

[1]Tensorflow: https://www.tensorflow.org/
[2]Keras: https://keras.io/

Table C.5.2.: Technical characteristics of heterogeneous Edge nodes and Cloud Server

	AMD Tahiti 7970	**NVIDIA GTX 970**	**Average**
DeepTune [8]	83.23%	80.29%	81.76%
DeepTune-CNN	85.88%	80.01%	82.94%
DeepTune-bilstm	84.85%	80.88%	82.87%
DeepTune-Attention	83.91%	81.03%	82.50%
CNN-BiLSTM-Attention	**87.35%**	**81.47%**	**84.41%**

optimization algorithm, Adam. The experiments were carried out on an NVIDIA Tesla V100 GPU. Finally, we used TensorBoard[3] to measure and visualize parameters like loss and accuracy.

C.5.3. Experimental Results

The average accuracy of each model, as measured in a 10-fold test set, is shown in Table C.5.2, where the best results are printed in **bold** font.

Baseline architecture: The baseline architecture is Deeptune, the model suggested in [8]. For the shake of fair comparison, we have retrained the model using the steps of Sec. C.5.2, rather than hard copying the results from [8]. We observe that Deeptune achieves an average accuracy of 81.76%, with better results on the AMD platform with 83.23%, while in the NVIDIA device it scored 80.29%.

Impact of Trigrams (R1): We observe that the notion of trigrams helps the model's code comprehension capabilities. Specifically, the Deeptune-CNN model already outperforms the baseline by 2.65% on AMD platform and 1.18% on average, but performs slightly worse on the NVIDIA platform (0.28% accuracy drop), which, however, is statistically insignificant. We also observe that the introduction of the CNN layer provides the greatest accuracy increment in the case of AMD GPU compared to that of Bi-LSTM and Attention.

Impact of preserving past and future information (R2): The Deeptune-BiLSTM model, performs better in both devices compared to Deeptune. More specifically the model scored 84.85% on the AMD device and 80.88% on the NVIDIA, resulting to an average precision of 82.87%, that is 1.11% improvement than the baseline. Moreover,

[3]TensorBoard: https://www.tensorflow.org/tensorboard/

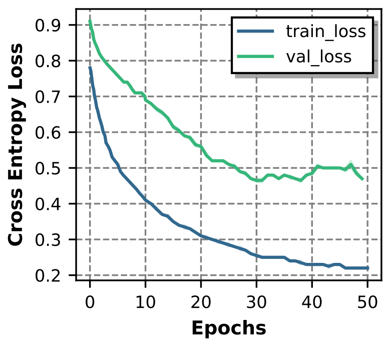


Figure C.5.1.. Train and validation loss of the Hybrid Architecture model.

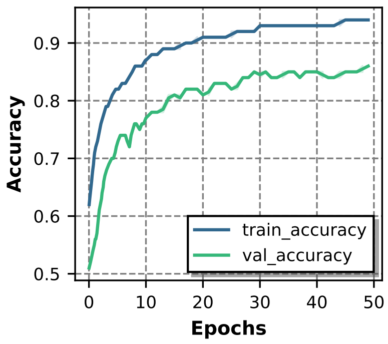


Figure C.5.2.. Train and validation accuracy of the Hybrid Architecture model.

we notice that similar to the case of trigrams, the accuracy increment is lower for the NVIDIA case.

Impact of feature significance (R3): The addition of an attention layer in Deeptune, also outperforms the baseline in both devices, scoring 83.91% on the AMD platform and 81.03% on the NVIDIA, leading to 82.50% on average. It appears to give the best results, on the NVIDIA platform, i.e., 0.73% higher than the baseline.

Hybrid Architecture: Last, our proposed hybrid model, CNN-BiLSTM-Attention, outperforms the baseline model and all other models. This reveals that by combining all of our proposed feature processing steps, we can obtain much greater contextual information from the processed kernel and, thus, maximize the accuracy of our model. Specifically, the hybrid architecture scored 87.35% on the AMD platform and 81.47% on the NVIDIA device, that is 4.12% and 1.18% higher than the baseline, respectively, resulting in an average accuracy increase of 2.65%, that is 84.41%.

For the shake of completeness, we also report the loss and accuracy curves over the training phase, for the Hybrid Architecture. Figures C.5.1 and C.5.2 show the respective results, where each curve corresponds to the average loss and accuracy over the 10-fold cross train and validation sets. As can be seen, the validation loss decreases until epoch 40, at which point it remains pretty constant. Because the training loss continues to decrease, we stop training until epoch 50 to avoid overfitting [441].

C.6. Conclusion

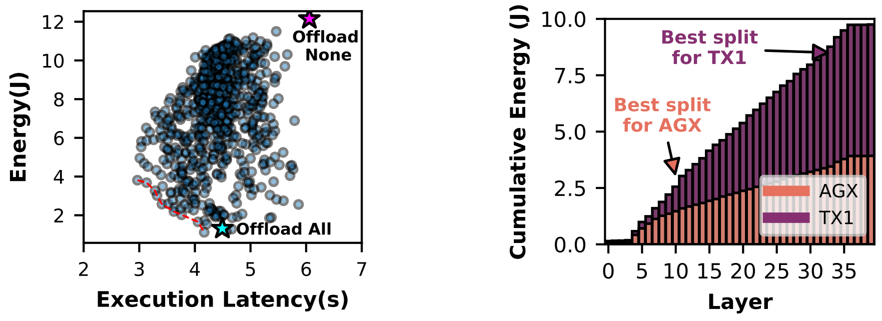
In the era of extreme device heterogeneity, choosing the most appropriate device for application execution forms a really challenging problem. In this chapter, we build upon the work of Cummins et al. [8], and pinpoint some of its limitations. Then, we examine the impact of additional feature extraction approaches for improving the accuracy of optimal device mapping prediction for OpenCL kernels. On average, our suggested model, namely CNN-BiLSTM-Attention, outperforms the baseline model, surpassing it on both prediction challenges. We achieve 4.12% higher prediction accuracy than the baseline on the AMD platform and 1.18% higher on the NVIDIA platform.

In future work, we will expand our method to source code modeling by using transformers to do unsupervised training on unlabeled C code in order to enhance programming language understanding [437]; continue our investigation into the optimization of CUDA kernels; and train models to aid in the development of energy-efficient embedded devices.

Appendix D.

RoaD-RuNNer: Collaborative DNN partitioning and offloading on heterogeneous edge systems

Deep Neural Networks (DNNs) are becoming extremely popular for many modern applications deployed at the edge of the computing continuum. Despite their effectiveness, DNNs are typically resource intensive, making it prohibitive to be deployed on resource- and/or energy-constrained devices found in such environments. To overcome this limitation, partitioning and offloading part of the DNN execution from edge devices to more powerful servers has been introduced as a prominent solution. While previous works have proposed resource management schemes to tackle this problem, they usually neglect the high dynamicity found in such environments, both regarding the diversity of the deployed DNN models, as well as the heterogeneity of the underlying hardware infrastructure.
In this chapter, we present RoaD-RuNNer, a framework for DNN partitioning and offloading for edge computing systems. RoaD-RuNNer relies on its prior knowledge and leverages collaborative filtering techniques to quickly estimate performance and energy requirements of individual layers over heterogeneous devices. By aggregating this information, it specifies a set of Pareto optimal DNN partitioning schemes that trade-off between performance and energy consumption. We evaluate our approach using a set of well-known DNN architectures and show that our framework i) outperforms existing state-of-the-art approaches by achieving 9.58× speedup on average and up to 88.73% less energy consumption, ii) achieves high prediction accuracy by limiting the prediction error down to 3.19% and 0.18% for latency and energy, respectively and iii) provides lightweight and dynamic performance characteristics.



(a) ResNet101 energy and latency for different partitioning schemes

(b) ResNet101 energy's optimal partitioning for different devices

Figure D.1.1.. DNN optimal partitioning schemes w.r.t. *a)* different optimization goals and *b)* device heterogeneity.

D.1. Introduction

Over the last years, the growth of applications that utilize sophisticated Machine Learning (ML) techniques to make value out of complex data is rapidly increasing and is expected to grow further in the future. In this direction, Deep Neural Networks (DNNs) are being widely adopted by many application domains, including, but not limited to, autonomous driving [442], biomedical and healthcare applications [443] and Intelligent Personal Assistants (IPAs) [444], mainly due to their capability in offering high prediction accuracy.

Such types of applications are typically deployed at the edge of the computing continuum, closer to where data are generated, in order to enhance security and minimizing the data transfer latency to the cloud [445]. Even though DNNs provide extremely accurate results, their computational and memory requirements can skyrocket, thus introducing several barriers on how to be efficiently deployed on resource-contrained edge computing devices [446]. Considering also that DNNs are gradually becoming deeper to support more discriminative results and more levels of hierarchy are integrated in the learned representation [447], the aforementioned problem becomes even more intense.

Traditionally, to overcome this limitation, the DNN inference of edge devices was offloaded to high-end servers hosted on cloud premises (e.g., Amazon Elastic Inference, Azure Machine Learning). However, this approach leads to huge amount of data travelling back and forth in the continuum resulting in high latency and energy consumption. Moreover, all the computational effort is being concentrated in the cloud, rendering it incapable to cope with the ever-increasing demand for resources [448]. Aiming to deliver more efficient and energy proportional computing systems, hardware vendors (e.g., Nvidia, Xilinx) are introducing more powerful edge devices, which often integrate conventional CPUs, hardware accelerators and specialized units for DNN computations (e.g., Nvidia's Tensor cores) as a system-on-chip (SoC). While such hardware devices deliver increased compute density, they are also extremely power hungry, thus, becoming restrictive for applications that operate under certain energy or battery constraints.

Aiming to identify the "golden ratio" between performance and energy efficiency for edge computing devices, while also limiting the network latency bottleneck, *DNN partitioning and offloading* has been identified as a promising solution [449, 450]. The aim of DNN partitioning is to provide a fine-grained layer-level split of the DNN, where a portion of the computation is performed locally on the edge device and the rest on the cloud. Still, identifying an optimal partitioning scheme is not trivial and depends on a triptych of user-defined requirements and hardware-oriented criteria, i.e., *i)* the architecture of the deployed DNN model (e.g., computational/energy requirements and data offloading overhead per layer), *ii)* the inherent nature of the application itself (e.g., latency critical applications would sacrifice energy for performance) and *iii)* the underlying hardware characteristics. As a motivational example, Fig. D.1.1a shows the performance and energy of all the possible partitioning schemes for a ResNet101 architecture. We see that model partitioning can provide significantly faster execution and less energy consumption compared to on-board and offloaded execution, while also each optimization objective is attained through different partitioning schemes. Moreover, Fig. D.1.1b further reveals, that the optimal partitioning scheme also relies on the underlying hardware, with different edge devices providing different splits.

To tackle these challenges, several prior research approaches have examined the problem of DNN partitioning and offloading [449–451]. These solutions assume that the architecture of the deployed model, as well as the underlying hardware are known a priori and apply the offloading scheme based on extensive profiling of the DNN. However, edge computing environments are extremely dynamic, both with respect to the devices arriving in the network and the alternative applications deployed. Moreover, novel DNN architectures are emerging [445], leading to more and more complex techniques, such as skip-layer connections, early exits and others.

In order to adapt to the challenges that new DNN architectures impose and to the dynamic nature of recent edge computing systems, we design an efficient resource management framework to dynamically allocate, partition and offload DNN layers among edge and cloud resources, aiming to provide performance and/or energy consumption optimizations and trade-offs. **The novel contributions of this work are:**

- **We present `RoaD-RuNNer`**, a novel resource management framework consisting of a set of *Offline* and *Online Decision Making Mechanisms*.
- **We implement a collaborative filtering based prediction mechanism** in order to provide per layer predictions regarding execution time and energy consumption.
- **We design and integrate a dynamic partition mechanism**, for efficiently splitting and offloading DNN layers.
- **We conduct an extensive experimental evaluation of our proposed framework**. We compare our framework with a set of baseline algorithms and state-of-the-art DNN offloading approaches over real hardware and networking, showing that it outperforms existing state-of-the-art approaches by achieving 9.58× speedup on average and up to 88.73% less energy consumption average.

The remainder of this chapter is organized as follows: Section D.2 presents related work. In Section D.3 we analyze our proposed prediction and offloading mechanism. Section D.4 provides our experimental and comparative evaluation of the proposed resource management scheme, while Section D.5 concludes this work.

D.2. Related Work

Several works have been conducted in order to address the resource management challenges of DNNs deployed on edge computing systems. Deep learning model partitioning and offloading is becoming a rising trend in recent edge computing systems. Research conducted by [452] considers various present and future challenges for efficiently deploying deep learning models on the edge. In a similar perspective, authors of [453] provide an overview of the overarching architectures, frameworks, and emerging key technologies towards training/inference for deep learning models at the network edge.

Aiming to provide DNN resource management schemes, authors of [451] propose a partitioning-based DNN offloading technique for edge computing, by dividing the DNN model into partitions and uploading them to the edge server. In a similar perspective, authors of [454] present a distributed progressive inference engine that addresses the challenge of partitioning CNN inference across device-server setups. In [455] a framework that dynamically partitions a DNN model that adapts to the changes of computational resources and network condition is proposed. Authors of [449] design a workload partitioning algorithm to decide efficient DNN partitioning policy in real-time, while in [456] authors utilize several IoT devices by creating a local collaborative network for a subset of deep learning models, mainly focusing on the impact of convolutional layers. Last but not least, authors of [450] present the efficiency of DNN processing on the cloud based on pre-loaded layers.

Although research has illuminated the resource management and offloading of neural network based applications on edge computing systems, no study to date, according to our knowledge, has incorporated collaborative filtering in order to produce an efficient, decentralized resource management solution for Neural Network offloading at single layer granularity, while targeting heterogeneous CPU/GPU architectures. *Neurosurgeon* [450], is the most similar approach to ours, however there exist several fundamental differences. We target a fully dynamic DNN offloading framework, in contrast to *Neurosurgeon*, where DNN weights are loaded a priori to the cloud server. Moreover, *Neurosurgeon* is limited to operate over models without skip-layer connections, in contrast to `RoaD-RuNNer`, which is designed to operate over all types of DNN/CNN.

D.3. `RoaD-RuNNer`: A collaborative DNN partitioning & offloading framework

`RoaD-RuNNer` tackles the problem of DNN partitioning and offloading over heterogeneous CPU/GPU edge computing systems, where a portion of the computational burden can be offloaded from the edge to the cloud for remote execution. Each individual edge node accommodates a set of DNN inference tasks which need to be executed. `RoaD-RuNNer`'s goal is to identify optimal DNN splittings, down to the granularity of a single layer and offload computationally heavy slices, aiming to optimize the total inference execution latency and/or the energy consumption. Figure D.3.1 depicts an overview of `RoaD-RuNNer`. The major components of our proposed framework are split in two major phases: (i) *Offline Phase* and (ii) *Online Phase.* The former is composed of *DNN-Profiler* and *Network Profiler*, while the latter consists of the *Predictor* and *Offloader* mechanisms. The core functionality of these components is described in the rest of this section.

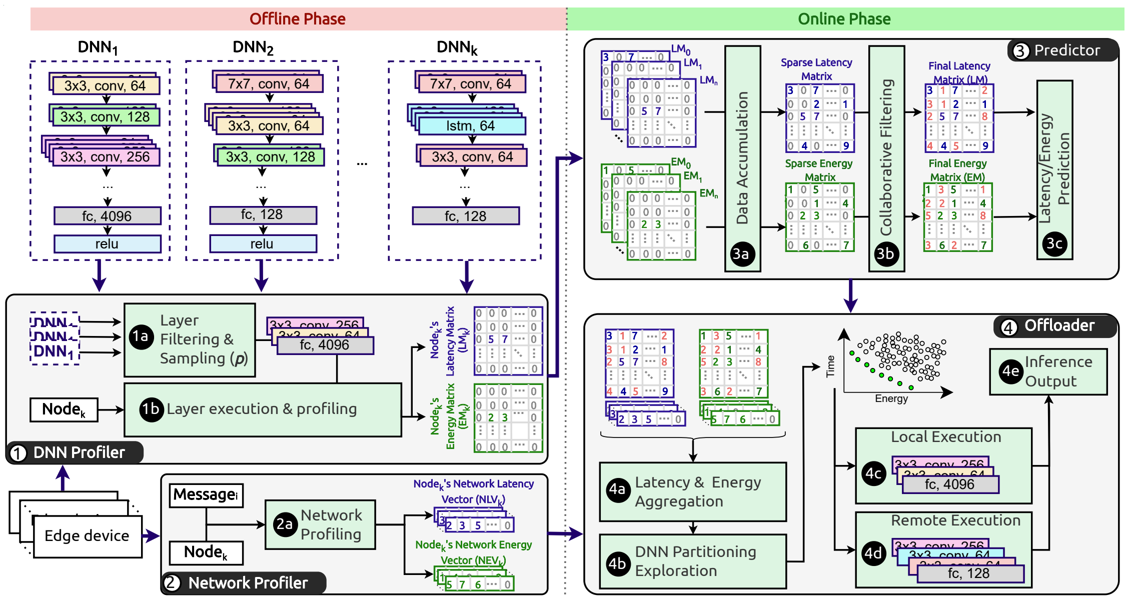


Figure D.3.1.. Overview of Online and Offline RoaD RuNNer Architecture.

D.3.1. Offline Phase

Offline Phase consists of two distinct components: (i) *DNN Profiler* (❶) and (ii) *Network Profiler* (❷), which are responsible for generating the required data and knowledge to be fed as input to the run-time mechanisms later. As input to the framework, we provide alternative neural network configurations to each single node of the composed edge computing network.

DNN Profiler: Aiming to take advantage of the inherent distributed nature of the edge computing paradigm and the heterogeneity in terms of edge devices and deep learning models, we implement a collaborative filtering mechanism [41, 457], in order to train our system to efficiently predict per layer execution time and energy consumption for each device, respectively. As a first step of the offline decision making, we integrate a *Layer Filtering and Sampling* (1a) component, aiming to filter and sample the layers of the alternative neural networks that are fed as input to our framework. More specifically, a small percentage p of the input layers in each DNN is randomly selected to be accurately profiled on the edge device itself. This ensures that profiling time does not skyrocket when the total number of layers in the DNN increases. The percentage p is defined through experimentation.

Next, the layers that have been selected through the filtering and sampling process are propagated as input to the *Layer Execution & Profiling* (1b) step. Each of the sampled layers is executed locally on each node, respectively. The execution is profiled, aiming to gather data regarding the execution latency and the energy consumption per layer. Thus, for every single node k, we extract the Node's Latency Matrix (LM_k) and Node's Energy Matrix (EM_k), respectively. The LM_k and EM_k are propagated as input for further processing to the *Predictor* mechanism, as described in Section D.3.2. This process is triggered once for each node.

