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

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Interference Aware Container Orchestration in Kubernetes Cluster

Αχιλλέας Α. Τζενετόπουλος
Α.Μ. : 03113412

Επιβλέπων : Δημήτριος Ι. Σούντρης
Καθηγητής ΕΜΠ

Αθήνα
Νοέμβριος 2019

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

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Interference Aware Container Orchestration in Kubernetes Cluster

Αχιλλέας Α. Τζενετόπουλος
Α.Μ. : 03113412

Επιβλέπων : Δημήτριος Ι. Σούντρης
Καθηγητής ΕΜΠ

Τριμελής Επιτροπή Εξέτασης

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

Ημερομηνία Εξέτασης:
18 Νοεμβρίου 2019

(Υπογραφή)

.....

Αχιλλέας Α. Τζενετόπουλος

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

Περίληψη

Σήμερα, ένας όλο και αυξανόμενος αριθμός απο εφαρμογές ανεβαίνει και εκτελείται σε περιβάλλοντα υπολογιστικού νέφους. Οι διαχειριστές των κέντρων δεδομένων και οι πάροχοι υπηρεσιών νέφους έχουν υιοθετήσει την συνύπαρξη και την απο κοινού μίσθωση πόρων ως πρώτης τάξης μέλημα όσον αφορά το σχεδιασμό των συστημάτων τους, με στόχο την αποτελεσματικότερη αντιμετώπιση και διαχείριση του αυξανόμενου όγκου υπολογιστικών απαιτήσεων. Την ίδια στιγμή, οι συνεχείς εξελίξεις στις τεχνολογίες υλικού των υπολογιστών, έχουν οδηγήσει στη χρήση ετερογενών συστημάτων, αλλά και τη σύνθεση τους σε ομάδες στα σύγχρονα κέντρα δεδομένων. Οι σύγχρονοι δρομολογητές και ενορχηστρωτές βασίζονται κυρίως σε απλές μετρικές του εκάστοτε συστήματος, όπως το ποσοστό χρήσης των κεντρικών μονάδων επεξεργασίας του συστήματος (CPUs) και της κεντρικής μνημης, για την τοποθέτηση των εισερχόμενων εφαρμογών στους διαθέσιμους πόρους. Ωστόσο, δεν λαμβάνεται υπόψιν η επίδραση των εφαρμογών που τοποθετούναι μαζί με άλλες σε διαμοιραζόμενους πόρους και ο ανταγωνισμός μεταξύ αυτών για την χρήση των πόρων αλλά ούτε και το πως η ετερογένεια των επιμέρους συστήμάτων μπορεί να επηρεάσει τη συνολική απόδοση.

Στην παρούσα εργασία, σχεδιάζουμε έναν δρομολογητή ενσωματωμένο σε ενορχηστρωτή σε περιβάλλον υπολογιστικού νέφους, ο οποίος έχει επίγνωση σχετικά με την ύπαρξη πιθανής συμφόρησης σε κάποιον απο τους διαμοιραζόμενους πόρους, καθώς και το διαφορετικό σχεδιασμό μεταξύ συστημάτων, ικανό να δρομολογεί αποτελεσματικά εφαρμογές που φθάνουν σε ένα κέντρο δεδομένων. Παρουσιάζουμε την επίδραση της άσκησης πίεσης σε διάφορους κοινόχρηστους πόρους ενός συστήματος και προτείνουμε έναν αντιπροσωπευτικό δείκτη, ικανό να αντικατοπτρίζει την κατάσταση του συστήματος, βασιζόμενοι σε παρατηρήσεις επι πειραμάτων. Ενσωματώνουμε τη λύση μας με τον Κυβερνήτη (Kubernetes) , έναν απο τους πιο ευρέως χρησιμοποιούμενους ενορχηστρωτές υπολογιστικών συστημάτων σε περιβάλλοντα νέφους σήμερα, και δείχνουμε πως μπορούμε να επιτύχουμε υψηλότερη απόδοση σε σύγκριση με τον προεπιλεγμένο δρομολογήτη, για μια ποικιλία αντιπροσωπευτικών τύπων εφαρμογών που χρησιμοποιούνται ευρέως σήμερα.

Λέξεις Κλειδιά— υπολογιστές νέφους, διαχείριση πόρων, χρονοδρομολόγηση, Kubernetes, Interference-aware, ετερογένεια

Abstract

Nowadays, there is an ever-increasing number of workloads pushed and executed on the Cloud. To effectively serve and manage these huge computational demands, data center operators and cloud providers have embraced workload co-location and multi-tenancy as first class system design concern. In addition, the continuous advancements in the computers' hardware technology have led to a heterogeneous pool of systems lying under data center environments. Current state-of-the-art schedulers and orchestrators rely on typical metrics, such as CPU or memory utilization, for placing incoming workloads on the available pool of resources, thus, not taking into consideration the interference effects each task cause, when co-located with others, as well as the impact of systems' underlying diversity on the performance.

In this thesis, we design an interference- and heterogeneity- aware cloud orchestrator, able to efficiently schedule applications arriving at a data center on a pool of available resources. We showcase the impact of applying stress on different shared resources of two heterogeneous server systems and we propose an indicator that depicts the state of the system based on these observations. We integrate our solution with Kubernetes, one of the most widely used cloud orchestration frameworks nowadays, and we show that we can achieve higher performance compared to its default scheduler, for a variety of cloud representative workloads.

Keywords— cloud computing, resource management, scheduling, Kubernetes, interference-aware,heterogeneity

Ευχαριστίες

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

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

Ακόμα θα ήθελα να ευχαριστήσω τους γονείς μου Ανδρέα και Χριστίνα, την αδερφή μου Κατερίνα και τους φίλους μου. Η συνεχής υποστήριξή τους καθ ΄όλη τη διάρκεια της ζωής και των σπουδών μου σε ότι και αν επιχειρούσα, μου έδινε δύναμη να συνεχίσω να επιδιώκω τους στόχους μου. Τέλος, ένα μεγάλο ευχαριστώ στην κοπέλα μου Θέμιδα, η οποία με υπομονή με στήριξε σε στιγμές πίεσης και άγχους.

Acknowledgments

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

Last but not least, I would like to thank my parents Andreas and Christina, my sister Katerina and my friends. The constant support throughout my life and studies in whatever I endeavor, gave me the strength to continue pushing, trying to pursue my goals. Finally, a big thank you to my girlfriend Themis, who patiently supported me in times of stress and anxiety.

Contents

List of Figures

List of Tables

Εκτεταμένη Περίληψη

1 Εισαγωγή

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

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

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

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

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

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

2 Κυβερνήτης (Kubernetes) και Ενορχήστρωση containers

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

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

Οι εφαρμογές αφού τοποθετηθούν μέσα σε containers , τοποθετούνται σε pod τα οποία μπορούν να περιέχουν ένα ή περισσότερα containers ή και μονάδες αποθήκευσης (volumes). Ο Κυβερνήτης υποστηρίζει μια πληθώρα υπηρεσιών και παρέχει ποικίλες δυνατότητες στους χρήστες και προγραμματιστές, ευνοόντας την αυτοματοποίηση των αναγκαιών εργασιών. Μια από αυτές είνα ο ενσωματωμένος δρομολογητής εργασιών. Ο τελευταίος βασιζόμενος σε μετρικές υψηλού επιπέδου, αφαιρετικές σε επίπεδο εικονικοποίησης όπως η χρήση των επεξεργαστών και μνήμης, παίρνει αποφάσεις σχετικά με την τοποθέτηση των pods. Η διαδικάσια με την οποία οι αποφάσεις αυτές λαμβάνονται είναι η εξής. Αρχικά εξετάζονται όλοι οι υποψήφιοι κόμβοι σχετικά με τη διαθεσιμότητά τους, και την ικανότητα τους να εξυπηρετήσουν την εισερχόμενη εφαρμογή. Στη συνέχεια όσοι από αυτούς κριθούν κατάλληλοι, βαθμολογούνται με τη χρήση μιας σειράς από συναρτήσεις αξιολόγησης.

3 Ανάλυση εκτέλεσης εφαρμογών σε περιβάλλοντα με έντονη χρήση των πόρων

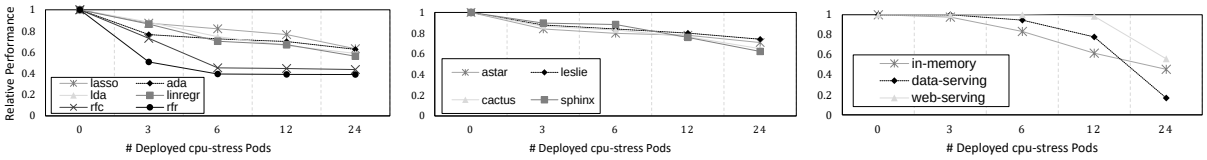
Ο Κυβερνήτης σε δοκιμή που έγινε σχετικά με τη διαχείριση καταστάσεων υψηλής πίεσης στους πόρους, απέτυχε να τοποθετήσει την εισερχόμενη εφαρμογή στον πιο κατάλληλο κόμβο, αυτόν που θα ελαχιστοποιούσε τον χρόνο εκτέλεσης της εφαρμογής προς δοκιμή. Μάλιστα, η συγκεκριμένη τοποθέτηση επέφερε καθυστέρηση $\times(-0.6873)$. Ουσιαστικά με το πείραμα αυτό, αναγνωρίσαμε την αδυναμία του Κυβερνήτη να αναγνωρίζει το είδος της πίεσης στους πόρους, και πως η έλλειψη διαφορετικών πόρων επιδρά τελικά στην απόδοση.