Network Profiler: The efficiency of an edge computing system is directly related to its ability to effectively operate over the existing network infrastructure. Moreover, offloading decision making mechanisms should consider the overhead imposed by the underlying network. Thus, for each individual node k we integrate an extended *Network Profiling* mechanism (2a). As input we provide a set of alternative workloads to be sent/received from each edge device to the cloud server. We profile the transmission latency and power of data sent and received from and to the edge device. The profiled messages range from KB to GB orders of magnitude.

After the profiling is finalized, each node k is characterized by two vectors: (i) The Network Latency Vector (NLV_k) and the Network Energy Vector (NEV_k), which represent the latency and energy requirements for alternative message sizes, respectively. The produced vectors are utilized as dataset to produce polymonial trendlines, in order to predict the latency and energy requirements of a given input message size. Thus, for each individual device, fourth-order polynomial curves are generated in order to be utilized for run-time prediction during the *Online Phase*.

D.3.2. Online Phase

An efficient dynamic resource management scheme should be able to make decisions in a run-time manner. Thus, we integrate into our framework two key components: (i) *Predictor* (❸) and (ii) *Offloader* (❹). The former is responsible for dynamically predicting latency and energy per layer, while the latter is responsible for network and collaborative filtering data aggregation, dynamic DNN partitioning andRoa offloading.

Predictor: The output matrices produced by the *DNN Profiler* (❶) of each single node k during the offline phase are accumulated to the cloud server (3a). Two new sparse matrices are produced, i.e. *Sparse Latency Matrix* and *Sparse Energy Matrix*, respectively. Each row of the matrix represents an edge node, while each column denotes a

Algorithm 1: DNN Partitioning Algorithm

```
Partition(network):
    for layer in (0,...,N-1) do
        /* predict local for layers 0 to layer */
        l1=predictLocal(0, layer)
        for j in (layer+1,...,N-1) do
            /* predict network,cloud for layers $i+1$ to $j$ */
            n=predictNetwork(layer+1, j)
            c=predictCloud(layer+1, j)
            /* predict network,cloud for layers $j+1$ to $N-1$ */
            l2=predictLocal(j+1, N-1)
            totalPredictions=accumulatePredictions(l1,n,c,l2)
    /* find Pareto Optimal */
    paretoPoints = DesignSpaceExploration(totalPredictions)
    return paretoPoints
```

single layer type. The *Collaborative Filtering* mechanism (3b) is triggered, based on matrix decomposition, i.e. based on the factorization of each matrix into a product of matrices. The unknown values are filled, and the *Final Latency Matrix(LM)* and the *Final Energy Matrix(EM)* are produced. After the predictions are finalized, each device retrieves its corresponding results from the cloud server, which are utilized for the final *Latency/Energy Prediction* (3c). Opposed to prior works, which mostly rely on extended profiling for each new deep learning model [451, 454], our framework is adaptable to dynamic scenarios. New nodes that dynamically join the network are integrated in the existing latency and energy matrices and the collaborative filtering matrices are updated. New incoming DNNs can benefit from the existing layers in the collaborative filtering matrices.

Offloader: The last component of the *Online Phase* is responsible for the aggregation of network and collaborative filtering data and the dynamic DNN partitioning and offloading. For each single node k, the network profiling vectors (NLV_k, NEV_k), and the collaborative filtering matrices (LM, EM) for latency and energy are aggregated (4a), in order to provide the final prediction per layer, including computation and transmission overhead. Next, we proceed to the *DNN Partitioning Exploration* (4b), in order to provide a set of Pareto optimal solutions, aiming to optimize performance and/or energy consumption. The DNN partitioning exploration algorithm is illustrated in Algorithm 1. A set of Pareto optimal solutions in generated. Each Pareto point corresponds to an alternative partition, on which a subset of DNN layers is executed locally (4c) and the rest are offloaded for remote execution on the cloud server (4d). Each partition is identified by two indexes i, j where layers 0 to i are executed locally, layers $i+1$ to j are executed on the cloud and the layers from $j+1$ and up to the end of the network are executed locally.

Table D.4.1.: Technical characteristics of heterogeneous Edge nodes and Cloud Server

Device	CPU	L2	L3	DRAM	GPU (cores)
Nano	4×Cortex-A57@1.4GHz	2MB	-	4GB	128@0.9GHz
X1	4×Cortex-A57@1.4GHz	2.5MB	-	4GB	256@1.0GHz
NX	6×Carmel@1.4GHz	6MB	4MB	8GB	384@1.1GHz
AGX	8×Carmel@2.2GHz	8MB	4MB	32GB	512@1.4GHz
Server	2×20 Intel Xeon 5218R@2.1GHz	1MB	28MB	128GB	5120@1.2GHz

Fully local and fully remote executions remain valid alternatives, in case they belong to the Pareto optimal solutions. Moreover, in contrast to existing approaches [450] our framework is designed to handle shortcut dependencies, i.e. skip-layer connections. Such kind of dependencies are resolved by encapsulating the layers that create the dependency in an larger atomic block that has a single input and output and are offloaded as an individual component.

D.4. Experimental Evaluation

D.4.1. Experimental Setup

Hardware Infrastructure: We deploy an in-house system setup consisting of heterogeneous CPU/GPU devices, the specifications of which are shown in Table D.4.1. As edge devices, we utilize a set of NVIDIA GPU-SoCs, to exploit power/performance trade-offs. As offloading machine, we employ a powerfull x86 server equipped with an NVIDIA V100 GPU, which forms a typical setup both in edge and cloud premises [458].

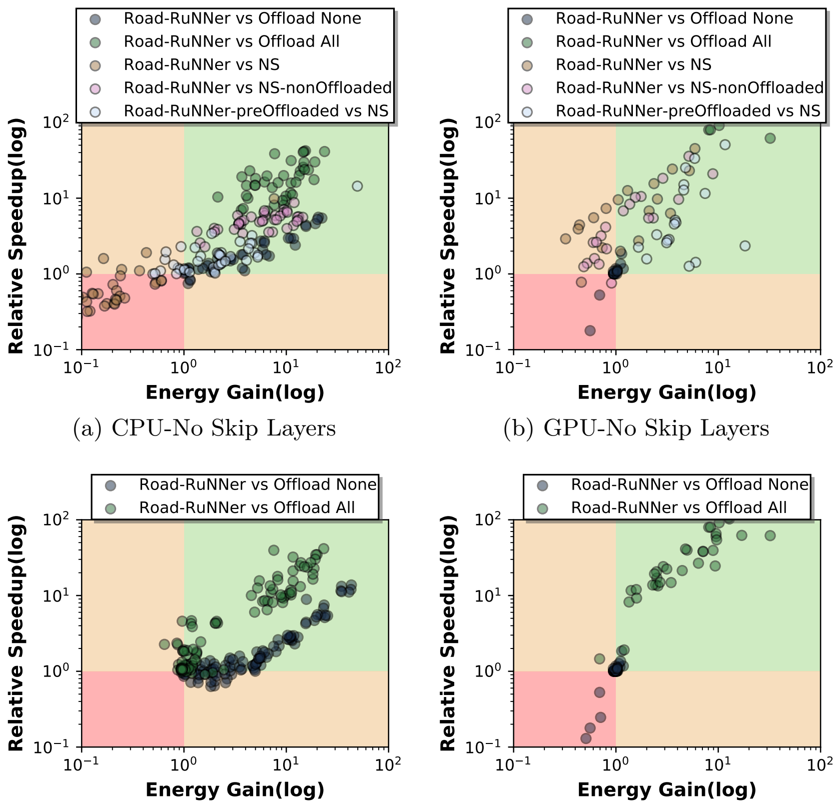
Technical Implementation: *RoaD-RuNNer* is implemented in Python programming language. All devices are interconnected through 80MB/s wireless network. We utilize the ZeroMQ messaging protocol for control and the FTP protocol for transmission of the actual data and layers. In order to overcome the architectural differences of heterogeneous CPUs all applications are integrated inside docker containers, while the corresponding GPU implementations are developed in CUDA. The collaborative filtering component used for estimating the performance and energy impact of different layers is accelerated through the use of C++ for increased performance.

Examined DNN models: We examine famous DNN architectures and known variations, i.e. AlexNet (alex), MobileNetV2 (mv2), Resnet18 (res18), Resnet34 (res34), Resnet50 (res50), Resnet101 (res101), Resnet152 (res152), VGG11, VGG13 and VGG16, which are widely used for performing object detection and image classification tasks at the edge [459, 460], over alternative input image sizes (224 × 224, 512 × 512 and 768 × 768). The input models are derived from PyTorch [142] and are integrated to `RoaD-RuNNer`.

Reference Baselines: We evaluate the impact of our approach based on three key metrics: (i) performance (ii) energy consumption and (iii) prediction accuracy of our collaborative filtering approach. We compare against various partition and offloading mechanisms. First, as a baseline we utilize two naive approaches: (i) *Offload None*, which executes all tasks locally, without offloading anything on the cloud and (ii) *Offload All*, in which all tasks are offloaded to the cloud for remote execution. Moreover, we implement from scratch and compare against a state-of-the-art resource management algorithm for DNN offloading, namely *Neurosurgeon*(NS) [450]. Since *Neurosurgeon* is designed with the assumption to operate with a priori offloaded layers on the cloud infrastructure, we also implement a version of *Neurosurgeon* with online layer offloading, namely *NS-nonOffloaded* and a version of `RoaD-RuNNer`-*preOffloaded*, where the layers are offloaded a priori to the cloud.

D.4.2. Evaluation

Performance and Energy Evaluation: In the first comparative experiment, we evaluate `RoaD-RuNNer` in terms of performance and energy consumption against the approaches presented in Section D.4.1, as illustrated in Fig. D.4.1. Given the fact that *Neurosurgeon* is designed to operate only over non-Residual Neural Networks, we utilize as benchmarks the VGG11,13,16 and AlexNet models. X axis indicates the corresponding energy gain, while Y axis denotes the relative speedup of our framework compared to other approaches for CPU (Fig.D.4.1a) and GPU (Fig.D.4.1b) execution, respectively. The output is divided in four distinct quadrants, on which `RoaD-RuNNer` does not achieve any speedup or energy gain (red), achieves either speedup only or energy gain only (orange) and achieves both speedup and energy optimization (green). We observe that *Road-RuNNer* clearly outperforms the *Offload All* and *Offload None* approaches, by leading up to 45.41× speedup, 95.84% energy reduction compared to the former and up to 6.61× performance optimization and 95.87% energy reduction compared to the latter, for CPU execution. Similar observations are derived for GPU execution. The initial version of *Neurosurgeon*, on which all layer weights are offloaded a priori, performs better in terms of performance and energy consumption, as the network overhead is a dominant factor. However, compared to the *NS-nonOffloaded*, where layer weights are offloaded dynam-



(a) CPU-No Skip Layers

(b) GPU-No Skip Layers

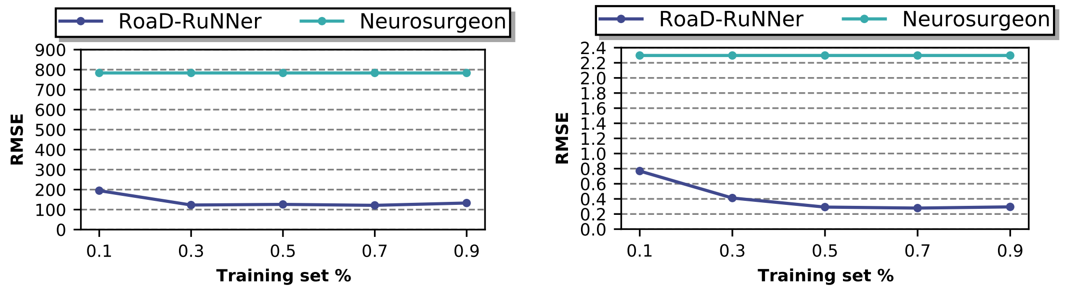
(c) CPU-Skip Layers

(d) GPU-Skip Layers

Figure D.4.1.. Performance and Energy Comparison of RoaD-RuNNer framework against other approaches for CPU and GPU nodes for alternative DNN workloads.

ically, we observe that `RoaD-RuNNer` provides on average 4.97× optimized performance and 81.85% less energy consumption for CPU, and up to 35.74× optimized performance and 88.73% less energy consumption for GPU, respectively. `RoaD-RuNNer` is designed to provide up to two partition points, in contrast to *Neurosurgeon*, on which there exists only a single breakpoint. Therefore, the latter pays the penalty of either executing locally all the resource intensive layers or offloading their weights, thus paying the network penalty. Similarly, `RoaD-RuNNer`-*preoffloaded*, where all model layers are offloaded offline, our approach outperforms *Neurosurgeon* by up to 54.09× in terms of performance and up to 58.06× in terms of energy consumption.

In contrast to *Neurosurgeon*, `RoaD-RuNNer` is designed to operate efficiently over workloads consisting of DNNs with skip layer connections. Thus, we add to the existing bench-



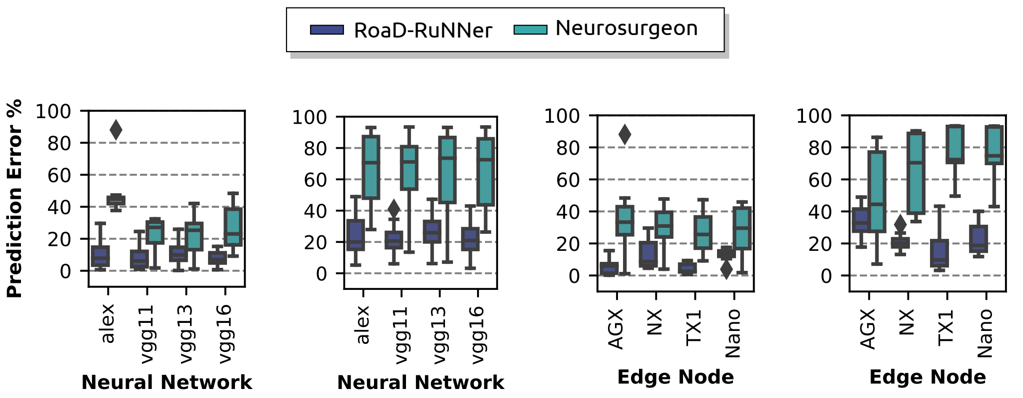
(a) Execution Time RMSE (b) Energy Consumption RMSE

Figure D.4.2.. Root Mean Square Error(RMSE) of execution time and energy consumption over alternative Collaborative Filtering fill percentages.

marks the Resnet18, Resnet34, Resnet50, Resnet101, Resnet152 and MobileNetV2 models and evaluate our approach. The evaluation is depicted in Fig. D.4.1c and Fig. D.4.1d for CPU and GPU executions, respectively. For CPU execution, our framework outperforms *Offload None*, by achieving up to 13.92× optimized performance and 46.07× less energy consumption and *Offload All* by achieving up to 45.41× optimized performance and 24.05× less energy consumption, respectively. Similarly, for GPU executions, `RoaD-RuNNer` achieves up to 1.85× speedup and up to 1.34× less energy consumption compared to *Offload None* and 112.98× speedup and 16.59× less energy consumption on average compared to *Offload All*.

Prediction Accuracy: The efficacy of our framework is directly related to the accuracy of the collaborative filtering mechanism. First, we evaluate the learning phase of the prediction mechanism in terms of Root Mean Square Error (RMSE) related to the size (percentage) of training set and compare with the *Neurosurgeon's* prediction approach. We observe that providing the 30% of our training set, the execution time and energy RMSE has converged. Thus, we set our training set size to 0.3. Compared to *Neurosurgeon*, our proposed mechanism achieves up to 6.45× and 8.48× less RMSE, for execution time and energy, respectively. *Neurosurgeon* displays linear behavior, as it utilizes 100% of the dataset during the training phase.

Next, as illustrated in Fig. D.4.3, we compare the execution time accuracy and energy prediction accuracy of `RoaD-RuNNer` (Fig. D.4.3a, D.4.3b) and *Neurosurgeon* per deep learning model and Edge node (Fig. D.4.3c, D.4.3d), respectively. Our framework achieves 68.01% less execution time prediction error and 63.8% less energy prediction error per DNN on average, while we achieve up to 69.6% less execution time prediction error and up to 34.9% less energy prediction error per edge device. In contrast to our prediction mechanism, *Neurosurgeon* is based on linear and logarithmic regression, making it im-



(a) Latency per DNN model (b) Energy per DNN model (c) Latency per Edge Device (d) Energy per Edge Device

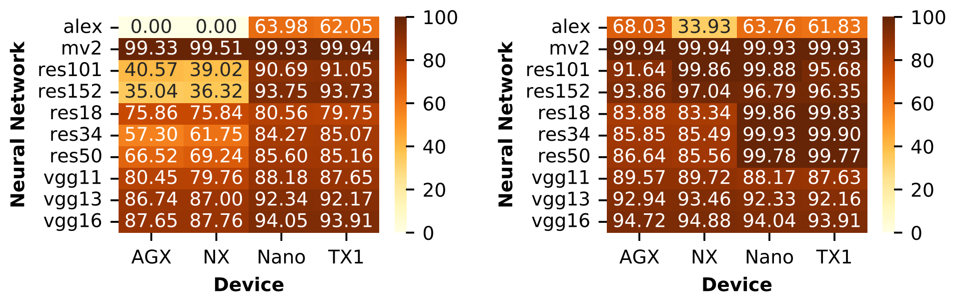
Figure D.4.3.. Execution Latency and Energy Consumption Prediction Accuracy for alternative DNN workloads and Edge nodes.

possible to provide high accuracy (or low error) and capture non-linear behaviors in the system, thus leading to high RMSE.

DNN Offload Analysis: Further discussion can be conducted on the decision making of `RoaD-RuNNer`, in order to achieve latency and energy optimization objectives. Thus, in Fig. D.4.4 the layer offloading percentage in terms of the number of layers offloaded for each model and edge node is depicted, in order to achieve each optimization objective, respectively. First, as depicted in Fig. D.4.4a, we see that the more powerful devices (AGX, NX) offload less computation for remote execution, compared to less powerful devices (Nano, TX1). More specifically, for AGX and NX the 58.7% of the target DNN is offloaded on average, opposed to Nano and TX1, where the 87.1% is offloaded. Similar observations are extracted for the energy optimization objective, as shown in Fig. D.4.4b. Furthermore, having execution latency as the optimization objective, the 74.45% of layers is offloaded on average, while for the energy optimization objective the corresponding percentage rises to 90.1%. This is due to the fact that the network imposes high latency overhead, thus making the data and layer transmission prohibitive in order to meet latency optimization objectives.