Χρησιμοποιώντας ως εφαρμογές πίεσης την ομάδα ibench, παρατηρήσαμε τη συμπεριφορά εφαρμογών από τις ομάδες scikit, spec 2006 και cloudsuite. Αυτές αποτελούν εκτελέσιμα αρχεία που προσομοιώνουν χαρακτηριστικά εφαρμογών που χρησιμοποιούνται ευρέως σε περιβάλλοντα νέφους. Έτσι, ασκώντας πίεση σε πόρους όπως τα διαφορετικά επίπεδα της κρυφής μνήμης, οι επεξεργαστές, αλλά και το εύρος ζώνης της διόδου που οδηγεί στη κεντρική μνήμη μπορέσαμε να παρατηρήσουμε τη συμπεριφορά των εφαρμογών, και πως η αποδοσή τους επηρεαζόταν ανάλογα με την ένταση της πίεσης αυτής. Το περιβάλλον του Κυβερνήτη, διευκόλυνε μέσω των λειτουργιών που παρέχει την εύκολη και γρήγορη αυξομείωση του όγκου εργασιών.

Η συγκομιδή των μετρικών του συστήματος έγινε με τη χρήση του Performance Counter Monitor (PCM) εργαλείου σχεδιασμένου από την Intel. Συγκεκριμένα εξάγαμε μετρικές όπως τα L2, L3 cache misses, τον αριθμό διαβασμάτων και αναγνώσεων από και προς τη κεντρική μνήμη, το ποσοστό χρήσης των επεξεργαστών, αλλά και τον αριθμό των εκτελούμενων εντολών ανα κύκλο ρολογιού των επεξεργαστών.

Οι δοκιμές έγιναν σε μια από τις δύο ομάδες επεξεργαστών του συστήματος, η οποία μοιράζεται το κοινό τελευταίο επίπεδο της κρυφής μνήμης. Πιέσαμε τα επίπεδα δύο και τρία της κρυφής μνήμης, τη δίοδο της κεντρικής μνήμης και την κεντρική επεξεργαστική μονάδα.

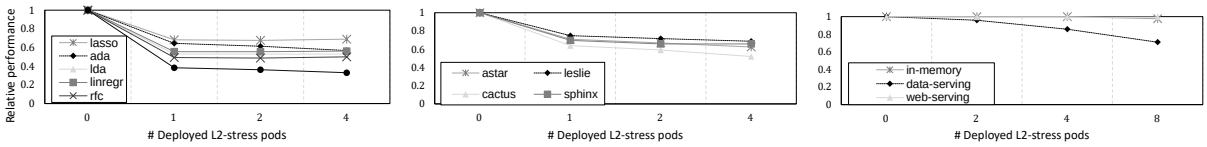
Παρατηρήθηκαν συνολικά διαφορετικές συμπεριφορές τόσο ανα ομάδα εφαρμογών όσο και μεταξύ των εφαρμογών που υπάγονται και στην ίδια ομάδα. Ενώ η επίδραση της πίεσης στη κεντρική υπολογιστική μονάδα και στο επίπεδο δύο της κρυφής μνήμης δεν είχε μεγάλη επίδραση, η πίεση στο δίοδο κεντρικής μνήμης και στο επίπεδο 3 (τελευταίο επίπεδο) της κρυφής μνήμης φάνηκε να επηρεάζει ραγδαία την απόδοση των εφαρμογών προς δοκιμή. Ακολούθως, δημιουργήσαμε κάποια σενάρια πιέσης τα οποία περιείχαν διαφορετικής έντασης πιέσεις σε περισσότερους από έναν πόρους ταυτόχρονα, με σκοπό την προσομοίωση πραγματικών συνθηκών πίεσης. Είναι αξιοσημείωτο πως η συμπεριφορά, και η επίδραση στην απόδοση είναι ανάλογη σε όλες τις ομάδες εφαρμογών. Με αυτό εννοούμε πως



(a) Scikit (b) Spec 2006 (c) Cloudsuite

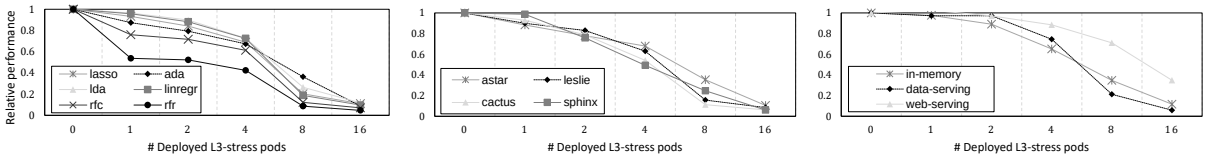
Figure 1: Επίδραση της πίεσης στις κεντρικές μονάδες επεξεργασίας, στην απόδοση των εφαρμογών

σε όλες τις εφαρμογές, τα σενάρια σε βαθμό απόδοσης κατατάσσονται με τη σειρά 6,1,4,2,3. Το γεγονός αυτό επιδυκνύει την εξάρτηση σχεδόν του συνόλου των εφαρμογών από την κατάσταση που βρίσκεται το σύστημα στο οποίο εκτελούνται, και αναδυκνείται η ανάγκη για ένα μέτρο κατάστασης του συστήματος, ικανό να κατατάξει ένα σύνολο διαφορετικών συστημάτων ανάλογα με την ικανότητα τους να φιλοξενήσουν και να αποπερατώσουν μια εισερχόμενη εφαρμογή.



(a) Scikit (b) Spec 2006 (c) Cloudsuite

Figure 2: Επίδραση της πίεσης στο επίπεδο δύο της κρυφής μνημης στην απόδοση των εφαρμογών.

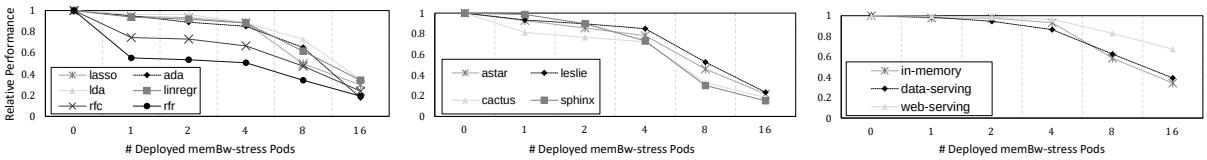


(a) Scikit (b) Spec 2006 (c) Cloudsuite

Figure 3: Επίδραση της πίεσης στο τελευταίο επίπεδο κρυφής μνήμης, στην απόδοση των εφαρμογών.

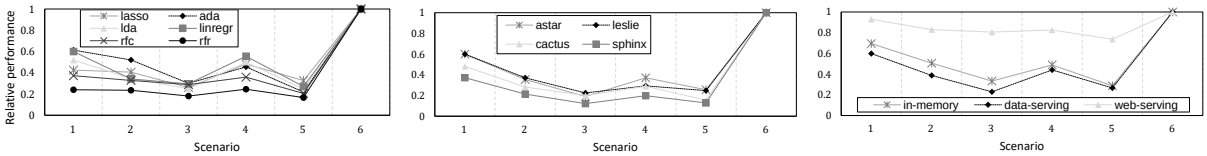
3.1 Ποσοτικοποίηση και μέτρηση πίεσης συστήματος

Η ένταση της πίεσης στα παραπάνω πειράματα καθοριζόταν από τον αριθμό των εφαρμογών ibench που εκτελούνταν στο σύστημα. Ωστόσο, στο σημείο αυτό κρίνεται απαραίτητη η διόπτευση της κατάστασης του συστήματος συνολικά με χρήση ενός ή συνδυασμόυ από μετρικές χαμηλού επιπέδου. Με στόχο την ποσοτικοποίηση της πίεσης ενός συστήματος, δοκιμάσαμε διαφορετικές μετρικές όπως το ποσοστό χρήσης των επεξεργαστών, τις εντολές ανα κύκλο επεξεργαστή, τον αριθμό των χαμένων προσπαθειών εύρεσης δεδομένων στο τελευταίο επίπεδο της κρυφής μνήμης, τον αριθμό διαβασμάτων από την κεντρική μνήμη καθώς επίσης και έναν προτεινόμενο συνδυασμό από εμάς αυτών των μετρικών. Η μετρική που



(a) Scikit (b) Spec 2006 (c) Cloudsuite

Figure 4: Επίδραση της πίεσης στο έυρος διόδου προς στη μνήμη, στην απόδοση των εφαρμογών.



(a) Scikit (b) Spec 2006 (c) Cloudsuite

Figure 5: Επίδραση διαφορετικών σεναρίων πίεσης στην απόδοση των εφαρμογών.

προτείνουμε είναι η ακόλουθη:

$$\frac{Reads + Writes}{IPC}$$

Στη συνέχεια, με σκοπό την μέτρηση ακρίβειας της νέας προτεινόμενης μετρικής, υπολογίσαμε τη συσχέτιση αυτής καθώς και άλλων μετρικών χαμηλού επιπέδου του συστήματος, με την διακύμανση της απόδοσης των εφαρμογών που εκτελέστηκαν. Σχεδόν σε όλες τις περιπτώσεις η μετρική μας παρουσίασε υψηλή συσχέτιση με την απόδοση του συστήματος, επιτυγχάνοντας υψηλή συσχέτιση ακόμη και σε σενάρια που περιελάμβαναν ταυτόχρονη πίεση διαφορετικών πόρων. Η ικανότητα αυτή της σύνθετης μετρικής να αποτυπώνει την κατάσταση του συστήματος είναι πολλά υποσχόμενη, και θεωρούμε πως μπορεί να χρησιμοποιηθεί για την εκλογή του καταλληλότερου συστήματος για την τοποθέτηση εφαρμογών σε ένα εικονικό περιβάλλον νέφους, αποτελούμενο από μεγάλες συστοιχίες υπολογιστών.