D.5. Conclusion

This work presents `RoaD-RuNNer`, a novel resource management framework for DNN par-



(a) Latency Objective (b) Energy Objective

Figure D.4.4.. DNN percentage offloading over heterogeneous Edge nodes for latency and energy optimization objectives.

titioning and offloading over heterogeneous CPU/GPU edge computing systems. Our framework strongly leverages collaborative filtering techniques to estimate performance and energy requirements of individual DNN layers over heterogeneous devices. By aggregating this information, it specifies a set of Pareto optimal DNN partitioning schemes that trade-off between performance and energy consumption. Our approach outperforms existing state-of-the-art approaches by achieving 9.58× speedup on average and up to 88.73% less energy consumption and high prediction accuracy by limiting the prediction error down to 3.19% and 0.18% for latency and energy, respectively.

Abbreviations & Acronyms

AI	Artificial Intelligence
AWS	Amazon Web Services
BE	Best Effort
CNN	Convolutional Neural Networks
CPU	Central Processing Unit
DC	Data Center
DCM	DC Monitoring
DLR	Deep Reinforcement Learning
DNN	Deep Neural Network
DPU	Data Processing Unit
DRAM	Dynamic Random-Access Memory
FaaS	Function-as-a-Service
FC	Fully-connected
FPGA	Field Programmable Gate Array
GCP	Google Cloud Platform
GPU	Graphics Processing Unit

HW	Hardware
ICT	Information and Communication Technology
ML	Machine Learning
MLaaS	ML-as-a-Service
KPI	Key Performance Indicator
LC	Latency Critical
LLC	Last-Level Cache
LSTM	Long Short-Term Memory
PCM	Performance Counter Monitoring
PMU	Performance Monitoring Units
PUE	Power Usage Effectiveness
QoS	Quality of Service
SLA	Service Level Agreements
SW	Software
TCO	Total Cost of Ownership
TPU	Tensor Processing Unit
VM	Virtual Machine
XaaS	Everything-as-a-Service

Γλωσσάρι

Artifical Intelligence	Τεχνητή Νοημοσύνη. Ικανότητα μιας μηχανής να αναπαράγει τις γνωστικές λειτουργίες ενός ανθρώπου.
Cloud Computing	Υπολογιστικό Νέφος. Σύνολο από υπολογιστικούς πόρους διακομιστών που διατίθενται κατάπαίτηση σε τελικούς χρήστες.
Cloud Provider	Πάροχος Υπηρεσιών Υπολογιστικού Νέφους. Εταιρεία που προσφέρει υπηρεσίες μίσθωσης πόρων στο Υπολογιστικό Νέφος.
Datacenter	Κέντρο Δεδομένων. Κτίριο ή χώρος μέσα σε ένα κτίριο στο οποίο φιλοξενούνται διακομιστές, αλλά και τηλεπικοινωνιακές και αποθηκευτικές υποδομές.
Design Space Exploration (DSE)	Εξερεύνηση του χώρου σχεδίασης.
Green Computing	Πράσινη Πληροφορική. Εργαλεία, υπηρεσίες και τεχνολογίες της πληροφορικής και των επικοινωνιών που συνεισφέρουν στην προστασία και την αποκατάσταση του φυσικού περιβάλλοντος.
Hyperscale Datacenter	Υπερκλιμακούμενο Κέντρο Δεδομένων
Long short-term Memory Networks	Δίκτυα μακράς βραχυπρόθεσμης μνήμης. Μορφή τεχνητών ανατροφοδοτούμενων νευρωνικών δικτύων που χρησιμοποιούνται στον τομέα της βαθειάς μάθησης.
Machine Learning	Μηχανική Μάθηση. Μέθοδος ανάλυσης δεδομένων που αυτοματοποιεί την ανάπτυξη αναλυτικών μοντέλων.

Multi-tenancy	Πολλαπλή μίσθωση. Η συντοποθέτηση εφαρμογών από διαφορετικούς χρήστες σε κοινούς διακομιστές σε περιβάλλοντα Υπολογιστικού Νέφους, προκειμένου να επιτευχθεί αύξηση της χρησιμοποίησης των πόρων του κέντρου δεδομένων και μείωση του κόστους λειτουργίας.
Pareto front	Μέτωπο Pareto. Σε προβλήματα βελτιστοποίησης με πολλαπλά κριτήρια, οι Pareto βέλτιστες λύσεις είναι αυτές για τις όποιες η βελτιστοποίηση του ενός κριτηρίου μπορεί να επιτευχθεί μόνο αν χειροτερεύσει τουλάχιστον ένα άλλο κριτήριο.
Resource Disaggregation	Αποσύνθεση Υπολογιστικών Πόρων. Σε σύγχρονα συστήματα υπολογιστικού νέφους, οι διαθέσιμοι υπολογιστικοί πόροι οργανώνονται σε ομογενείς ομάδες, οι οποίοι μπορούν να συντεθούν δυναμικά και να χρησιμοποιούνται μέσω πρωτοκόλλων δικτύου.
Server	Διακομιστής. Υλικό ή/και λογισμικό που παρέχει υπολογιστικούς πόρους και αναλαμβάνει την παροχή διάφορων υπηρεσιών, εξυπηρετώντας αιτήσεις άλλων προγραμμάτων.

Publications

Journals

1. Achilleas Tzenetopoulos, Dimosthenis Masouros, Nikolaos Kapsoulis, Antonios Litke, Dimitrios Soudris, Theodora A. Varvarigou: HLF-Kubed: Blockchain-Based Resource Monitoring for Edge Clusters. Ledger 7 (2022)

2. Dimosthenis Masouros, Sotirios Xydis, Dimitrios Soudris: Rusty: Runtime Interference-Aware Predictive Monitoring for Modern Multi- Tenant Systems. IEEE Transactions on Parallel and Distributed Systems 32(1): 184-198 (2021)

3. Farzad Samie, Vasileios Tsoutsouras, Dimosthenis Masouros, Lars Bauer, Dimitrios Soudris, Jörg Henkel: Fast Operation Mode Selection for Highly Efficient IoT Edge Devices. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. 39(3): 572-584 (2020)

4. Dimosthenis Masouros, Sotirios Xydis, Dimitrios Soudris: Rusty: Runtime System Predictability Leveraging LSTM Neural Networks. IEEE Comput. Archit. Lett. 18(2): 103-106 (2019)

5. Vasileios Tsoutsouras, Iraklis Anagnostopoulos, Dimosthenis Masouros, Dimitrios Soudris: A Hierarchical Distributed Runtime Resource Management Scheme for NoC-Based Many-Cores. ACM Trans. Embed. Comput. Syst. 17(3): 65:1-65:26 (2018)

6. Vasileios Tsoutsouras, Dimosthenis Masouros, Sotirios Xydis, Dimitrios Soudris: SoftRM: Self-Organized Fault-Tolerant Resource Management for Failure Detection and Recovery in NoC Based Many-Cores. ACM Trans. Embed. Comput. Syst. 16(5s): 144:1-144:19 (2017)

Conferences

1. Dimosthenis Masouros, Christian Pinto, Michele Gazzetti, Sotirios Xydis, Dimitrios Soudris: Adrias: Interference-Aware Memory Orchestration for Disaggregated Cloud Infrastructures. HPCA 2023: 855-869

2. Aggelos Ferikoglou, Argyris Kokkinis, Dimitrios Danopoulos, Ioannis Oroutzoglou, Anastassios Nanos, Stathis Karanastasis, Márton Sipos, Javad Fadaie Ghotbi, Juan Jose Vegas Olmos, Dimosthenis Masouros, Kostas Siozios: The SERRANO platform: Stepping towards seamless application development & deployment in the heterogeneous edge-cloud continuum. DATE 2023: 1-4

3. Andreas Kosmas Kakolyris, Manolis Katsaragakis, Dimosthenis Masouros, Dimitrios Soudris: RoaD-RuNNer: Collaborative DNN partitioning and offloading on heterogeneous edge systems. DATE 2023: 1-6

4. Manolis Katsaragakis, Konstantinos Stavrakakis, Dimosthenis Masouros, Lazaros Papadopoulos and Dimitrios Soudris: Adjacent LSTM-based Page Scheduling for Hybrid DRAM/NVM Memory Systems. Accepted at PARMA-DITAM@HiPEAC 2023

5. Ioannis Fakinos, Achilleas Tzenetopoulos, Dimosthenis Masouros, Sotirios Xydis, Dimitrios Soudris: Sequence Clock: A Dynamic Resource Orchestrator for Serverless Architectures. CLOUD 2022: 81-90

6. Achilleas Tzenetopoulos, Dimosthenis Masouros, Konstantina Koliogeorgi, Sotirios Xydis, Dimitrios Soudris, Antony Chazapis, Christos Kozanitis, Angelos Bilas, Christian Pinto, Huy-Nam Nguyen, Stelios Louloudakis, Georgios Gardikis, George Vamvakas, Michelle Aubrun, Christi Symeonidou, Vassilis Spitadakis, Konstantinos Xylogiannopoulos, Bernhard Peischl, Tahir Emre Kalayci, Alexander Stocker, Jean-Thomas Acquaviva: EVOLVE: Towards Converging Big-Data, High-Performance and Cloud-Computing Worlds. DATE 2022: 975-980

7. Antonis Karteris, Manolis Katsaragakis, Dimosthenis Masouros, Dimitrios Soudris: SGRM: Stackelberg Game-Based Resource Management for Edge Computing Systems. DATE 2022: 1203-1208

8. Ioannis Oroutzoglou, Argyris Kokkinis, Aggelos Ferikoglou, Dimitrios Danopoulos, Dimosthenis Masouros, Kostas Siozios: Optimizing Savitzky-Golay Filter on GPU and FPGA Accelerators for Financial Applications. MOCAST 2022: 1-4

9. Argyris Kokkinis, Aggelos Ferikoglou, Ioannis Oroutzoglou, Dimitrios Danopoulos, Dimosthenis Masouros, Kostas Siozios: HW/SW Acceleration of Multiple Workloads Within the SERRANO's Computing Continuum - Invited Paper. SAMOS 2022: 394-405

10. Dimitrios Danopoulos, Ioannis Stamoulias, George Lentaris, Dimosthenis Masouros, Ioannis Kanaropoulos, Andreas Kosmas Kakolyris, Dimitrios Soudris: LSTM Acceleration with FPGA and GPU Devices for Edge Computing Applications in B5G MEC. SAMOS 2022: 406-419

11. Petros Vavaroutsos, Ioannis Oroutzoglou, Dimosthenis Masouros, Dimitrios Soudris: Towards making the most of NLP-based device mapping optimization for OpenCL kernels. COINS 2022: 1-6

12. Dimitra Nikitopoulou, Dimosthenis Masouros, Sotirios Xydis, Dimitrios Soudris: Performance Analysis and Auto-tuning for SPARK in-memory analytics. DATE 2021: 76-81

13. Konstantina Koliogeorgi, Fekhr Eddine Keddous, Dimosthenis Masouros, Antony Chazapis, Michelle Aubrun, Sotirios Xydis, Angelos Bilas, Romain Hugues, Jean-Thomas Acquaviva, Huy-Nam Nguyen, Dimitrios Soudris: FPGA acceleration in EVOLVE's Converged Cloud-HPC Infrastructure. FPL 2021: 376-377

14. Aggelos Ferikoglou, Dimosthenis Masouros, Achilleas Tzenetopoulos, Sotirios Xydis, Dimitrios Soudris: Resource Aware GPU Scheduling in Kubernetes Infrastructure. PARMA-DITAM@HiPEAC 2021: 4:1-4:12

15. Aristotelis Kretsis, Panagiotis Kokkinos, Polyzois Soumplis, Juan Jose Vegas Olmos, Marcell Fehér, Márton Sipos, Daniel E Lucani, Dmitry Khabi, Dimosthenis Masouros, Kostas Siozios, Paraskevas Bourgos, Sofia Tsekeridou, Ferad Zyulkyarov, Efstathios Karanastasis, Efthymios Chondrogiannis, Vassiliki Andronikou, Aitor Fernandez Gomez, Silviu Panica, Gabriel Iuhasz, Anastassios Nanos, Charalampos Chalios, Manos Varvarigos: SERRANO: Transparent Application Deployment in a Secure, Accelerated and Cognitive Cloud Continuum. 2021 IEEE International Mediterranean Conference on Communications and Networking (MeditCom)

16. Achilleas Tzenetopoulos, Evangelos Apostolakis, Aphrodite Tzomaka, Christos Papakostopoulos, Konstantinos Stavrakakis, Manolis Katsaragakis, Ioannis Oroutzoglou, Dimosthenis Masouros, Sotirios Xydis, Dimitrios Soudris: FaaS and Curious: Performance Implications of Serverless Functions on Edge Computing Platforms. ISC Workshops 2021: 428-438

17. Ioannis Oroutzoglou, Dimosthenis Masouros, Konstantina Koliogeorgi, Sotirios Xydis, Dimitrios Soudris: Exploration of GPU sharing policies under GEMM workloads. SCOPES 2020: 66-69

18. Argyris Kokkinis, Aggelos Ferikoglou, Dimitrios Danopoulos, Dimosthenis Masouros, Kostas Siozios: Leveraging HW Approximation for Exploiting Performance-Energy Trade-offs Within the Edge-Cloud Computing Continuum. ISC Workshops 2021: 406-415

19. Achilleas Tzenetopoulos, Dimosthenis Masouros, Sotirios Xydis, Dimitrios Soudris: Interference-Aware Orchestration in Kubernetes. ISC Workshops 2020: 321-330

20. Dimosthenis Masouros, Konstantina Koliogeorgi, Georgios Zervakis, Alexandra Kosvyra, Achilleas Chytas, Sotirios Xydis, Ioanna Chouvarda, Dimitrios Soudris: Co-design Implications of Cost-effective On-demand Acceleration for Cloud Healthcare Analytics: The AEGLE approach. DATE 2019: 622-625

21. Manolis Katsaragakis, Dimosthenis Masouros, Vasileios Tsoutsouras, Farzad Samie, Lars Bauer, Jörg Henkel, Dimitrios Soudris: DMRM: Distributed Market-Based Resource Management of Edge Computing Systems. DATE 2019: 1391-1396

22. Konstantina Koliogeorgi, Dimosthenis Masouros, Georgios Zervakis, Sotirios Xydis, Tobias Becker, Georgi Gaydadjiev, Dimitrios Soudris: AEGLE's Cloud Infrastructure for Resource Monitoring and Containerized Accelerated Analytics. ISVLSI 2017: 362-367

23. Dimosthenis Masouros, Ioannis Bakolas, Vasileios Tsoutsouras, Kostas Siozios, Dimitrios Soudris: From edge to cloud: Design and implementation of a healthcare Internet of Things infrastructure. PATMOS 2017: 1-6

Bibliography

[1] S. Dang, O. Amin, B. Shihada, and M.-S. Alouini, “What should 6g be?,” *Nature Electronics*, vol. 3, no. 1, pp. 20–29, 2020.

[2] B. Varghese, N. Wang, S. Barbhuiya, P. Kilpatrick, and D. S. Nikolopoulos, “Challenges and opportunities in edge computing,” in *2016 IEEE International Conference on Smart Cloud (SmartCloud)*, pp. 20–26, IEEE, 2016.

[3] A. Labrinidis and H. V. Jagadish, “Challenges and opportunities with big data,” *Proceedings of the VLDB Endowment*, vol. 5, no. 12, pp. 2032–2033, 2012.

[4] D. C. Nguyen, M. Ding, P. N. Pathirana, A. Seneviratne, and A. Y. Zomaya, “Federated learning for covid-19 detection with generative adversarial networks in edge cloud computing,” *IEEE Internet of Things Journal*, 2021.

[5] J. Guo, Z. Chang, S. Wang, H. Ding, Y. Feng, L. Mao, and Y. Bao, “Who limits the resource efficiency of my datacenter: an analysis of alibaba datacenter traces,” in *Proceedings of the International Symposium on Quality of Service*, p. 39, ACM, 2019.

[6] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch, “Heterogeneity and dynamicity of clouds at scale: Google trace analysis,” in *Proceedings of the third ACM symposium on cloud computing*, pp. 1–13, 2012.

[7] R. Bianchini, M. Fontoura, E. Cortez, A. Bonde, A. Muzio, A.-M. Constantin, T. Moscibroda, G. Magalhaes, G. Bablani, and M. Russinovich, “Toward ml-centric cloud platforms,” *Communications of the ACM*, vol. 63, no. 2, pp. 50–59, 2020.

[8] C. V. networking Index, “Forecast and methodology, 2016-2021, white paper,” *San Jose, CA, USA*, vol. 1, 2016.

[9] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, “Borg, omega, and kubernetes,” *Queue*, vol. 14, pp. 70–93, Jan. 2016.

[10] *Amazon Web Services.*
https://aws.amazon.com/.

[11] *Google Cloud Platform.*
https://cloud.google.com/.

[12] *Microsoft Azure Cloud Computing Platform & Services.*
https://azure.microsoft.com.

[13] C. Kozyrakis, A. Kansal, S. Sankar, and K. Vaid, “Server engineering insights for

large-scale online services," *IEEE micro*, vol. 30, no. 4, pp. 8–19, 2010.

[14] S. Kanev, J. P. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G.-Y. Wei, and D. Brooks, "Profiling a warehouse-scale computer," *ACM SIGARCH Computer Architecture News*, vol. 43, no. 3, pp. 158–169, 2016.

[15] A. S. Dhodapkar and J. E. Smith, "Comparing program phase detection techniques," in *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, p. 217, IEEE Computer Society, 2003.

[16] C. Isci and M. Martonosi, "Phase characterization for power: Evaluating control-flow-based and event-counter-based techniques," in *The Twelfth International Symposium on High-Performance Computer Architecture, 2006.*, pp. 121–132, IEEE, 2006.

[17] R. S. Kannan, M. Laurenzano, J. Ahn, J. Mars, and L. Tang, "Caliper: Interference estimator for multi-tenant environments sharing architectural resources," *ACM Trans. Archit. Code Optim.*, vol. 16, pp. 22:1–22:25, June 2019.

[18] H. Yang, A. Breslow, J. Mars, and L. Tang, "Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers," *ACM SIGARCH Computer Architecture News*, vol. 41, no. 3, pp. 607–618, 2013.

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

[20] *Processor Counter Monitor (PCM).*
https://github.com/opcm/pcm.

[21] C. Delimitrou and C. Kozyrakis, "ibench: Quantifying interference for datacenter applications," in *2013 IEEE international symposium on workload characterization (IISWC)*, pp. 23–33, IEEE, 2013.

[22] J. L. Henning, "Spec cpu2006 benchmark descriptions," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 4, pp. 1–17, 2006.

[23] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, *et al.*, "Scikit-learn: Machine learning in python," *Journal of machine learning research*, vol. 12, no. Oct, pp. 2825–2830, 2011.