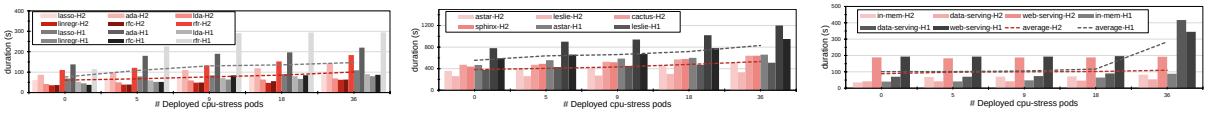
3.2 Ετερογένεια

Ακολούθως, σε μια προσπάθεια να διερευνήσουμε τα αποτελέσματα και τις επιπτώσεις της ετερογένειας σε ένα σύνθετο σύστημα υπολογιστών, εκτελέσαμε τις προηγούμενες μετρήσεις σε ένα διαφορετικό σύστημα και στη συνέχεια συγκρίναμε τις μετρήσεις μεταξύ των δύο.

Το δεύτερο σύστημα το οποίο εξετάστηκε, έχει περισσότερους πυρήνες συνολικά, μεγαλύτερη μνήμη, μεγαλύτερο το δεύτερο επίπεδο και μικρότερο το τρίτο επίπεδο της κρυφής μνήμης, περισσότερες διόδους, με μεγαλύτερο εύρος προς τη μνήμη συγκριτικά με το προηγούμενο.

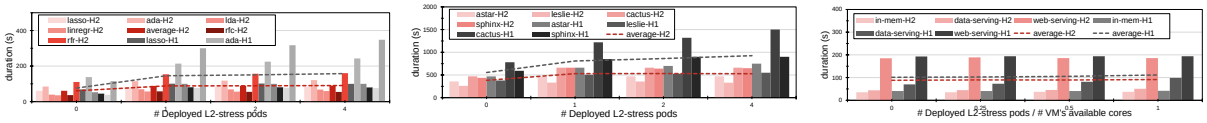
Όπως καταδεικνύνουν και τα ακόλουθα σχήματα, το δεύτερο σύστημα που εξετάστηκε, πετυχαίνει υψηλότερη απόδοση σε όλων των τύπων τις πιέσεις, σε

κάθεμια από τις δοκιμαζόμενες ομάδες εφαρμογών.



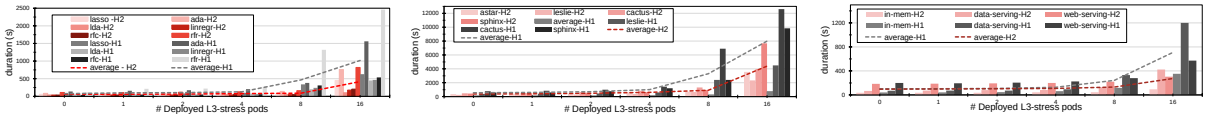
(a) Scikit (b) Spec 2006 (c) Cloudsuite

Figure 6: Επίδραση της πίεσης στις κεντρικές μονάδες επεξεργασίας, στην απόδοση των εφαρμογών



(a) Scikit (b) Spec 2006 (c) Cloudsuite

Figure 7: Επίδραση της πίεσης στο επίπεδο δύο της κρυφής μνημης στην απόδοση των εφαρμογών.

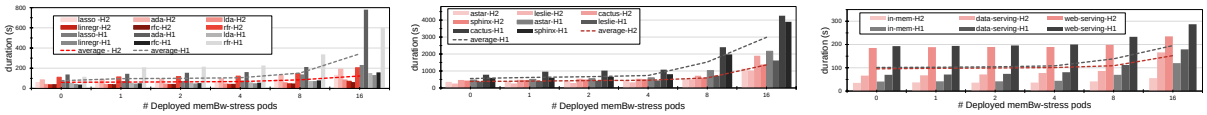


(a) Scikit (b) Spec 2006 (c) Cloudsuite

Figure 8: Επίδραση της πίεσης στο τελευταίο επίπεδο κρυφής μνήμης, στην απόδοση των εφαρμογών.

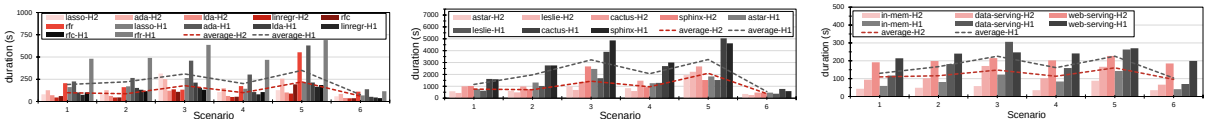
4 Σχεδιασμός ενσωματωμένου στον Κυβερνήτη δρομολογητή, ενήμερου σχετικά με την κατάσταση του συστήματος

Ο ήδη υπάρχων δρομολογήτης του Κυβερνήτη, κρίνουμε πώς είναι σχεδιασμένος με τρόπο ώστε να εξασφαλίζει κυρίως τη βιωσιμότητα μιας εφαρμογής. Από την άλλη μέσω της απομόνωσης πόρων, είναι ικάνος να διασφαλίσει καλύτερη απόδοση σχετικά με την εκτέλεση εφαρμογών, οδηγώντας όμως πολλές φορές σε υπο-χρησιμοποίηση των δεσμευμένων πόρων, γεγονός που συμβάλλει αρνητικά τόσο στο κόστος χρήστη όσο και στη δυνατότητα αύξησης της χωρητικότητας των κέντρων δεδομένων. Με κύριο γνώμονα την επίτευξη του διπλού στόχου που αφορά την ταυτόχρονη μεγιστοποίηση της απόδοσης των εκτελούμενων εφαρμογών αλλά και την μεγιστοποίηση του ποσοστού χρήσης των διαθέσιμων πόρων, προτείνουμε τον σχεδιασμό ενός δρομολογήτη, ενσωματωμένου στον Κυβερνήτη ο οποίος με χρήση της μετρικής που αναφέρουμε στην προηγούμενη ενότητα, θα τοποθετεί τις εισερχόμενες εφαρμογές στους κατάλληλους κόμβους.



(a) Scikit (b) Spec 2006 (c) Cloudsuite

Figure 9: Επίδραση της πίεσης στο έυρος διόδου προς στη μνήμη, στην απόδοση των εφαρμογών.



(a) Scikit (b) Spec 2006 (c) Cloudsuite

Figure 10: Επίδραση διαφορετικών σεναρίων πίεσης στην απόδοση των εφαρμογών.

4.1 Το πρόβλημα της δρομολόγησης

Το πρόβλημα τοποθέτησης εφαρμογών σε ομάδες κόμβων που διαμοιράζονται κοινούς πόρους θα μπορούσε να αναχθεί σε γενίκευση του Number Partitioning. Τελικά το πιο κοντινό πρόβλημα είναι αυτό του minTotal Dynamic Bin Packing. Ο στόχος εδώ είναι η ελαχιστοποίηση του βάρους κάθε ομάδας (bin). Ο συνολικός αριθμός των ομάδων είναι πεπερασμένος, και τα αντικείμενα προς τοποθέτηση φθάνουν και φεύγουν από τους κόμβους σε αυθαίρετα χρονικά διαστήματα και δεν μπορούν να αλλάξουν κόμβο αφού τοποθετηθούν. Ωστόσο στη δική μας περίπτωση ο ορισμός του βάρους κάθε αντικειμένου είναι περίπλοκος, μη σταθερός και εξαρτάται από την κατάσταση που βρίσκεται η εκάστοτε ομάδα.

4.2 Υλοποίηση

Για το σχεδιασμό του δικού μας δρομολογητή, χρησιμοποιήσαμε την ευριστική μέθοδο της τοποθέτησης κάθε εισερχόμενης εφαρμογής, στο καλύτερο ΄δοχείο΄ κάθε δεδομένη χρονική στιγμή, σύμφωνα με τα δικά μας κριτήρια βαθμολόγησης. Προσεγγίσεις γνωστών αλγορίθμων σε προβλήματα δρομολόγησης σε πολυπύρηνους επεξεργαστές, όπως αυτός του ελάχιστου χρόνου επεξεργασίας, δεν θα μπορούσαν να εφαρμοστούν στην περίπτωση μας. Αυτό συμβαίνει γιατί ο υπολογισμός του τελικού σχήματος δρομολόγησης γίνεται έκτός σύνδεσης΄ δηλαδή προηγείται του χρόνου εκτέλεσης. Κάτι τέτοιο είναι αδύνατο σε περιβάλλον όπου άγνωστος αριθμός και τύπος εφαρμογών καταφθάνουν στο σύστημα. Εκτός αυτού στη δική μας περίπτωση δεν γνωρίζουμε το χρόνο περάτωσης κάθε εφαρμογής σε αντίθεση με τον προαναφερθέντα αλγόριθμο.

Εξαγωγή μετρικών και διαμοιρασμός

Στα πλαίσια του συστήματος που σχεδιάσαμε, πρώτο στάδιο ήταν η εξαγωγή μετρικών. Οι μετρικές αυτές εξήχθησαν με τη χρήση του εργαλείου PCM της

Intel. Είναι μετρικές που αφορούν τόσο μεμονωμένες κεντρικές επεξεργαστικές μονάδες, όσο και σύνολα αυτών (socket). Οι μετρικές αυτές αφού μετρώνται, και επεξεργάζονται μερικώς σε κάθε ξεχωριστό μηχάνημα, αποστέλλονται στον κόμβο master του Κυβερνήτη. Αυτό επιτυγχάνεται με τη χρήση ενός Network File System (NFS) μεταξύ της εικονικής μηχανής στην οποία ανήκει ο κόμβος master και όλων των διαφορετικών μηχανημάτων servers που ανήκουν στην συστοιχία, η οποία ορίζεται και ως το σύστημα μας.