[24] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "Clearing the clouds: A study of emerging scale-out workloads on modern hardware," *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2012.

[25] MarketsandMarkets, *Cloud Computing Market by Service, Deployment Model, Organization Size, Vertical And Region - Global Forecast to 2026*, 2021.

[26] X. Sun, C. Hu, R. Yang, P. Garraghan, T. Wo, J. Xu, J. Zhu, and C. Li, "Rose: Cluster resource scheduling via speculative over-subscription," in *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pp. 949–960, IEEE, 2018.

[27] C. Delimitrou and C. Kozyrakis, "Quasar: resource-efficient and qos-aware cluster management," in *ACM SIGARCH Computer Architecture News*, vol. 42, pp. 127–144, ACM, ACM New York, NY, USA, 2014.

[28] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis, "Improving resource efficiency at scale with heracles," *ACM Transactions on Computer Systems (TOCS)*, vol. 34, no. 2, pp. 1–33, 2016.

[29] D. Masouros, S. Xydis, and D. Soudris, "Rusty: Runtime interference-aware predictive monitoring for modern multi-tenant systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, pp. 184–198, jan 2020.

[30] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis, "Heracles: Improving resource efficiency at scale," in *ACM SIGARCH Computer Architecture News*, vol. 43, pp. 450–462, ACM, 2015.

[31] R. S. Kannan, L. Subramanian, A. Raju, J. Ahn, J. Mars, and L. Tang, "Grandslam: Guaranteeing slas for jobs in microservices execution frameworks," in *Proceedings of the Fourteenth EuroSys Conference 2019*, p. 34, ACM, 2019.

[32] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at google with borg," in *Proceedings of the Tenth European Conference on Computer Systems*, pp. 1–17, 2015.

[33] S. Shravan, J. Lakshmi, and N. Bisht, "Towards improving data center utilisation by reducing fragmentation," in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, pp. 941–945, IEEE, 2018.

[34] X. Ke, C. Guo, S. Ji, S. Bergsma, Z. Hu, and L. Guo, "Fundy: A scalable and extensible resource manager for cloud resources," in *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*, pp. 540–550, IEEE, 2021.

[35] P. Koutsovasilis, M. Gazzetti, and C. Pinto, "A holistic system software integration of disaggregated memory for next-generation cloud infrastructures," in *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pp. 576–585, 2021.

[36] G. Ayers, N. P. Nagendra, D. I. August, H. K. Cho, S. Kanev, C. Kozyrakis, T. Krishnamurthy, H. Litz, T. Moseley, and P. Ranganathan, "Asmdb: understanding and mitigating front-end stalls in warehouse-scale computers," in *Proceedings of the 46th International Symposium on Computer Architecture*, pp. 462–473, 2019.

[37] M. Ferdman, C. Kaynak, and B. Falsafi, "Proactive instruction fetch," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, (New York, NY, USA), pp. 152–162, ACM, 2011.

[38] H. Li, D. S. Berger, L. Hsu, D. Ernst, P. Zardoshti, S. Novakovic, M. Shah, S. Rajadnya, S. Lee, I. Agarwal, *et al.*, "Pond: Cxl-based memory pooling systems for cloud platforms," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pp. 574–587, 2023.

[39] D. S. Berger, D. Ernst, H. Li, P. Zardoshti, M. Shah, S. Rajadnya, S. Lee, L. Hsu,

I. Agarwal, M. D. Hill, *et al.*, “Design tradeoffs in cxl-based memory pools for public cloud platforms,” *IEEE Micro*, vol. 43, no. 2, pp. 30–38, 2023.

[40] P. Duraisamy, W. Xu, S. Hare, R. Rajwar, D. Culler, Z. Xu, J. Fan, C. Kennelly, B. McCloskey, D. Mijailovic, *et al.*, “Towards an adaptable systems architecture for memory tiering at warehouse-scale,” in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pp. 727–741, 2023.

[41] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang, “The hibench benchmark suite: Characterization of the mapreduce-based data analysis,” in *2010 IEEE 26th International Conference on Data Engineering Workshops (ICDEW 2010)*, pp. 41–51, IEEE, 2010.

[42] J. Carlson, *Redis in action.*
Simon and Schuster, 2013.

[43] B. Fitzpatrick, “Distributed caching with memcached,” *Linux journal*, vol. 2004, no. 124, p. 5, 2004.

[44] C. Pinto, D. Syrivelis, M. Gazzetti, P. Koutsovasilis, A. Reale, K. Katrinis, and H. P. Hofstee, “Thymesisflow: a software-defined, hw/sw co-designed interconnect stack for rack-scale memory disaggregation,” in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 868–880, IEEE, 2020.

[45] “Data never sleeps..” https://www.domo.com/solution/data-never-sleeps-6.
Accessed: 08-11-2021.

[46] J. Dean and S. Ghemawat, “Mapreduce: simplified data processing on large clusters,” *Communications of the ACM*, vol. 51, pp. 107–113, Jan. 2008.

[47] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, “The hadoop distributed file system,” in *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*, pp. 1–10, Ieee, 2010.

[48] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, *et al.*, “Apache spark: a unified engine for big data processing,” *Communications of the ACM*, vol. 59, no. 11, pp. 56–65, 2016.

[49] “Apache spark as a service.” https://www.databricks.com/glossary/apache-spark-as-a-service.
Accessed: 19-11-2022.

[50] “Spark configuration.” https://spark.apache.org/docs/latest/configuration.html.
Accessed: 17-11-2022.

[51] “Tuning spark.” https://spark.apache.org/docs/latest/tuning.html.
Accessed: 24-06-2021.

[52] D. L. Hahs-Vaughn and R. G. Lomax, *Statistical concepts: A second course.*
Routledge, 2020.

[53] S. Venkataraman, Z. Yang, M. Franklin, B. Recht, and I. Stoica, “Ernest: efficient performance prediction for large-scale advanced analytics,” in *13th {USENIX$}$ Symposium on Networked Systems Design and Implementation (${$NSDI$}$ 16)*,

pp. 363–378, 2016.

[54] Z. Hu, D. Li, D. Zhang, Y. Zhang, and B. Peng, "Optimizing resource allocation for data-parallel jobs via gcn-based prediction," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 9, pp. 2188–2201, 2021.

[55] Z. Yu, Z. Bei, and X. Qian, "Datasize-aware high dimensional configurations auto-tuning of in-memory cluster computing," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 564–577, 2018.

[56] D. Nikitopoulou, D. Masouros, S. Xydis, and D. Soudris, "Performance analysis and auto-tuning for spark in-memory analytics," in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 76–81, IEEE, 2021.

[57] J. Xin, K. Hwang, and Z. Yu, "Locat: Low-overhead online configuration auto-tuning of spark sql applications," in *Proceedings of the 2022 International Conference on Management of Data*, SIGMOD '22, (New York, NY, USA), p. 674–684, Association for Computing Machinery, 2022.

[58] Y. Censor, "Pareto optimality in multiobjective problems," *Applied Mathematics and Optimization*, vol. 4, no. 1, pp. 41–59, 1977.

[59] M. Kumar, D. Husain, N. Upreti, D. Gupta, *et al.*, "Genetic algorithm: Review and application," *Available at SSRN 3529843*, 2010.

[60] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: Nsga-ii," *IEEE transactions on evolutionary computation*, vol. 6, no. 2, pp. 182–197, 2002.

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

[62] Y. Li and B. C. Lee, "Phronesis: Efficient performance modeling for high-dimensional configuration tuning," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 19, no. 4, pp. 1–26, 2022.

[63] N. Luo, Z. Yu, Z. Bei, C. Xu, C. Jiang, and L. Lin, "Performance modeling for spark using svm," in *2016 7th International Conference on Cloud Computing and Big Data (CCBD)*, pp. 127–131, IEEE, 2016.

[64] C. Cummins, P. Petoumenos, Z. Wang, and H. Leather, "End-to-end deep learning of optimization heuristics," in *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pp. 219–232, IEEE, 2017.

[65] G. Briscoe and A. Marinos, "Digital ecosystems in the clouds: towards community cloud computing," in *2009 3rd IEEE international conference on digital ecosystems and technologies*, pp. 103–108, IEEE, 2009.

[66] M. Al-Ruithe, E. Benkhelifa, and K. Hameed, "Key issues for embracing the cloud computing to adopt a digital transformation: A study of saudi public sector," *Procedia computer science*, vol. 130, pp. 1037–1043, 2018.

[67] S. Rendle, D. Fetterly, E. J. Shekita, and B.-y. Su, "Robust large-scale machine learning in the cloud," in *Proceedings of the 22nd ACM SIGKDD International*

Conference on Knowledge Discovery and Data Mining, pp. 1125–1134, 2016.

[68] M. Díaz, C. Martín, and B. Rubio, “State-of-the-art, challenges, and open issues in the integration of internet of things and cloud computing,” *Journal of Network and Computer applications*, vol. 67, pp. 99–117, 2016.

[69] A. Botta, W. De Donato, V. Persico, and A. Pescapé, “Integration of cloud computing and internet of things: a survey,” *Future generation computer systems*, vol. 56, pp. 684–700, 2016.

[70] M. Aazam, I. Khan, A. A. Alsaffar, and E.-N. Huh, “Cloud of things: Integrating internet of things and cloud computing and the issues involved,” in *Proceedings of 2014 11th International Bhurban Conference on Applied Sciences & Technology (IBCAST) Islamabad, Pakistan, 14th-18th January, 2014*, pp. 414–419, IEEE, 2014.

[71] S. Mathew and J. Varia, “Overview of amazon web services,” *Amazon Whitepapers*, 2014.

[72] L. Qian, Z. Luo, Y. Du, and L. Guo, “Cloud computing: An overview,” in *IEEE international conference on cloud computing*, pp. 626–631, Springer, 2009.

[73] J. Wang, J. Pan, F. Esposito, P. Calyam, Z. Yang, and P. Mohapatra, “Edge cloud offloading algorithms: Issues, methods, and perspectives,” *ACM Computing Surveys (CSUR)*, vol. 52, no. 1, pp. 1–23, 2019.

[74] M. Ribeiro, K. Grolinger, and M. A. Capretz, “Mlaas: Machine learning as a service,” in *2015 IEEE 14th International Conference on Machine Learning and Applications (ICMLA)*, pp. 896–902, IEEE, 2015.

[75] Q. Weng, W. Xiao, Y. Yu, W. Wang, C. Wang, J. He, Y. Li, L. Zhang, W. Lin, and Y. Ding, “{MLaaS$}$ in the wild: Workload analysis and scheduling in ${$large-scale$}$ heterogeneous ${$gpu$}$ clusters,” in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pp. 945–960, 2022.

[76] L. A. Barroso and U. Hölzle, “The datacenter as a computer: An introduction to the design of warehouse-scale machines,” *Synthesis lectures on computer architecture*, vol. 4, no. 1, pp. 1–108, 2009.

[77] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. Elsevier, 2011.

[78] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger, “Dark silicon and the end of multicore scaling,” in *2011 38th Annual international symposium on computer architecture (ISCA)*, pp. 365–376, IEEE, 2011.

[79] G. E. Moore *et al.*, “Cramming more components onto integrated circuits,” 1965.

[80] M. Aldossary and K. Djemame, “Energy consumption-based pricing model for cloud computing,” in *32nd UK Performance Engineering Workshop*, pp. 16–27, University of Bradford, 2016.

[81] A. Mazrekaj, I. Shabani, and B. Sejdiu, “Pricing schemes in cloud computing: an overview,” *International Journal of Advanced Computer Science and Applications*, vol. 7, no. 2, pp. 80–86, 2016.

[82] S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Serverless computation with {OpenLambda$}$," in *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*, 2016.

[83] I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, A. Slominski, *et al.*, "Serverless computing: Current trends and open problems," in *Research advances in cloud computing*, pp. 1–20, Springer, 2017.

[84] Y. Xing and Y. Zhan, "Virtualization and cloud computing," in *Future wireless networks and information systems*, pp. 305–312, Springer, 2012.

[85] E. Keller, J. Szefer, J. Rexford, and R. B. Lee, "Nohype: virtualized cloud infrastructure without the virtualization," in *Proceedings of the 37th annual international symposium on Computer architecture*, pp. 350–361, 2010.

[86] S. Govindan, J. Liu, A. Kansal, and A. Sivasubramaniam, "Cuanta: quantifying effects of shared on-chip resource interference for consolidated virtual machines," in *Proceedings of the 2nd ACM Symposium on Cloud Computing*, pp. 1–14, 2011.

[87] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa, "Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations," in *Proceedings of the 44th annual IEEE/ACM International Symposium on Microarchitecture*, pp. 248–259, ACM, 2011.

[88] X. Pu, L. Liu, Y. Mei, S. Sivathanu, Y. Koh, and C. Pu, "Understanding performance interference of i/o workload in virtualized cloud environments," in *2010 IEEE 3rd International Conference on Cloud Computing*, pp. 51–58, IEEE, 2010.

[89] S. Chen, C. Delimitrou, and J. F. Martínez, "Parties: Qos-aware resource partitioning for multiple interactive services," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 107–120, 2019.

[90] R. Lin, Y. Cheng, M. De Andrade, L. Wosinska, and J. Chen, "Disaggregated data centers: Challenges and trade-offs," *IEEE Communications Magazine*, vol. 58, no. 2, pp. 20–26, 2020.

[91] S.-Y. Tsai, Y. Shan, and Y. Zhang, "Disaggregating persistent memory and controlling them remotely: An exploration of passive disaggregated {Key-Value$}$ stores," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pp. 33–48, 2020.

[92] S. Angel, M. Nanavati, and S. Sen, "Disaggregation and the application," in *12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 20)*, 2020.

[93] A. Thusoo, Z. Shao, S. Anthony, D. Borthakur, N. Jain, J. Sen Sarma, R. Murthy, and H. Liu, "Data warehousing and analytics infrastructure at facebook," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pp. 1013–1020, 2010.

[94] C. L. Abad, Y. Lu, and R. H. Campbell, "Dare: Adaptive data replication for efficient cluster scheduling," in *2011 IEEE international conference on cluster computing*,

pp. 159–168, Ieee, 2011.

[95] Y. Mei, L. Cheng, V. Talwar, M. Y. Levin, G. Jacques-Silva, N. Simha, A. Banerjee, B. Smith, T. Williamson, S. Yilmaz, *et al.*, "Turbine: Facebook's service management platform for stream processing," in *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pp. 1591–1602, IEEE, 2020.

[96] J. Sampé, G. Vernik, M. Sánchez-Artigas, and P. García-López, "Serverless data analytics in the ibm cloud," in *Proceedings of the 19th International Middleware Conference Industry*, pp. 1–8, 2018.

[97] O. Hadary, L. Marshall, I. Menache, A. Pan, E. E. Greeff, D. Dion, S. Dorminey, S. Joshi, Y. Chen, M. Russinovich, *et al.*, "Protean:{VM$}$ allocation service at scale," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pp. 845–861, 2020.

[98] A. Newell, D. Skarlatos, J. Fan, P. Kumar, M. Khutornenko, M. Pundir, Y. Zhang, M. Zhang, Y. Liu, L. Le, *et al.*, "Ras: Continuously optimized region-wide datacenter resource allocation," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pp. 505–520, 2021.

[99] C. Tang, K. Yu, K. Veeraraghavan, J. Kaldor, S. Michelson, T. Kooburat, A. Anbudurai, M. Clark, K. Gogia, L. Cheng, *et al.*, "Twine: A unified cluster management system for shared infrastructure," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pp. 787–803, 2020.

[100] E. Boutin, J. Ekanayake, W. Lin, B. Shi, J. Zhou, Z. Qian, M. Wu, and L. Zhou, "Apollo: Scalable and coordinated scheduling for {Cloud-Scale$}$ computing," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pp. 285–300, 2014.

[101] Z. Zhang, C. Li, Y. Tao, R. Yang, H. Tang, and J. Xu, "Fuxi: a fault-tolerant resource management and job scheduling system at internet scale," in *Proceedings of the VLDB Endowment*, vol. 7, pp. 1393–1404, VLDB Endowment Inc., 2014.

[102] A. Beloglazov and R. Buyya, "Adaptive threshold-based approach for energy-efficient consolidation of virtual machines in cloud data centers.," *MGC Middleware*, vol. 4, no. 10.1145, pp. 1890799–803, 2010.

[103] N. Wu and Y. Xie, "A survey of machine learning for computer architecture and systems," *ACM Computing Surveys (CSUR)*, vol. 55, no. 3, pp. 1–39, 2022.

[104] K. Rzadca, P. Findeisen, J. Swiderski, P. Zych, P. Broniek, J. Kusmierek, P. Nowak, B. Strack, P. Witusowski, S. Hand, *et al.*, "Autopilot: workload autoscaling at google," in *Proceedings of the Fifteenth European Conference on Computer Systems*, pp. 1–16, 2020.

[105] M. Tirmazi, A. Barker, N. Deng, M. E. Haque, Z. G. Qin, S. Hand, M. Harchol-Balter, and J. Wilkes, "Borg: the next generation," in *Proceedings of the fifteenth European conference on computer systems*, pp. 1–14, 2020.

[106] P. Ambati, I. Goiri, F. Frujeri, A. Gun, K. Wang, B. Dolan, B. Corell, S. Pasupuleti, T. Moscibroda, S. Elnikety, *et al.*, "Providing slos for resource-harvesting vms in

cloud platforms," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pp. 735–751, 2020.

[107] A. G. Kumbhare, R. Azimi, I. Manousakis, A. Bonde, F. Frujeri, N. Mahalingam, P. A. Misra, S. A. Javadi, B. Schroeder, M. Fontoura, *et al.*, "{Prediction-Based$}$ power oversubscription in cloud platforms," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pp. 473–487, 2021.

[108] X. Yu, P. Joshi, J. Xu, G. Jin, H. Zhang, and G. Jiang, "Cloudseer: Workflow monitoring of cloud infrastructures via interleaved logs," in *ACM SIGPLAN Notices*, vol. 51, pp. 489–502, ACM, ACM New York, NY, USA, 2016.

[109] Y. Gan, Y. Zhang, K. Hu, D. Cheng, Y. He, M. Pancholi, and C. Delimitrou, "Seer: Leveraging big data to navigate the complexity of performance debugging in cloud microservices," in *Proceedings of the twenty-fourth international conference on architectural support for programming languages and operating systems*, pp. 19–33, 2019.

[110] L. M. Dang, M. Piran, D. Han, K. Min, H. Moon, *et al.*, "A survey on internet of things and cloud computing for healthcare," *Electronics*, vol. 8, no. 7, p. 768, 2019.

[111] R. Mangharam, *The car and the cloud: Automotive architectures for 2020*, 2012.

[112] M. S. Mekala and P. Viswanathan, "A survey: Smart agriculture iot with cloud computing," in *2017 international conference on microelectronic devices, circuits and systems (ICMDCS)*, pp. 1–7, IEEE, 2017.

[113] J. Meza, T. Xu, K. Veeraraghavan, and O. Mutlu, "A large scale study of data center network reliability," in *Proceedings of the Internet Measurement Conference 2018*, pp. 393–407, 2018.

[114] D. Chiou, "The microsoft catapult project," in *2017 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 124–124, IEEE Computer Society, 2017.

[115] J. Fowers, K. Ovtcharov, M. K. Papamichael, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, L. Adams, M. Ghandi, *et al.*, "Inside project brainwave's cloud-scale, real-time ai processor," *IEEE Micro*, vol. 39, no. 3, pp. 20–28, 2019.

[116] A. Vasan, A. Sivasubramaniam, V. Shimpi, T. Sivabalan, and R. Subbiah, "Worth their watts?-an empirical study of datacenter servers," in *HPCA-16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, pp. 1–10, IEEE, 2010.

[117] J. Mars and L. Tang, "Whare-map: Heterogeneity in" homogeneous" warehouse-scale computers," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, pp. 619–630, 2013.

[118] A. Barbalace, R. Lyerly, C. Jelesnianski, A. Carno, H.-R. Chuang, V. Legout, and B. Ravindran, "Breaking the boundaries in heterogeneous-isa datacenters," *ACM SIGARCH Computer Architecture News*, vol. 45, no. 1, pp. 645–659, 2017.

[119] J. Mars, L. Tang, and R. Hundt, "Heterogeneity in "homogeneous" warehouse-scale

computers: A performance opportunity," *IEEE Computer Architecture Letters*, vol. 10, no. 2, pp. 29–32, 2011.

[120] C. Zhang, M. Yu, W. Wang, and F. Yan, "{MArk$}$: Exploiting cloud services for ${$cost-effective$}$,${$slo-aware$}$ machine learning inference serving," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pp. 1049–1062, 2019.

[121] D. Narayanan, K. Santhanam, F. Kazhamiaka, A. Phanishayee, and M. Zaharia, "Analysis and exploitation of dynamic pricing in the public cloud for ml training," in *VLDB DISPA Workshop 2020*, 2020.

[122] W. Zhang, X. Zhao, S. Jiang, and H. Jiang, "Chameleondb: a key-value store for optane persistent memory," in *Proceedings of the Sixteenth European Conference on Computer Systems*, pp. 194–209, 2021.

[123] L. Lersch, X. Hao, I. Oukid, T. Wang, and T. Willhalm, "Evaluating persistent memory range indexes," *Proceedings of the VLDB Endowment*, vol. 13, no. 4, pp. 574–587, 2019.

[124] A. Shanbhag, N. Tatbul, D. Cohen, and S. Madden, "Large-scale in-memory analytics on intel® optane™ dc persistent memory," in *Proceedings of the 16th International Workshop on Data Management on New Hardware*, pp. 1–8, 2020.

[125] R. Nathuji, C. Isci, and E. Gorbatov, "Exploiting platform heterogeneity for power efficient data centers," in *Fourth International Conference on Autonomic Computing (ICAC'07)*, pp. 5–5, IEEE, 2007.

[126] 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)*, pp. 1–10, IEEE, 2016.

[127] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, *et al.*, "An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 3–18, 2019.

[128] L. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, S. Zhang, *et al.*, "Bigdatabench: A big data benchmark suite from internet services," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pp. 488–499, IEEE, 2014.

[129] J. Scheuner and P. Leitner, "A cloud benchmark suite combining micro and applications benchmarks," in *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, pp. 161–166, 2018.

[130] Y. Xiang, X. Wang, Z. Huang, Z. Wang, Y. Luo, and Z. Wang, "Dcaps: dynamic cache allocation with partial sharing," in *Proceedings of the Thirteenth EuroSys Conference*, p. 13, ACM, 2018.

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

IEEE, 2020.

[132] S. Di, D. Kondo, and F. Cappello, "Characterizing cloud applications on a google data center," in *2013 42nd International Conference on Parallel Processing*, pp. 468–473, IEEE, 2013.

[133] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets.," *HotCloud*, vol. 10, no. 10-10, p. 95, 2010.

[134] S. Mohan, S. Mullapudi, S. Sammeta, P. Vijayvergia, and D. C. Anastasiu, "Stock price prediction using news sentiment analysis," in *2019 IEEE Fifth International Conference on Big Data Computing Service and Applications (BigDataService)*, pp. 205–208, IEEE, 2019.

[135] B. Langmead and A. Nellore, "Cloud computing for genomic data analysis and collaboration," *Nature Reviews Genetics*, vol. 19, no. 4, pp. 208–219, 2018.

[136] K. Koliogeorgi, S. Xydis, G. Gaydadjiev, and D. J. Soudris, "Dataflow acceleration for short read alignment on ngs data," *IEEE Transactions on Computers*, 2022.

[137] K. Koliogeorgi, N. Voss, S. Fytraki, S. Xydis, G. Gaydadjiev, and D. Soudris, "Dataflow acceleration of smith-waterman with traceback for high throughput next generation sequencing," in *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 74–80, IEEE, 2019.

[138] G. Yeung, D. Borowiec, A. Friday, R. Harper, and P. Garraghan, "Towards {GPU$}$ utilization prediction for cloud deep learning," in *12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 20)*, 2020.

[139] C. Pinto, Y. Gkoufas, A. Reale, S. Seelam, and S. Eliuk, "Hoard: A distributed data caching system to accelerate deep learning training on the cloud," *arXiv preprint arXiv:1812.00669*, 2018.

[140] M. U. Yaseen, A. Anjum, O. Rana, and N. Antonopoulos, "Deep learning hyper-parameter optimization for video analytics in clouds," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, vol. 49, no. 1, pp. 253–264, 2018.

[141] R. Poddar, G. Ananthanarayanan, S. Setty, S. Volos, and R. A. Popa, "Visor:{Privacy-Preserving$}$ video analytics as a cloud service," in *29th USENIX Security Symposium (USENIX Security 20)*, pp. 1039–1056, 2020.

[142] L. A. Barroso, J. Dean, and U. Holzle, "Web search for a planet: The google cluster architecture," *IEEE micro*, vol. 23, no. 2, pp. 22–28, 2003.

[143] D. Ardelean, A. Diwan, and C. Erdman, "Performance analysis of cloud applications," in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pp. 405–417, 2018.

[144] J. Dean and L. A. Barroso, "The tail at scale," *Communications of the ACM*, vol. 56, no. 2, pp. 74–80, 2013.

[145] M. Nguyen, Z. Li, F. Duan, H. Che, and H. Jiang, "The tail at scale: how to predict it?," in *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*, 2016.

[146] G. Da Cunha Rodrigues, R. N. Calheiros, V. T. Guimaraes, G. L. d. Santos, M. B.

De Carvalho, L. Z. Granville, L. M. R. Tarouco, and R. Buyya, “Monitoring of cloud computing environments: concepts, solutions, trends, and future directions,” in *Proceedings of the 31st annual ACM symposium on applied computing*, pp. 378–383, 2016.

[147] G. Aceto, A. Botta, W. De Donato, and A. Pescapè, “Cloud monitoring: A survey,” *Computer Networks*, vol. 57, no. 9, pp. 2093–2115, 2013.

[148] L. Wu, J. Bogatinovski, S. Nedelkoski, J. Tordsson, and O. Kao, “Performance diagnosis in cloud microservices using deep learning,” in *International Conference on Service-Oriented Computing*, pp. 85–96, Springer, 2020.

[149] R. R. Schmidt, E. E. Cruz, and M. Iyengar, “Challenges of data center thermal management,” *IBM Journal of Research and Development*, vol. 49, no. 4.5, pp. 709–723, 2005.

[150] P. Singh, L. Klein, D. Agonafer, J. M. Shah, and K. D. Pujara, “Effect of relative humidity, temperature and gaseous and particulate contaminations on information technology equipment reliability,” in *International Electronic Packaging Technical Conference and Exhibition*, vol. 56888, p. V001T09A015, American Society of Mechanical Engineers, 2015.

[151] N. El-Sayed, I. A. Stefanovici, G. Amvrosiadis, A. A. Hwang, and B. Schroeder, “Temperature management in data centers: Why some (might) like it hot,” in *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems*, pp. 163–174, 2012.

[152] R. Brondolin and M. D. Santambrogio, “A black-box monitoring approach to measure microservices runtime performance,” *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 17, no. 4, pp. 1–26, 2020.

[153] Y. Gan, M. Liang, S. Dev, D. Lo, and C. Delimitrou, “Sage: practical and scalable ml-driven performance debugging in microservices,” in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 135–151, 2021.

[154] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, “Borg, omega, and kubernetes: Lessons learned from three container-management systems over a decade,” *Queue*, vol. 14, no. 1, pp. 70–93, 2016.

[155] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica, “Mesos: A platform for fine-grained resource sharing in the data center.,” in *NSDI*, vol. 11, pp. 22–22, 2011.

[156] M. Kogias and E. Bugnion, “Hovercraft: achieving scalability and fault-tolerance for microsecond-scale datacenter services,” in *Proceedings of the Fifteenth European Conference on Computer Systems*, pp. 1–17, 2020.

[157] M. A. Mukwevho and T. Celik, “Toward a smart cloud: A review of fault-tolerance methods in cloud systems,” *IEEE Transactions on Services Computing*, vol. 14, no. 2, pp. 589–605, 2018.

[158] M. K. Aguilera, N. Ben-David, R. Guerraoui, V. J. Marathe, A. Xygkis, and I. Zablotchi, “Microsecond consensus for microsecond applications,” in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pp. 599–616, 2020.

[159] C. Delimitrou and C. Kozyrakis, “Bolt: I know what you did last summer... in the cloud,” *ACM SIGARCH Computer Architecture News*, vol. 45, no. 1, pp. 599–613, 2017.

[160] V. Gandhi and J. Mickens, “Rethinking isolation mechanisms for datacenter multitenancy,” in *12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 20)*, 2020.

[161] C. A. Ardagna, R. Asal, E. Damiani, and Q. H. Vu, “From security to assurance in the cloud: A survey,” *ACM Computing Surveys (CSUR)*, vol. 48, no. 1, pp. 1–50, 2015.

[162] T. L. Duc, R. G. Leiva, P. Casari, and P.-O. Östberg, “Machine learning methods for reliable resource provisioning in edge-cloud computing: A survey,” *ACM Computing Surveys (CSUR)*, vol. 52, no. 5, pp. 1–39, 2019.

[163] A. Jain, M. A. Laurenzano, L. Tang, and J. Mars, “Continuous shape shifting: Enabling loop co-optimization via near-free dynamic code rewriting,” in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1–12, IEEE, 2016.

[164] Y. Sato, T. Yuki, and T. Endo, “An autotuning framework for scalable execution of tiled code via iterative polyhedral compilation,” *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 15, no. 4, pp. 1–23, 2019.

[165] J. Zhao, H. Cui, Y. Zhang, J. Xue, and X. Feng, “Revisiting loop tiling for datacenters: live and let live,” in *Proceedings of the 2018 International Conference on Supercomputing*, pp. 328–340, 2018.

[166] N. Kulkarni, F. Qi, and C. Delimitrou, “Leveraging approximation to improve datacenter resource efficiency,” *IEEE Computer Architecture Letters*, vol. 17, no. 2, pp. 171–174, 2018.

[167] S. Li, S. Park, and S. Mahlke, “Sculptor: Flexible approximation with selective dynamic loop perforation,” in *Proceedings of the 2018 International Conference on Supercomputing*, pp. 341–351, 2018.

[168] A. Tiwari, C. Chen, J. Chame, M. Hall, and J. K. Hollingsworth, “A scalable auto-tuning framework for compiler optimization,” in *2009 IEEE International Symposium on Parallel & Distributed Processing*, pp. 1–12, IEEE, 2009.

[169] A. Kaplunovich and Y. Yesha, “Automatic tuning of hyperparameters for neural networks in serverless cloud,” in *2020 IEEE International Conference on Big Data (Big Data)*, pp. 2751–2756, IEEE, 2020.

[170] Y. Guo, H. Shan, S. Huang, K. Hwang, J. Fan, and Z. Yu, “Gml: Efficiently autotuning flink’s configurations via guided machine learning,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 12, pp. 2921–2935, 2021.

[171] J. Hauswald, Y. Kang, M. A. Laurenzano, Q. Chen, C. Li, T. Mudge, R. G. Dreslinski, J. Mars, and L. Tang, “Djinn and tonic: Dnn as a service and its implications for future warehouse scale computers,” in *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pp. 27–40, IEEE, 2015.

[172] A. Kokkinis, A. Ferikoglou, D. Danopoulos, D. Masouros, and K. Siozios, “Leveraging hw approximation for exploiting performance-energy trade-offs within the edge-cloud computing continuum,” in *International Conference on High Performance Computing*, pp. 406–415, Springer, 2021.

[173] N. Kulkarni, F. Qi, and C. Delimitrou, “Pliant: Leveraging approximation to improve datacenter resource efficiency,” in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 159–171, IEEE, 2019.

[174] G. Armeniakos, G. Zervakis, D. Soudris, and J. Henkel, “Hardware approximate techniques for deep neural network accelerators: A survey,” *ACM Computing Surveys (CSUR)*, 2022.

[175] F. Romero, Q. Li, N. J. Yadwadkar, and C. Kozyrakis, “{INFaaS$}$: Automated model-less inference serving,” in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pp. 397–411, 2021.

[176] L. Wang, L. Yang, Y. Yu, W. Wang, B. Li, X. Sun, J. He, and L. Zhang, “Morphling: Fast, near-optimal auto-configuration for cloud-native model serving,” in *Proceedings of the ACM Symposium on Cloud Computing*, pp. 639–653, 2021.

[177] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, *et al.*, “Pytorch: An imperative style, high-performance deep learning library,” *Advances in neural information processing systems*, vol. 32, pp. 8026–8037, 2019.

[178] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, *et al.*, “{TensorFlow$}$: A system for ${$large-scale$}$ machine learning,” in *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, pp. 265–283, 2016.

[179] N. Hasabnis, “Auto-tuning tensorflow threading model for cpu backend,” in *2018 IEEE/ACM Machine Learning in HPC Environments (MLHPC)*, pp. 14–25, IEEE, 2018.

[180] D. Mebratu, N. Hasabnis, P. Mercati, G. Sharma, and S. Najnin, “Automatic tuning of tensorflow’s cpu backend using gradient-free optimization algorithms,” in *International Conference on High Performance Computing*, pp. 249–266, Springer, 2021.

[181] Z. Bei, Z. Yu, N. Luo, C. Jiang, C. Xu, and S. Feng, “Configuring in-memory cluster computing using random forest,” *Future Generation Computer Systems*, vol. 79, pp. 1–15, 2018.

[182] M. Li, Z. Liu, X. Shi, and H. Jin, “Atcs: Auto-tuning configurations of big data frameworks based on generative adversarial nets,” *IEEE Access*, vol. 8, pp. 50485–50496, 2020.

[183] S. Akoush, R. Sohan, A. Rice, A. W. Moore, and A. Hopper, "Predicting the performance of virtual machine migration," in *2010 IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pp. 37–46, 2010.

[184] V. Shrivastava, P. Zerfos, K.-W. Lee, H. Jamjoom, Y.-H. Liu, and S. Banerjee, "Application-aware virtual machine migration in data centers," in *2011 Proceedings IEEE INFOCOM*, pp. 66–70, IEEE, 2011.

[185] L. M. Vaquero, L. Rodero-Merino, and R. Buyya, "Dynamically scaling applications in the cloud," *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 1, pp. 45–52, 2011.

[186] T. Lorido-Botran, J. Miguel-Alonso, and J. A. Lozano, "A review of auto-scaling techniques for elastic applications in cloud environments," *Journal of grid computing*, vol. 12, no. 4, pp. 559–592, 2014.

[187] S. Islam, K. Lee, A. Fekete, and A. Liu, "How a consumer can measure elasticity for cloud platforms," in *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*, pp. 85–96, 2012.

[188] C. Lu, K. Ye, G. Xu, C.-Z. Xu, and T. Bai, "Imbalance in the cloud: An analysis on alibaba cluster trace," in *2017 IEEE International Conference on Big Data (Big Data)*, pp. 2884–2892, IEEE, 2017.

[189] H. M. Makrani, H. Sayadi, N. Nazari, S. M. P. Dinakarrao, A. Sasan, T. Mohsenin, S. Rafatirad, and H. Homayoun, "Adaptive performance modeling of data-intensive workloads for resource provisioning in virtualized environment," *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS)*, vol. 5, no. 4, pp. 1–24, 2021.

[190] J. Chase and D. Niyato, "Joint optimization of resource provisioning in cloud computing," *IEEE Transactions on Services Computing*, vol. 10, no. 3, pp. 396–409, 2015.

[191] S. Mireslami, L. Rakai, M. Wang, and B. H. Far, "Dynamic cloud resource allocation considering demand uncertainty," *IEEE Transactions on Cloud Computing*, vol. 9, no. 3, pp. 981–994, 2019.

[192] F. Romero and C. Delimitrou, "Mage: Online and interference-aware scheduling for multi-scale heterogeneous systems," in *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, pp. 1–13, 2018.

[193] P. Garefalakis, K. Karanasos, P. Pietzuch, A. Suresh, and S. Rao, "Medea: scheduling of long running applications in shared production clusters," in *Proceedings of the thirteenth EuroSys conference*, pp. 1–13, 2018.

[194] L. Pons, J. Sahuquillo, V. Selfa, S. Petit, and J. Pons, "Phase-aware cache partitioning to target both turnaround time and system performance," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 11, pp. 2556–2568, 2020.

[195] A. Bhattacharyya, C. Amza, and E. de Lara, "Phase aware performance modeling for cloud applications," in *2020 IEEE 13th International Conference on Cloud*

Computing (CLOUD), pp. 507–511, IEEE, 2020.

[196] R. Nishtala, V. Petrucci, P. Carpenter, and M. Sjalander, "Twig: Multi-agent task management for colocated latency-critical cloud services," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 167–179, IEEE, 2020.

[197] A. Tzenetopoulos, D. Masouros, S. Xydis, and D. Soudris, "Interference-aware orchestration in kubernetes," in *International Conference on High Performance Computing*, pp. 321–330, Springer, 2020.

[198] B. Pawlowski, "Design of a composable infrastructure platform," in *Linux Storage and Filesystems Conference*, (Boston, MA), USENIX Association, Feb. 2019.

[199] I.-H. Chung, B. Abali, and P. Crumley, "Towards a composable computer system," in *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region*, pp. 137–147, 2018.