Μοντέλο συστήματος και μοντέλο εφαρμογών

Το πρώτο επίπεδο αφαίρεσης της υλοποίησης μας, δηλαδή το δοχείο στο οποίο θα τοποθετούνται οι εισερχόμενες εφαρμογές είναι το socket. Κάθε socket αποτελείται από μια ομάδα επεξεργαστικών μονάδων οι οποίες μοιράζονται μεταξύ τους το τελευταίο επίπεδο της κρυφής μνήμης, αλλά και τη δίοδο προς τη κεντρική μνήμη. Για κάθε τέτοια ομάδα επεξεργαστών κρατάμε συγκεκριμένα τον αριθμό των διαβασμάτων και γραψίματων από και προς την κεντρική μνήμη, το μέσο όρο του ποσοστού των πυρήνων που βρίσκονται σε ανενεργή κατάσταση και το μέσο όρο των εντολών ανα κύκλο ρολογιού που εκτελούνται σε κάθε υπολογιστική μονάδα που ανήκει στη συγκεκριμένη ομάδα. Στην περίπτωση της ετερογένειας, λαμβάνουμε ακόμη υπόψιν τον αριθμό αλλά και το έυρος των διόδων προς τη κεντρική μνήμη.

Όσον αφορά τις εφαρμογές, αυτές τις έχουμε χαρακτηρίσει κρατώντας για καθεμία το συνολικό χρόνο που διαρκεί η εκτέλεση τους, όταν δεν υπάρχει καμία πίεση στο σύστημα. Επιπλέον ο χαρακτηρισμός αυτών περιλαμβάνει και τον μέσο όρο των αναγνώσεων και διαβασμάτων από και προς τη μνήμη.

Αλγόριθμος

Ο αλγόριθμος του προτεινόμενου δρομολογητή περιέχει δύο στάδια για την επιλογή του κόμβου τοποθέτησης της εκάστοτε εφαρμογής.

Στάδιο 1ο: Στο πρώτο στάδιο της υλοποίησης μας, αφού εξαχθούν οι μετρικές για κάθε ομάδα επεξεργαστών (socket) , με χρήση μιας συνάρτησης βαθμολόγησης, επιλέγεται η ομάδα με την καλύτερη βαθμολογία. Οι επεξεργαστές της εκάστοτε ομάδας, ανήκουν σε παραπάνω από έναν κόμβους. Η συνάρτηση αυτή είναι η ακόλουθη:

$$Score_j = Stress_j \times HF_i \times DF(t)_j$$

Ως $Stress_j$ ορίζουμε την πίεση που δέχεται η ομάδα j και ορίζεται από την εξίσωση $Stress_j = \frac{Reads_j + Writes_j}{IPC_j}$. Επιπρόσθετα, γίνεται και ένας επιπλέον έλεγχος σχετικά με τη διαθεσιμότητα των επεξεργαστών της ομάδας, με χρήση

του C6-state. Ως C6-state ορίζεται το ποσοστό μιας κεντρικής μονάδας επεξεργασίας το οποίο βρίσκεται σε κατάσταση αναμονής. Αν το ποσοστό ξεπερνά μια συγκεκριμένη τιμή, θεωρούμε πως ο πόρος αυτός είναι διαθέσιμος. Σε αντίθετη περίπτωση, προστίθεται και αυτό σαν μέρος της εξίσωσης.

Με τον όρο HF_i ορίζουμε τον παράγοντα ετερογένειας του μηχανήματος i, τον οποίο εισάγουμε πειραματικά στην εξίσωση, με σκοπό να λαμβάνουμε υπόψιν τη διαφορετικότητα μεταξύ των πόρων δύο ή περισσότερων μηχανημάτων. Εδώ σαν παράγοντες χρησιμοποιήσαμε τον αριθμό των διόδων προς τη μνήμη αλλά και το εύρος αυτών, για κάθε ξεχωριστό μηχάνημα.

$$HF_i = \frac{1}{\#links \times bandwidth}$$

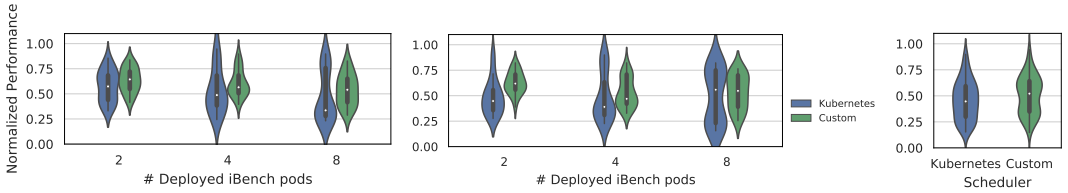
Τέλος με το $DF(t)$ ορίζουμε τον παράγοντα, ο οποίος λαμβάνει υπόψιν του εκτός από την πίεση που δέχεται ένας διαμοιρασμένος πόρος, ή ομάδα επεξεργαστών, τη διάρκεια της πίεσης αυτής.