[200] I. Burstein, "Nvidia data center processing unit (dpu) architecture," in *2021 IEEE Hot Chips 33 Symposium (HCS)*, pp. 1–20, IEEE, 2021.

[201] K. Ebrahimi, G. F. Jones, and A. S. Fleischer, "A review of data center cooling technology, operating conditions and the corresponding low-grade waste heat recovery opportunities," *Renewable and Sustainable Energy Reviews*, vol. 31, pp. 622–638, 2014.

[202] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren, "Inside the social network's (datacenter) network," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pp. 123–137, 2015.

[203] Q. Wu, Q. Deng, L. Ganesh, C.-H. Hsu, Y. Jin, S. Kumar, B. Li, J. Meza, and Y. J. Song, "Dynamo: Facebook's data center-wide power management system," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 469–480, 2016.

[204] C. Wang, K. Schwan, V. Talwar, G. Eisenhauer, L. Hu, and M. Wolf, "A flexible architecture integrating monitoring and analytics for managing large-scale data centers," in *Proceedings of the 8th ACM international conference on Autonomic computing*, pp. 141–150, 2011.

[205] M. Du and F. Li, "Atom: efficient tracking, monitoring, and orchestration of cloud resources," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 8, pp. 2172–2189, 2017.

[206] J. Varia, S. Mathew, *et al.*, "Overview of amazon web services," *Amazon Web Services*, vol. 105, pp. 1–22, 2014.

[207] *IBM Cloud.*
https://www.ibm.com/cloud.

[208] *https://cloud.google.com/monitoring*, (Last accessed 16/3/2022).

[209] *https://www.ibm.com/cloud/cloud-monitoring*, (Last accessed 16/3/2022).

[210] *https://docs.microsoft.com/en-us/azure/azure-monitor/overview*, (Last accessed 16/3/2022).

[211] *Prometheus - Monitoring system & time series database.*

https://prometheus.io/.

[212] *Dynatrace official website.*
https://www.dynatrace.com/.

[213] *Instana - APM for Dynamic Applications.*
https://www.instana.com/.

[214] *ITRS Group: for the always-on enterprise.*
https://www.itrsgroup.com/.

[215] *Splunk official website.*
https://www.splunk.com/.

[216] D. Terpstra, H. Jagode, H. You, and J. Dongarra, "Collecting performance data with papi-c," in *Tools for High Performance Computing 2009*, pp. 157–173, Springer, 2010.

[217] T. Willhalm, R. Dementiev, and P. Fay, "Intel performance counter monitor-a better way to measure cpu utilization," *Dosegljivo: https://software. intel. com/en-us/articles/intelperformance-counter-monitor-a-better-way-to-measure-cpu-utilization.[Dostopano: September 2014]*, 2012.

[218] E. Bauman, G. Ayoade, and Z. Lin, "A survey on hypervisor-based monitoring: approaches, applications, and evolutions," *ACM Computing Surveys (CSUR)*, vol. 48, no. 1, p. 10, 2015.

[219] I. Cohen, J. S. Chase, M. Goldszmidt, T. Kelly, and J. Symons, "Correlating instrumentation data to system states: A building block for automated diagnosis and control.," in *OSDI*, vol. 4, pp. 16–16, 2004.

[220] A. Fox, E. Kiciman, and D. Patterson, "Combining statistical monitoring and predictable recovery for self-management," in *Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems*, pp. 49–53, 2004.

[221] Y. Tan, H. Nguyen, Z. Shen, X. Gu, C. Venkatramani, and D. Rajan, "Prepare: Predictive performance anomaly prevention for virtualized cloud systems," in *2012 IEEE 32nd International Conference on Distributed Computing Systems*, pp. 285–294, IEEE, 2012.

[222] P. Bodík, R. Griffith, C. Sutton, A. Fox, M. I. Jordan, and D. A. Patterson, "Statistical machine learning makes automatic control practical for internet datacenters.," *HotCloud*, vol. 9, pp. 12–12, 2009.

[223] M. A. Munawar and P. A. Ward, "Leveraging many simple statistical models to adaptively monitor software systems," in *International Symposium on Parallel and Distributed Processing and Applications*, pp. 457–470, Springer, 2007.

[224] M. Jiang, M. A. Munawar, T. Reidemeister, and P. A. Ward, "System monitoring with metric-correlation models: problems and solutions," in *Proceedings of the 6th international conference on Autonomic computing*, pp. 13–22, 2009.

[225] M. A. Munawar and P. A. Ward, "A comparative study of pairwise regression techniques for problem determination," in *Proceedings of the 2007 conference of the center for advanced studies on Collaborative research*, pp. 152–166, 2007.

[226] K. Shen, C. Stewart, C. Li, and X. Li, “Reference-driven performance anomaly identification,” *ACM SIGMETRICS Performance Evaluation Review*, vol. 37, no. 1, pp. 85–96, 2009.

[227] P. Bodík, M. Goldszmidt, and A. Fox, “Hilighter: Automatically building robust signatures of performance behavior for small-and large-scale systems.,” in *SysML*, 2008.

[228] L. Cherkasova, K. Ozonat, N. Mi, J. Symons, and E. Smirni, “Anomaly? application change? or workload change? towards automated detection of application performance anomaly and change,” in *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*, pp. 452–461, IEEE, 2008.

[229] G. Zhao, S. Hassan, Y. Zou, D. Truong, and T. Corbin, “Predicting performance anomalies in software systems at run-time,” *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 30, no. 3, pp. 1–33, 2021.

[230] X. Gu and H. Wang, “Online anomaly prediction for robust cluster systems,” in *2009 IEEE 25th International Conference on Data Engineering*, pp. 1000–1011, IEEE, 2009.

[231] A. W. Williams, S. M. Pertet, and P. Narasimhan, “Tiresias: Black-box failure prediction in distributed systems,” in *2007 IEEE international parallel and distributed processing symposium*, pp. 1–8, IEEE, 2007.

[232] S. Huang, C. Fung, K. Wang, P. Pei, Z. Luan, and D. Qian, “Using recurrent neural networks toward black-box system anomaly prediction,” in *2016 IEEE/ACM 24th International Symposium on Quality of Service (IWQoS)*, pp. 1–10, IEEE, 2016.

[233] Y. Tan, X. Gu, and H. Wang, “Adaptive system anomaly prediction for large-scale hosting infrastructures,” in *Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, pp. 173–182, 2010.

[234] Y. Gan, M. Pancholi, S. Hu, D. Cheng, Y. He, and C. Delimitrou, “Seer: Leveraging big data to navigate the increasing complexity of cloud debugging,” in *10th {USENIX$}$ Workshop on Hot Topics in Cloud Computing (HotCloud 18)*, 2018.

[235] S. Blagodurov, S. Zhuravlev, A. Fedorova, and A. Kamali, “A case for numa-aware contention management on multicore systems,” in *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, pp. 557–558, ACM, 2010.

[236] Y. Sfakianakis, C. Kozanitis, C. Kozyrakis, and A. Bilas, “Quman: Profile-based improvement of cluster utilization,” *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 15, no. 3, p. 27, 2018.

[237] B. Balaji, C. Kakovitch, and B. Narayanaswamy, “Fireplace: Placing firecraker virtual machines with hindsight imitation,” *Proceedings of Machine Learning and Systems*, vol. 3, pp. 652–663, 2021.

[238] W. Attaoui and E. Sabir, “Multi-criteria virtual machine placement in cloud computing environments: a literature review,” *arXiv preprint arXiv:1802.05113*, 2018.

[239] J. Ahn, C. Kim, J. Han, Y.-r. Choi, and J. Huh, "Dynamic virtual machine scheduling in clouds for architectural shared resources," in *4th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 12)*, 2012.

[240] M. Nelson, B.-H. Lim, G. Hutchins, *et al.*, "Fast transparent migration for virtual machines.," in *USENIX Annual technical conference, general track*, pp. 391–394, 2005.

[241] F. Zhang, G. Liu, X. Fu, and R. Yahyapour, "A survey on virtual machine migration: Challenges, techniques, and open issues," *IEEE Communications Surveys & Tutorials*, vol. 20, no. 2, pp. 1206–1243, 2018.

[242] R. Nathuji, A. Kansal, and A. Ghaffarkhah, "Q-clouds: managing performance interference effects for qos-aware clouds," in *Proceedings of the 5th European conference on Computer systems*, pp. 237–250, ACM, 2010.

[243] D. Novaković, N. Vasić, S. Novaković, D. Kostić, and R. Bianchini, "{DeepDive}: Transparently identifying and managing performance interference in virtualized environments," in *2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pp. 219–230, 2013.

[244] N. Mishra, J. D. Lafferty, and H. Hoffmann, "Esp: A machine learning approach to predicting application interference," in *2017 IEEE International Conference on Autonomic Computing (ICAC)*, pp. 125–134, IEEE, 2017.

[245] S. A. Jyothi, C. Curino, I. Menache, S. M. Narayanamurthy, A. Tumanov, J. Yaniv, R. Mavlyutov, I. Goiri, S. Krishnan, J. Kulkarni, *et al.*, "Morpheus: Towards automated {SLOs} for enterprise clusters," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pp. 117–134, 2016.

[246] I. Gog, M. Schwarzkopf, A. Gleave, R. N. Watson, and S. Hand, "Firmament: Fast, centralized cluster scheduling at scale," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pp. 99–115, 2016.

[247] Y. Li, C. Shan, R. Chen, X. Tang, W. Cai, S. Tang, X. Liu, G. Wang, X. Gong, and Y. Zhang, "Gaugur: Quantifying performance interference of colocated games for improving resource utilization in cloud gaming," in *Proceedings of the 28th international symposium on high-performance parallel and distributed computing*, pp. 231–242, 2019.

[248] C. Curino, S. Krishnan, K. Karanasos, S. Rao, G. M. Fumarola, B. Huang, K. Chaliparambil, A. Suresh, Y. Chen, S. Heddaya, *et al.*, "Hydra: a federated resource manager for data-center scale analytics," in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pp. 177–192, 2019.

[249] Y. Li, C. Shan, R. Chen, X. Tang, W. Cai, S. Tang, X. Liu, G. Wang, X. Gong, and Y. Zhang, "Gaugur: Quantifying performance interference of colocated games for improving resource utilization in cloud gaming," in *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '19, (New York, NY, USA), pp. 231–242, ACM, 2019.

[250] Y. Xu, Z. Musgrave, B. Noble, and M. Bailey, "Bobtail: Avoiding long tails in the

cloud," in *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, (Lombard, IL), pp. 329–341, USENIX Association, Apr. 2013.

[251] R. Xu, S. Mitra, J. Rahman, P. Bai, B. Zhou, G. Bronevetsky, and S. Bagchi, "Pythia: Improving datacenter utilization via precise contention prediction for multiple co-located workloads," in *Proceedings of the 19th International Middleware Conference*, pp. 146–160, ACM, 2018.

[252] J. Leverich and C. Kozyrakis, "Reconciling high server utilization and sub-millisecond quality-of-service," in *Proceedings of the Ninth European Conference on Computer Systems*, pp. 1–14, 2014.

[253] Y. Sfakianakis, M. Marazakis, and A. Bilas, "Skynet: Performance-driven resource management for dynamic workloads," in *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*, pp. 527–539, IEEE, 2021.

[254] H. Kasture and D. Sanchez, "Ubik: Efficient cache sharing with strict qos for latency-critical workloads," *ACM SIGPLAN Notices*, vol. 49, no. 4, pp. 729–742, 2014.

[255] N. Morris, C. Stewart, L. Chen, R. Birke, and J. Kelley, "Model-driven computational sprinting," in *Proceedings of the Thirteenth EuroSys Conference*, p. 38, ACM, 2018.

[256] H. Zhu and M. Erez, "Dirigent: Enforcing qos for latency-critical tasks on shared multicore systems," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 2, pp. 33–47, 2016.

[257] Y. D. Barve, S. Shekhar, A. D. Chhokra, S. Khare, A. Bhattacharjee, Z. Kang, H. Sun, and A. Gokhale, "Fecbench: A holistic interference-aware approach for application performance modeling," *arXiv preprint arXiv:1904.05833*, 2019.

[258] A. D. Breslow, A. Tiwari, M. Schulz, L. Carrington, L. Tang, and J. Mars, "Enabling fair pricing on high performance computer systems with node sharing," *Scientific Programming*, vol. 22, no. 2, pp. 59–74, 2014.

[259] Y. Huang, X. Yan, G. Jiang, T. Jin, J. Cheng, A. Xu, Z. Liu, and S. Tu, "Tangram: bridging immutable and mutable abstractions for distributed data analytics," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pp. 191–206, 2019.

[260] B. Caldwell, S. Goodarzy, S. Ha, R. Han, E. Keller, E. Rozner, and Y. Im, "Fluidmem: Full, flexible, and fast memory disaggregation for the cloud," in *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*, pp. 665–677, IEEE, 2020.

[261] W. Cao and L. Liu, "Hierarchical orchestration of disaggregated memory," *IEEE Transactions on Computers*, vol. 69, no. 6, pp. 844–855, 2020.

[262] H. A. Maruf, Y. Zhong, H. Wang, M. Chowdhury, A. Cidon, and C. Waldspurger, "Memtrade: A disaggregated-memory marketplace for public clouds," *arXiv preprint arXiv:2108.06893*, 2021.

[263] H. Al Maruf and M. Chowdhury, “Effectively prefetching remote memory with leap,” in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pp. 843–857, 2020.

[264] K. Lim, Y. Turner, J. R. Santos, A. AuYoung, J. Chang, P. Ranganathan, and T. F. Wenisch, “System-level implications of disaggregated memory,” in *IEEE International Symposium on High-Performance Comp Architecture*, pp. 1–12, IEEE, 2012.

[265] J. Weiner, N. Agarwal, D. Schatzberg, L. Yang, H. Wang, B. Sanouillet, B. Sharma, T. Heo, M. Jain, C. Tang, *et al.*, “Tmo: transparent memory offloading in datacenters,” in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 609–621, 2022.

[266] H. Li, D. S. Berger, S. Novakovic, L. Hsu, D. Ernst, P. Zardoshti, M. Shah, I. Agarwal, M. Hill, M. Fontoura, *et al.*, “First-generation memory disaggregation for cloud platforms,” *arXiv preprint arXiv:2203.00241*, 2022.

[267] N. Nordlund, V. Vassiliadis, M. Gazzetti, D. Syrivelis, and L. Tassiulas, “Energy-aware learning agent (eala) for disaggregated cloud scheduling,” in *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*, pp. 578–583, IEEE, 2021.

[268] F. V. Zacarias, R. Nishtala, and P. Carpenter, “Contention-aware application performance prediction for disaggregated memory systems,” in *Proceedings of the 17th ACM International Conference on Computing Frontiers*, pp. 49–59, 2020.

[269] S. Chintapalli, D. Dagit, B. Evans, R. Farivar, T. Graves, M. Holderbaugh, Z. Liu, K. Nusbaum, K. Patil, B. J. Peng, *et al.*, “Benchmarking streaming computation engines: Storm, flink and spark streaming,” in *2016 IEEE international parallel and distributed processing symposium workshops (IPDPSW)*, pp. 1789–1792, IEEE, 2016.

[270] H. Herodotou, Y. Chen, and J. Lu, “A survey on automatic parameter tuning for big data processing systems,” *ACM Computing Surveys (CSUR)*, vol. 53, no. 2, pp. 1–37, 2020.

[271] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B.-G. Chun, “Making sense of performance in data analytics frameworks,” in *12th {USENIX$}$ Symposium on Networked Systems Design and Implementation (${$NSDI$}$ 15)*, pp. 293–307, 2015.

[272] R. Tous, A. Gounaris, C. Tripiana, J. Torres, S. Girona, E. Ayguadé, J. Labarta, Y. Becerra, D. Carrera, and M. Valero, “Spark deployment and performance evaluation on the marenostrum supercomputer,” in *2015 IEEE International Conference on Big Data (Big Data)*, pp. 299–306, IEEE, 2015.

[273] A. J. Awan, M. Brorsson, V. Vlassov, and E. Ayguade, “How data volume affects spark based data analytics on a scale-up server,” in *BPOE*, pp. 81–92, Springer, 2015.

[274] B. Chambers and M. Zaharia, *Spark: The definitive guide: Big data processing*

made simple.
" O'Reilly Media, Inc.", 2018.

[275] "Explore best practices for spark performance optimization." https://developer.ibm.com/blogs/spark-performance-optimization-guidelines/.
Accessed: 12-09-2022.

[276] "Spark job tuning tips." https://cloud.google.com/dataproc/docs/support/spark-job-tuning.
Accessed: 12-09-2022.

[277] A. Gounaris and J. Torres, "A methodology for spark parameter tuning," *Big data research*, vol. 11, pp. 22–32, 2018.

[278] P. Petridis, A. Gounaris, and J. Torres, "Spark parameter tuning via trial-and-error," in *INNS Conference on Big Data*, pp. 226–237, Springer, 2016.

[279] Y. Zhu, J. Liu, M. Guo, Y. Bao, W. Ma, Z. Liu, K. Song, and Y. Yang, "Bestconfig: tapping the performance potential of systems via automatic configuration tuning," in *Proceedings of the 2017 Symposium on Cloud Computing*, pp. 338–350, 2017.

[280] A. Fekry, L. Carata, T. Pasquier, and A. Rice, "Accelerating the configuration tuning of big data analytics with similarity-aware multitask bayesian optimization," in *2020 IEEE International Conference on Big Data (Big Data)*, pp. 266–275, IEEE, 2020.

[281] A. Fekry, L. Carata, T. Pasquier, A. Rice, and A. Hopper, "Tuneful: An online significance-aware configuration tuner for big data analytics," *arXiv preprint arXiv:2001.08002*, 2020.

[282] A. Anderson, S. Dubois, A. Cuesta-Infante, and K. Veeramachaneni, "Sample, estimate, tune: Scaling bayesian auto-tuning of data science pipelines," in *2017 IEEE International Conference on Data Science and Advanced Analytics (DSAA)*, pp. 361–372, IEEE, 2017.

[283] Z. Bei, Z. Yu, H. Zhang, W. Xiong, C. Xu, L. Eeckhout, and S. Feng, "Rfhoc: A random-forest approach to auto-tuning hadoop's configuration," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 5, pp. 1470–1483, 2015.

[284] D. B. Prats, F. A. Portella, C. H. Costa, and J. L. Berral, "You only run once: Spark auto-tuning from a single run," *IEEE Transactions on Network and Service Management*, vol. 17, no. 4, pp. 2039–2051, 2020.

[285] Z. Bei, N. S. Kim, K. Hwang, and Z. Yu, "Osc: An online self-configuring big data framework for optimization of qos (tc-2020-02-0128. r1)," *IEEE Transactions on Computers*, 2021.

[286] G. Wang, J. Xu, and B. He, "A novel method for tuning configuration parameters of spark based on machine learning," in *2016 IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pp. 586–593, IEEE, 2016.

[287] Z. Zong, L. Wen, X. Hu, R. Han, C. Qian, and L. Lin, "Mespaconfig: Memory-

sparing configuration auto-tuning for co-located in-memory cluster computing jobs," *IEEE Transactions on Services Computing*, 2021.

[288] N. Nguyen, M. M. H. Khan, Y. Albayram, and K. Wang, "Understanding the influence of configuration settings: An execution model-driven framework for apache spark platform," in *2017 IEEE 10th International Conference on Cloud Computing (CLOUD)*, pp. 802–807, IEEE, 2017.

[289] A. Klimovic, H. Litz, and C. Kozyrakis, "Selecta: Heterogeneous cloud storage configuration for data analytics," in *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, pp. 759–773, 2018.

[290] A. Fekry, L. Carata, T. Pasquier, A. Rice, and A. Hopper, "To tune or not to tune? in search of optimal configurations for data analytics," in *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pp. 2494–2504, 2020.

[291] S. Shen, V. Van Beek, and A. Iosup, "Statistical characterization of business-critical workloads hosted in cloud datacenters," in *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pp. 465–474, IEEE, 2015.

[292] Z. Ren, J. Wan, W. Shi, X. Xu, and M. Zhou, "Workload analysis, implications, and optimization on a production hadoop cluster: A case study on taobao," *IEEE Transactions on Services Computing*, vol. 7, no. 2, pp. 307–321, 2013.

[293] M. Kambadur, T. Moseley, R. Hank, and M. A. Kim, "Measuring interference between live datacenter applications," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, p. 51, IEEE Computer Society Press, 2012.

[294] K. Du Bois, S. Eyerman, and L. Eeckhout, "Per-thread cycle accounting in multicore processors," *ACM Trans. Archit. Code Optim.*, vol. 9, pp. 29:1–29:22, Jan. 2013.

[295] E. Ebrahimi, C. J. Lee, O. Mutlu, and Y. N. Patt, "Fairness via source throttling: A configurable and high-performance fairness substrate for multi-core memory systems," in *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, (New York, NY, USA), pp. 335–346, ACM, 2010.

[296] L. Subramanian, V. Seshadri, A. Ghosh, S. Khan, and O. Mutlu, "The application slowdown model: Quantifying and controlling the impact of inter-application interference at shared caches and main memory," in *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, (New York, NY, USA), pp. 62–75, ACM, 2015.

[297] A. Herdrich, E. Verplanke, P. Autee, R. Illikkal, C. Gianos, R. Singhal, and R. Iyer, "Cache qos: From concept to reality in the intel® xeon® processor e5-2600 v3 product family," in *High Performance Computer Architecture (HPCA), 2016 IEEE International Symposium on*, pp. 657–668, IEEE, 2016.

[298] H. David, E. Gorbatov, U. R. Hanebutte, R. Khanna, and C. Le, "Rapl: memory power estimation and capping," in *Proceedings of the 16th ACM/IEEE interna-*

tional symposium on Low power electronics and design, pp. 189–194, ACM, 2010.

[299] R. Bertran, M. Gonzalez, X. Martorell, N. Navarro, and E. Ayguade, "Decomposable and responsive power models for multicore processors using performance counters," in *Proceedings of the 24th ACM International Conference on Supercomputing*, pp. 147–158, ACM, 2010.

[300] J. C. McCullough, Y. Agarwal, J. Chandrashekar, S. Kuppuswamy, A. C. Snoeren, and R. K. Gupta, "Evaluating the effectiveness of model-based power characterization," in *USENIX Annual Technical Conf*, vol. 20, 2011.

[301] M. Zaman, A. Ahmadi, and Y. Makris, "Workload characterization and prediction: A pathway to reliable multi-core systems," in *2015 IEEE 21st International On-Line Testing Symposium (IOLTS)*, pp. 116–121, IEEE, 2015.

[302] D. Merkel, "Docker: Lightweight linux containers for consistent development and deployment," *Linux J.*, vol. 2014, Mar. 2014.

[303] J. Hauswald, M. A. Laurenzano, Y. Zhang, C. Li, A. Rovinski, A. Khurana, R. G. Dreslinski, T. Mudge, V. Petrucci, L. Tang, *et al.*, "Sirius: An open end-to-end voice and vision personal assistant and its implications for future warehouse scale computers," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 223–238, 2015.

[304] S. Blagodurov, S. Zhuravlev, and A. Fedorova, "Contention-aware scheduling on multicore systems," *ACM Transactions on Computer Systems (TOCS)*, vol. 28, no. 4, p. 8, 2010.

[305] Y. Liu, S. C. Draper, and N. S. Kim, "Sleepscale: Runtime joint speed scaling and sleep states management for power efficient data centers," in *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pp. 313–324, June 2014.

[306] Q. Wu, Q. Deng, L. Ganesh, C. Hsu, Y. Jin, S. Kumar, B. Li, J. Meza, and Y. J. Song, "Dynamo: Facebook's data center-wide power management system," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pp. 469–480, June 2016.

[307] D. Lo, L. Cheng, R. Govindaraju, L. A. Barroso, and C. Kozyrakis, "Towards energy proportionality for large-scale latency-critical workloads," in *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, (Piscataway, NJ, USA), pp. 301–312, IEEE Press, 2014.

[308] R. Cochran, C. Hankendi, A. K. Coskun, and S. Reda, "Pack & cap: Adaptive dvfs and thread packing under power caps," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, (New York, NY, USA), pp. 175–185, ACM, 2011.

[309] B. Su, J. Gu, L. Shen, W. Huang, J. L. Greathouse, and Z. Wang, "Ppep: Online performance, power, and energy prediction framework and dvfs space exploration," in *Proceedings of the 47th Annual IEEE/ACM International Symposium*

on Microarchitecture, MICRO-47, (Washington, DC, USA), pp. 445–457, IEEE Computer Society, 2014.

[310] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM symposium on Cloud computing*, pp. 143–154, ACM, 2010.

[311] A. Cassandra, "Apache cassandra," *Website. Available online at http://planetcassandra. org/what-is-apache-cassandra*, p. 13, 2014.

[312] *NGINX official website.*
https://www.nginx.com/.

[313] *Solr official website.*
https://lucene.apache.org/solr/.

[314] *MySQL official website.*
https://www.mysql.com/.

[315] *CloudSuite 3.0 github page.*
https://github.com/parsa-epfl/cloudsuite.

[316] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[317] M. G. Moghaddam, W. Guan, and C. Ababei, "Investigation of lstm based prediction for dynamic energy management in chip multiprocessors," in *Green and Sustainable Computing Conference (IGSC), 2017 Eighth International*, pp. 1–8, IEEE, 2017.

[318] M. Hashemi, K. Swersky, J. A. Smith, G. Ayers, H. Litz, J. Chang, C. Kozyrakis, and P. Ranganathan, "Learning memory access patterns," *arXiv preprint arXiv:1803.02329*, 2018.

[319] R. Pascanu, Ç. Gülçehre, K. Cho, and Y. Bengio, "How to construct deep recurrent neural networks," *CoRR*, vol. abs/1312.6026, 2013.

[320] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

[321] *Grafana official website.*
https://grafana.com/.

[322] S. J. Pan and Q. Yang, "A survey on transfer learning," *IEEE Transactions on knowledge and data engineering*, vol. 22, no. 10, pp. 1345–1359, 2009.

[323] M. Caneill, N. De Palma, A. Ait-Bachir, B. Dine, R. Mokhtari, and Y. G. Cinar, "Online metrics prediction in monitoring systems," in *IEEE INFOCOM 2018-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pp. 226–231, IEEE, 2018.

[324] Q. Chen, Z. Wang, J. Leng, C. Li, W. Zheng, and M. Guo, "Avalon: towards qos awareness and improved utilization through multi-resource management in datacenters," in *Proceedings of the ACM International Conference on Supercomputing*, pp. 272–283, 2019.

[325] C. Delimitrou, D. Sanchez, and C. Kozyrakis, “Tarcil: Reconciling scheduling speed and quality in large shared clusters,” in *Proceedings of the Sixth ACM Symposium on Cloud Computing*, pp. 97–110, 2015.

[326] J. Fried, Z. Ruan, A. Ousterhout, and A. Belay, “Caladan: Mitigating interference at microsecond timescales,” in *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*, pp. 281–297, USENIX Association, 2020.

[327] Z. Yan, D. Lustig, D. W. Nellans, and A. Bhattacharjee, “Nimble page management for tiered memory systems,” in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019* (I. Bahar, M. Herlihy, E. Witchel, and A. R. Lebeck, eds.), pp. 331–345, ACM, 2019.

[328] A. Roozbeh, J. Soares, G. Q. Maguire, F. Wuhib, C. Padala, M. Mahloo, D. Turull, V. Yadhav, and D. Kostić, “Software-defined “hardware” infrastructures: A survey on enabling technologies and open research directions,” *IEEE Communications Surveys & Tutorials*, vol. 20, no. 3, pp. 2454–2485, 2018.

[329] A. Roozbeh, *Realizing Next-Generation Data Centers via Software-Defined “Hardware” Infrastructures and Resource Disaggregation: Exploiting your cache.* PhD thesis, KTH Royal Institute of Technology, 2021.

[330] K. Katrinis, D. Syrivelis, D. Pnevmatikatos, G. Zervas, D. Theodoropoulos, I. Koutsopoulos, K. Hasharoni, D. Raho, C. Pinto, F. Espina, S. Lopez-Buedo, Q. Chen, M. Nemirovsky, D. Roca, H. Klos, and T. Berends, “Rack-scale disaggregated cloud data centers: The dredbox project vision,” in *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, pp. 690–695, 2016.

[331] C. Reaño, F. Silla, G. Shainer, and S. Schultz, “Local and remote gpus perform similar with edr 100g infiniband,” in *Proceedings of the Industrial Track of the 16th International Middleware Conference*, pp. 1–7, 2015.

[332] A. Guleria, J. Lakshmi, and C. Padala, “Emf: Disaggregated gpus in datacenters for efficiency, modularity and flexibility,” in *2019 IEEE International Conference on Cloud Computing in Emerging Markets (CCEM)*, pp. 1–8, IEEE, 2019.

[333] L. Bindschaedler, A. Goel, and W. Zwaenepoel, “Hailstorm: Disaggregated compute and storage for distributed lsm-based databases,” in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 301–316, 2020.

[334] L. Liu, W. Cao, S. Sahin, Q. Zhang, J. Bae, and Y. Wu, “Memory disaggregation: Research problems and opportunities,” in *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pp. 1664–1673, IEEE, 2019.

[335] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. G. Shin, “Efficient memory disaggregation with infiniswap,” in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pp. 649–667, 2017.

[336] S. Legtchenko, H. Williams, K. Razavi, A. Donnelly, R. Black, A. Douglas,

N. Cheriere, D. Fryer, K. Mast, A. D. Brown, *et al.*, "Understanding {Rack-Scale$}$ disaggregated storage," in *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 17)*, 2017.

[337] M. Nanavati, J. Wires, and A. Warfield, "Decibel: Isolation and sharing in disaggregated {Rack-Scale$}$ storage," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pp. 17–33, 2017.

[338] A. Klimovic, H. Litz, and C. Kozyrakis, "Reflex: Remote flash ≈ local flash," *ACM SIGARCH Computer Architecture News*, vol. 45, no. 1, pp. 345–359, 2017.

[339] Y. Shan, Y. Huang, Y. Chen, and Y. Zhang, "{LegoOS$}$: A disseminated, distributed ${$os$}$ for hardware resource disaggregation," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pp. 69–87, 2018.

[340] I. Calciu, M. T. Imran, I. Puddu, S. Kashyap, H. A. Maruf, O. Mutlu, and A. Kolli, "Rethinking software runtimes for disaggregated memory," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 79–92, 2021.

[341] M. Bielski, I. Syrigos, K. Katrinis, D. Syrivelis, A. Reale, D. Theodoropoulos, N. Alachiotis, D. Pnevmatikatos, E. Pap, G. Zervas, *et al.*, "dredbox: Materializing a full-stack rack-scale system prototype of a next-generation disaggregated datacenter," in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1093–1098, IEEE, 2018.

[342] K. Asanović, "FireBox: A hardware building block for 2020 Warehouse-Scale computers," in *12th Usenix Conference on File and Storage Technologies*, (Santa Clara, CA), USENIX Association, Feb. 2014.

[343] P. X. Gao, A. Narayan, S. Karandikar, J. Carreira, S. Han, R. Agarwal, S. Ratnasamy, and S. Shenker, "Network requirements for resource disaggregation," in *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, pp. 249–264, 2016.

[344] Y. Zhu, W. Yu, B. Jiao, K. Mohror, A. Moody, and F. Chowdhury, "Efficient user-level storage disaggregation for deep learning," in *2019 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 1–12, IEEE, 2019.

[345] Z. Ruan, M. Schwarzkopf, M. K. Aguilera, and A. Belay, "{AIFM$}$:${$high-performance$}$,${$application-integrated$}$ far memory," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pp. 315–332, 2020.

[346] J. Kim, W. Choe, and J. Ahn, "Exploring the design space of page management for multi-tiered memory systems," in *2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021* (I. Calciu and G. Kuenning, eds.), pp. 715–728, USENIX Association, 2021.

[347] S. A. Javadi, A. Suresh, M. Wajahat, and A. Gandhi, "Scavenger: A black-box batch workload resource manager for improving utilization in cloud environments,"

in *Proceedings of the ACM Symposium on Cloud Computing*, pp. 272–285, 2019.

[348] E. Amaro, C. Branner-Augmon, Z. Luo, A. Ousterhout, M. K. Aguilera, A. Panda, S. Ratnasamy, and S. Shenker, "Can far memory improve job throughput?," in *Proceedings of the Fifteenth European Conference on Computer Systems*, pp. 1–16, 2020.

[349] V. Nitu, B. Teabe, A. Tchana, C. Isci, and D. Hagimont, "Welcome to zombieland: practical and energy-efficient memory disaggregation in a datacenter," in *Proceedings of the Thirteenth EuroSys Conference*, pp. 1–12, 2018.

[350] C. Wang, H. Ma, S. Liu, Y. Li, Z. Ruan, K. Nguyen, M. D. Bond, R. Netravali, M. Kim, and G. H. Xu, "Semeru: A {Memory-Disaggregated$}$ managed runtime," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pp. 261–280, 2020.

[351] A. Lagar-Cavilla, J. Ahn, S. Souhlal, N. Agarwal, R. Burny, S. Butt, J. Chang, A. Chaugule, N. Deng, J. Shahid, *et al.*, "Software-defined far memory in warehouse-scale computers," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 317–330, 2019.

[352] Z. Guo, Y. Shan, X. Luo, Y. Huang, and Y. Zhang, "Clio: A hardware-software co-designed disaggregated memory system," *arXiv preprint arXiv:2108.03492*, 2021.

[353] "Welcome to the home of thymesisflow." https://github.com/OpenCAPI/ThymesisFlow/.
(Last accessed: 16/02/2022).

[354] O. Consortium, "OpenCAPI Specification." Online: http://opencapi.org, 2017.
Accessed: January 2019.

[355] "memtier_benchmark: A high-throughput benchmarking tool for redis & memcached." https://github.com/RedisLabs/memtier_benchmark.
(Last accessed: 08/02/2022).

[356] B. Schroeder, A. Wierman, and M. Harchol-Balter, "Open versus closed: A cautionary tale," in *3rd Symposium on Networked Systems Design and Implementation (NSDI 2006), May 8-10, 2007, San Jose, California, USA, Proceedings* (L. L. Peterson and T. Roscoe, eds.), USENIX, 2006.

[357] Y. Zhang, D. Meisner, J. Mars, and L. Tang, "Treadmill: Attributing the source of tail latency through precise load testing and statistical inference," in *43rd ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2016, Seoul, South Korea, June 18-22, 2016*, pp. 456–468, IEEE Computer Society, 2016.

[358] M. Kogias, S. Mallon, and E. Bugnion, "Lancet: A self-correcting latency measuring tool," in *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019* (D. Malkhi and D. Tsafrir, eds.), pp. 881–896, USENIX Association, 2019.

[359] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload analysis

of a large-scale key-value store," in *ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS '12, London, United Kingdom, June 11-15, 2012* (P. G. Harrison, M. F. Arlitt, and G. Casale, eds.), pp. 53–64, ACM, 2012.

[360] Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Characterizing facebook's memcached workload," *IEEE Internet Comput.*, vol. 18, no. 2, pp. 41–49, 2014.

[361] S. K. Sadasivam, B. W. Thompto, R. Kalla, and W. J. Starke, "Ibm power9 processor architecture," *IEEE Micro*, vol. 37, no. 2, pp. 40–51, 2017.

[362] K. Hsieh, S. M. Khan, N. Vijaykumar, K. K. Chang, A. Boroumand, S. Ghose, and O. Mutlu, "Accelerating pointer chasing in 3d-stacked memory: Challenges, mechanisms, evaluation," in *34th IEEE International Conference on Computer Design, ICCD 2016, Scottsdale, AZ, USA, October 2-5, 2016*, pp. 25–32, IEEE Computer Society, 2016.

[363] S. Chen, S. GalOn, C. Delimitrou, S. Manne, and J. F. Martínez, "Workload characterization of interactive cloud services on big and small server platforms," in *2017 IEEE International Symposium on Workload Characterization, IISWC 2017, Seattle, WA, USA, October 1-3, 2017*, pp. 125–134, IEEE Computer Society, 2017.

[364] P. Hintjens, *ZeroMQ: messaging for many applications.*
" O'Reilly Media, Inc.", 2013.

[365] J. P. Barrett, "The coefficient of determination–some limitations," *The American Statistician*, vol. 28, no. 1, pp. 19–20, 1974.

[366] M. Breughe, S. Eyerman, and L. Eeckhout, "Mechanistic analytical modeling of superscalar in-order processor performance," *ACM Trans. Archit. Code Optim.*, vol. 11, no. 4, pp. 50:1–50:26, 2014.

[367] B. Zhang, Y. J. Ong, and T. Nakamura, "Simpo: Simultaneous prediction and optimization," in *IEEE International Conference on Services Computing, SCC 2022, Barcelona, Spain, July 10-16, 2022* (C. A. Ardagna, H. Bian, C. K. Chang, R. N. Chang, E. Damiani, S. Dustdar, J. Marco, M. P. Singh, E. Teniente, R. Ward, Z. Wang, F. Xhafa, and J. Zhang, eds.), pp. 120–122, IEEE, 2022.

[368] "Fine tuning and enhancing performance of apache spark jobs." Spark-AI Summit 2020.

[369] J. Lorraine, P. Vicol, and D. Duvenaud, "Optimizing millions of hyperparameters by implicit differentiation," in *The 23rd International Conference on Artificial Intelligence and Statistics, AISTATS 2020, 26-28 August 2020, Online [Palermo, Sicily, Italy]* (S. Chiappa and R. Calandra, eds.), vol. 108 of *Proceedings of Machine Learning Research*, pp. 1540–1552, PMLR, 2020.

[370] M. M. Khan and W. Yu, "Robotune: High-dimensional configuration tuning for cluster-based data analytics," in *50th International Conference on Parallel Processing*, pp. 1–10, 2021.

[371] W. Schenck, S. El Sayed, M. Foszczynski, W. Homberg, and D. Pleiter, "Evaluation and performance modeling of a burst buffer solution," *ACM SIGOPS Operating*

Systems Review, vol. 50, no. 2, pp. 12–26, 2017.

[372] A. Yasin, “A top-down method for performance analysis and counters architecture,” in *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 35–44, IEEE, 2014.

[373] S. Ioffe and C. Szegedy, “Batch normalization: Accelerating deep network training by reducing internal covariate shift,” in *International conference on machine learning*, pp. 448–456, PMLR, 2015.

[374] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, “Dropout: a simple way to prevent neural networks from overfitting,” *The journal of machine learning research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[375] D. Kingma and J. Ba, “Adam: A method for stochastic optimization,” in *International Conference on Learning Representations*, 12 2014.

[376] I. Loshchilov and F. Hutter, “SGDR: Stochastic gradient descent with warm restarts,” in *International Conference on Learning Representations*, 2017.

[377] B. Blonder, C. Lamanna, C. Violle, and B. J. Enquist, “The n-dimensional hypervolume,” *Global Ecology and Biogeography*, vol. 23, no. 5, pp. 595–609, 2014.

[378] X.-S. Yang, S. F. Chien, and T. O. Ting, *Bio-inspired computation in telecommunications.*

Morgan Kaufmann, 2015.

[379] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O’Reilly, and S. Amarasinghe, “Opentuner: An extensible framework for program autotuning,” in *Proceedings of the 23rd international conference on Parallel architectures and compilation*, pp. 303–316, 2014.

[380] J. Blank and K. Deb, “pymoo: Multi-objective optimization in python,” *IEEE Access*, vol. 8, pp. 89497–89509, 2020.

[381] J. Duchi, E. Hazan, and Y. Singer, “Adaptive subgradient methods for online learning and stochastic optimization.,” *Journal of machine learning research*, vol. 12, no. 7, 2011.

[382] A. Farahani, S. Voghoei, K. Rasheed, and H. R. Arabnia, “A brief review of domain adaptation,” *Advances in data science and information engineering*, pp. 877–894, 2021.

[383] D. Petcu, “Multi-cloud: expectations and current approaches,” in *Proceedings of the 2013 international workshop on Multi-cloud applications and federated clouds*, pp. 1–6, 2013.

[384] S. S. Gill, M. Xu, C. Ottaviani, P. Patros, R. Bahsoon, A. Shaghaghi, M. Golec, V. Stankovski, H. Wu, A. Abraham, *et al.*, “Ai for next generation computing: Emerging trends and future directions,” *Internet of Things*, vol. 19, p. 100514, 2022.

[385] J. Shalf, “The future of computing beyond moore’s law,” *Philosophical Transactions of the Royal Society A*, vol. 378, no. 2166, p. 20190061, 2020.

[386] H. Zhu, K. Kaffes, Z. Chen, Z. Liu, C. Kozyrakis, I. Stoica, and X. Jin,

"{RackSched$}$: A ${$microsecond-scale$}$ scheduler for ${$rack-scale$}$ computers," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pp. 1225–1240, 2020.

[387] M. A. Ganaie, M. Hu, *et al.*, "Ensemble deep learning: A review," *arXiv preprint arXiv:2104.02395*, 2021.

[388] W. Raghupathi and V. Raghupathi, "Big data analytics in healthcare: promise and potential," *Health information science and systems*, vol. 2, no. 1, pp. 1–10, 2014.

[389] A. Pentland, T. Reid, and T. Heibeck, "Big data and health: Revolutionizing medicine and public health," *WISH Big Data and Health Report*, vol. 5, p. 2019, 2013.

[390] E. Curry, "The big data value chain: definitions, concepts, and theoretical approaches," in *New horizons for a data-driven economy*, pp. 29–37, Springer, Cham, 2016.

[391] M. Viceconti, P. Hunter, and R. Hose, "Big data, big knowledge: big data for personalized healthcare," *IEEE journal of biomedical and health informatics*, vol. 19, no. 4, pp. 1209–1215, 2015.

[392] *Keycloak - Open Source Identity and Access Management For Modern Applications and Services.*

[393] M. Gupta, F. Patwa, J. Benson, and R. Sandhu, "Multi-layer authorization framework for a representative hadoop ecosystem deployment," in *Proceedings of the 22nd ACM on Symposium on Access Control Models and Technologies*, pp. 183–190, ACM, 2017.

[394] C. Maramis, A. Gkoufas, A. Vardi, E. Stalika, K. Stamatopoulos, A. Hatzidimitriou, N. Maglaveras, and I. Chouvarda, "Irprofiler–a software toolbox for high throughput immune receptor profiling," *BMC bioinformatics*, vol. 19, no. 1, p. 144, 2018.

[395] Y. Guo, X. Ding, Y. Shen, G. J. Lyon, and K. Wang, "Seqmule: automated pipeline for analysis of human exome/genome sequencing data," *Scientific reports*, vol. 5, p. 14283, 2015.

[396] C. Trapnell, L. Pachter, and S. L. Salzberg, "Tophat: discovering splice junctions with rna-seq," *Bioinformatics*, vol. 25, no. 9, pp. 1105–1111, 2009.

[397] *Apache Zeppelin.*

[398] E. J. Houtgast, V. Sima, K. Bertels, and Z. AlArs, "An efficient gpuaccelerated implementation of genomic short read mapping with bwamem," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 4, pp. 38–43, 2017.

[399] N. Ahmed, V.-M. Sima, E. Houtgast, K. Bertels, and Z. Al-Ars, "Heterogeneous hardware/software acceleration of the bwa-mem dna alignment algorithm," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, pp. 240–246, IEEE Press, 2015.

[400] B. Dutro, *Hardware acceleration of the samtools variant caller*, 2015.

[401] J. Arram, T. Kaplan, W. Luk, and P. Jiang, "Leveraging fpgas for accelerating

short read alignment," *IEEE/ACM Transactions on Computational Biology and Bioinformatics (TCBB)*, vol. 14, no. 3, pp. 668–677, 2017.

[402] J. Hauswald, M. A. Laurenzano, Y. Zhang, C. Li, A. Rovinski, A. Khurana, R. G. Dreslinski, T. Mudge, V. Petrucci, L. Tang, and J. Mars, "Sirius: An Open End-to-End Voice and Vision Personal Assistant and Its Implications for Future Warehouse Scale Computers," *SIGARCH Comput. Archit. News*, vol. 43, pp. 223–238, mar 2015.

[403] P. Thinakaran, J. R. Gunasekaran, B. Sharma, M. T. Kandemir, and C. R. Das, "Kube-knots: Resource harvesting through dynamic container orchestration in gpu-based datacenters," in *2019 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 1–13, IEEE, 2019.

[404] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, Omega, and Kubernetes," *Commun. ACM*, vol. 59, pp. 50–57, apr 2016.

[405] "GPU Memory Over-provisioning."

[406] Y. Ukidave, X. Li, and D. Kaeli, "Mystic: Predictive scheduling for gpu based cloud servers using machine learning," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 353–362, IEEE, 2016.

[407] Q. Chen, H. Yang, J. Mars, and L. Tang, "Baymax: Qos awareness and increased utilization for non-preemptive accelerators in warehouse scale computers," *ACM SIGPLAN Notices*, vol. 51, no. 4, pp. 681–696, 2016.

[408] D. Kang, T. J. Jun, D. Kim, J. Kim, and D. Kim, "Convgpu: Gpu management middleware in container based virtualized environment," in *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 301–309, IEEE, 2017.

[409] J. Gleeson and E. de Lara, "Heterogeneous GPU reallocation," *9th USENIX Workshop on Hot Topics in Cloud Computing, HotCloud 2017, co-located with USENIX ATC 2017*, 2017.

[410] Y. Peng, Y. Bao, Y. Chen, C. Wu, and C. Guo, "Optimus: an efficient dynamic resource scheduler for deep learning clusters," in *Proceedings of the Thirteenth EuroSys Conference*, pp. 1–14, 2018.

[411] V. Gupta, K. Schwan, N. Tolia, V. Talwar, and P. Ranganathan, "Pegasus: Coordinated scheduling for virtualized accelerator-based systems," in *2011 USENIX Annual Technical Conference (USENIX ATC 11)*, 2011.

[412] I. Tanasic, I. Gelado, J. Cabezas, A. Ramirez, N. Navarro, and M. Valero, "Enabling preemptive multiprogramming on GPUs," in *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pp. 193–204, 2014.

[413] T.-A. Yeh, H.-H. Chen, and J. Chou, "KubeShare: A Framework to Manage GPUs as First-Class and Shared Resources in Container Cloud," in *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '20, (New York, NY, USA), pp. 173–184, Association for Computing Machinery, 2020.

[414] S. Wang, O. J. Gonzalez, X. Zhou, and T. Williams, "An Efficient and Non-Intrusive

GPU Scheduling Framework for Deep Learning Training Systems," *Sc*, 2020.

[415] "Kubernetes GPU Scheduler Extension."

[416] "Alibaba GPU Sharing Scheduler Extension."

[417] V. Inc., "Containers on virtual machines or bare metal ?," *Deploying and Securely Managing Containerized Applications at Scale, White Paper*, dec 2018.

[418] "NVIDIA Data Center GPU Manager."

[419] V. J. Reddi, C. Cheng, D. Kanter, P. Mattson, G. Schmuelling, C.-J. Wu, B. Anderson, M. Breughe, M. Charlebois, W. Chou, R. Chukka, C. Coleman, S. Davis, P. Deng, G. Diamos, J. Duke, D. Fick, J. S. Gardner, I. Hubara, S. Idgunji, T. B. Jablin, J. Jiao, T. S. John, P. Kanwar, D. Lee, J. Liao, A. Lokhmotov, F. Massa, P. Meng, P. Micikevicius, C. Osborne, G. Pekhimenko, A. T. R. Rajan, D. Sequeira, A. Sirasao, F. Sun, H. Tang, M. Thomson, F. Wei, E. Wu, L. Xu, K. Yamada, B. Yu, G. Yuan, A. Zhong, P. Zhang, and Y. Zhou, "MLPerf Inference Benchmark," 2019.

[420] "CUDA Streams."

[421] M.-p. Service, *Multi-process service*, 2020.

[422] B. S. Everitt and A. Skrondal, *The Cambridge dictionary of statistics.*
Cambridge, 2010.

[423] W. Polasek, "Time series analysis and its applications: With r examples," 2013.

[424] John A. Gubner, *Probability and Random Processes for Electrical and Computer Engineers.*
Cambridge, 2554.

[425] Howard J. Seltman, "Experimental Design and Analysis," *Carnegie Mellon University Pittsburgh*, vol. 20, no. 2, 2016.

[426] R. Agarwal and V. Dhar, "Big data, data science, and analytics: The opportunity and challenge for is research," 2014.

[427] A. H. Ashouri, W. Killian, J. Cavazos, G. Palermo, and C. Silvano, "A survey on compiler autotuning using machine learning," *ACM Computing Surveys (CSUR)*, vol. 51, no. 5, pp. 1–42, 2018.

[428] D. Grewe, Z. Wang, and M. F. O'Boyle, "Portable mapping of data parallel programs to opencl for heterogeneous systems," in *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pp. 1–10, IEEE, 2013.

[429] A. Magni, C. Dubach, and M. O'Boyle, "Automatic optimization of thread-coarsening for graphics processors," in *Proceedings of the 23rd international conference on Parallel architectures and compilation*, pp. 455–466, 2014.

[430] T. Ben-Nun, A. S. Jakobovits, and T. Hoefler, "Neural code comprehension: A learnable representation of code semantics," *Advances in Neural Information Processing Systems*, vol. 31, 2018.

[431] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, "A survey of machine learning for big code and naturalness," *ACM Computing Surveys (CSUR)*, vol. 51, no. 4,

pp. 1–37, 2018.

[432] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. F. O'Boyle, J. Thomson, M. Toussaint, and C. K. Williams, "Using machine learning to focus iterative optimization," in *International Symposium on Code Generation and Optimization (CGO'06)*, pp. 11–pp, IEEE, 2006.

[433] J. Cavazos and M. F. O'boyle, "Method-specific dynamic compilation using logistic regression," *ACM SIGPLAN Notices*, vol. 41, no. 10, pp. 229–240, 2006.

[434] Y. Liu, E. Z. Zhang, and X. Shen, "A cross-input adaptive framework for gpu program optimizations," in *2009 IEEE International Symposium on Parallel & Distributed Processing*, pp. 1–10, IEEE, 2009.

[435] C. Cummins, P. Petoumenos, M. Steuwer, and H. Leather, "Autotuning opencl workgroup size for stencil patterns," *arXiv preprint arXiv:1511.02490*, 2015.

[436] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[437] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[438] A. Poznanski and L. Wolf, "Cnn-n-gram for handwriting word recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 2305–2314, 2016.

[439] M. Schuster and K. K. Paliwal, "Bidirectional recurrent neural networks," *IEEE transactions on Signal Processing*, vol. 45, no. 11, pp. 2673–2681, 1997.

[440] S. Siami-Namini, N. Tavakoli, and A. S. Namin, "The performance of lstm and bilstm in forecasting time series," in *2019 IEEE International Conference on Big Data (Big Data)*, pp. 3285–3292, IEEE, 2019.

[441] D. M. Hawkins, "The problem of overfitting," *Journal of chemical information and computer sciences*, vol. 44, no. 1, pp. 1–12, 2004.

[442] S. Grigorescu, B. Trasnea, T. Cocias, and G. Macesanu, "A survey of deep learning techniques for autonomous driving," *Journal of Field Robotics*, vol. 37, no. 3, pp. 362–386, 2020.

[443] R. Miotto, F. Wang, S. Wang, X. Jiang, and J. T. Dudley, "Deep learning for healthcare: review, opportunities and challenges," *Briefings in bioinformatics*, vol. 19, no. 6, pp. 1236–1246, 2018.

[444] Y. Liang, D. O'Keeffe, and N. Sastry, "Paige: Towards a hybrid-edge design for privacy-preserving intelligent personal assistants," in *Proceedings of the Third ACM International Workshop on Edge Systems, Analytics and Networking*, pp. 55–60, 2020.

[445] Y. Mao, C. You, J. Zhang, K. Huang, and K. B. Letaief, "A survey on mobile edge computing: The communication perspective," *IEEE communications surveys & tutorials*, vol. 19, no. 4, pp. 2322–2358, 2017.

[446] Z. Zhao, K. M. Barijough, and A. Gerstlauer, "Deepthings: Distributed adaptive deep learning inference on resource-constrained iot edge clusters," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 11, pp. 2348–2359, 2018.

[447] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295–2329, 2017.

[448] P. Mach and Z. Becvar, "Mobile edge computing: A survey on architecture and computation offloading," *IEEE communications surveys & tutorials*, vol. 19, no. 3, pp. 1628–1656, 2017.

[449] L. Zeng, X. Chen, Z. Zhou, L. Yang, and J. Zhang, "Coedge: Cooperative dnn inference with adaptive workload partitioning over heterogeneous edge devices," *IEEE/ACM Transactions on Networking*, vol. 29, no. 2, pp. 595–608, 2020.

[450] Y. Kang, J. Hauswald, C. Gao, A. Rovinski, T. Mudge, J. Mars, and L. Tang, "Neurosurgeon: Collaborative intelligence between the cloud and mobile edge," *ACM SIGARCH Computer Architecture News*, vol. 45, no. 1, pp. 615–629, 2017.

[451] H.-J. Jeong, H.-J. Lee, C. H. Shin, and S.-M. Moon, "Ionn: Incremental offloading of neural network computations from mobile devices to edge servers," in *Proceedings of the ACM Symposium on Cloud Computing*, pp. 401–411, 2018.

[452] Z. Zhou, X. Chen, E. Li, L. Zeng, K. Luo, and J. Zhang, "Edge intelligence: Paving the last mile of artificial intelligence with edge computing," *Proceedings of the IEEE*, vol. 107, no. 8, pp. 1738–1762, 2019.

[453] Y. Chen, B. Zheng, Z. Zhang, Q. Wang, C. Shen, and Q. Zhang, "Deep learning on mobile and embedded devices: State-of-the-art, challenges, and future directions," *ACM Computing Surveys (CSUR)*, vol. 53, no. 4, pp. 1–37, 2020.

[454] S. Laskaridis, S. I. Venieris, M. Almeida, I. Leontiadis, and N. D. Lane, "Spinn: synergistic progressive inference of neural networks over device and cloud," in *Proceedings of the 26th annual international conference on mobile computing and networking*, pp. 1–15, 2020.

[455] L. Zhou, H. Wen, R. Teodorescu, and D. H. Du, "Distributing deep neural networks with containerized partitions at the edge," in *2nd USENIX Workshop on Hot Topics in Edge Computing (HotEdge 19)*, 2019.

[456] R. Hadidi, J. Cao, M. S. Ryoo, and H. Kim, "Toward collaborative inferencing of deep neural networks on internet-of-things devices," *IEEE Internet of Things Journal*, vol. 7, no. 6, pp. 4950–4960, 2020.

[457] E.-I. Christoforidis, S. Xydis, and D. Soudris, "Cf-tune: Collaborative filtering autotuning for energy efficient many-core processors," *IEEE Computer Architecture Letters*, vol. 17, no. 1, pp. 25–28, 2017.

[458] "Edge computing: gaining the digital edge." https://atos.net/en/solutions/edge-computing-infrastructure.
Accessed: 10-09-2022.

[459] Z. Li, F. Liu, W. Yang, S. Peng, and J. Zhou, "A survey of convolutional neural networks: analysis, applications, and prospects," *IEEE transactions on neural networks and learning systems*, 2021.

[460] E. Arnold, O. Y. Al-Jarrah, M. Dianati, S. Fallah, D. Oxtoby, and A. Mouzakitis, "A survey on 3d object detection methods for autonomous driving applications," *IEEE Transactions on Intelligent Transportation Systems*, vol. 20, no. 10, pp. 3782–3795, 2019.