Στο σημείο αυτό σχεδιάσαμε και εξετάσαμε διαφορετικές προσεγγίσεις για το συγκεκριμένο παράγοντα.

- Μέσος χρόνος εκτιμώμενης αποπεράτωσης των δρομολογημένων εφαρμογών: t_{avg}
- Μέσος χρόνος εκτιμώμενης αποπεράτωσης συν γραμμική πτώση της πίεσης στο διάστημα από το μέσο χρόνο μέχρι τον μέγιστο: $t_{avg} + \frac{t_{max} - t_{agv}}{2}$
- Αυξημένο βάρος στο χρόνο εκτέλεσης: t_{avg}^2
- Αυξημένο βάρος στη πίεση : $stress_j \times t_{avg}$
- Φθίνουσα εκτίμηση χρόνου, βασισμένη στη σχέση μεταξύ μέσου και μέγιστου χρόνου εκτέλεσης των δρομολογημένων εφαρμογών: $t_{max} \times (1 + e^{-\frac{t_{avg}}{t_{max}}})$

Στάδιο 2ο:

Στη συνέχεια, αφού επιλεχθεί η καταλληλότερη ομάδα επεξεργαστών, χρειάζεται να προσδιοριστεί το επόμενο επίπεδο αφαίρεσης του συστήματος, το οποίο είναι ο κόμβος ή εικονική μηχανή. Εξετάζουμε λοιπόν τους κόμβους των οποίων οι πυρήνες ανήκουν στην επικρατέστερη ομάδα. Στο στάδιο αυτό ώς κριτήριο επιλογής χρησιμοποιούμε τη διαθεσιμότητα του εκάστοτε κόμβου. Εξετάζουμε δηλαδή το ποσοστό των επεξεργαστών κάθε κόμβου που βρίσκονται σε λειτουργία αναμονής. Αξίζει να διευκρινιστεί εδώ πως ως αναμονή εννοούμε την κατάσταση κατά την οποία ο επεξεργαστής είναι ανενεργός και όχι κατάσταση όπου αναμένει άλλες λειτουργίες να εκτελεσθούν όπως για παράδειγμα μεταφορές από τη μνήμη.



(a) Πίεση στο εύρος ζώνης διόδου προς τη μνήμη

(b) Πίεση τελευταίου επιπέδου της κρυφής μνήμης

(c) Πίεση σε πολλαπλούς πόρους

Figure 11: Κανονικοποιημένη απόδοση εφαρμογών που τοποθετήθηκαν με προϋπάρχον δρομολογημένο φορτίο.

5 Αποτελέσματα και Αξιολόγηση

Στο κεφάλαιο αυτό αξιολογούμε την αποτελεσματικότητα κάθε προσέγγισης σχετικά με την απόδοση των εφαρμογών που τοποθετούνται σύμφωνα με αυτήν. Συγκεκριμένα συγκρίνουμε την υπάρχουσα υλοποίηση στον Κυβερνήτη με όλες τις διαφορετικές υλοποιήσεις που προαναφέρθηκαν. Οι δοκιμές γίνονται αρχικά σε ένα ομογενές σύστημα/μηχάνημα και στη συνέχεια σε συστοιχία αποτελούμενη από δύο ετερογενή μηχανήματα.

5.1 Ομογενές Σύστημα

Άσκηση τεχνητής πίεσης σε μία ομάδα επεξεργαστών

Στο πρώτο μέρος πειραμάτων και αξιολόγησης, εξετάσαμε τη συμπεριφορά της υλοποίησης μας, και τη λήψη αποφάσεων της σε κατάσταση κατα την οποία μια από τις δύο ομάδες επεξεργαστών δέχεται πιέσεις σε διαφορετικούς πόρους της. Αφού λοιπόν τοποθετήσουμε τις εφαρμογές πίεσης, δοκιμάζουμε να δρομολογήσουμε 25 εφαρμογές από τις ομάδες scikit-learn, cloudsuite, spec2006. Συγκρίνουμε τα αποτελέσματα αυτά, με αυτά που προκύπτουν από τη υπάρχουσα υλοποίηση δρομολογητή στον Κυβερνήτη.

Πιέζουμε με τη σειρά τη δίοδο προς τη μνήμη και το τελευταίο επίπεδο της κρυφής μνήμης και λαμβάνουμε τα αποτελέσματα που φαίνονται στα διαγράμματα 11a και 11b. Όπως φαίνεται και από τις κατανομές της κανονικοποιημένης απόδοσης των εφαρμογών, η υλοποίηση μας, γνωρίζοντας την αυξημένη πίεση σε κρίσιμους πόρους αποφεύγει την τοποθέτηση του εισερχόμενου φορτίου σε αυτούς. Έτσι επιτυγχάνει έως και 61.2% και 38.3% για πίεση έυρους ζώνης της διόδου προς τη μνήμη και του τελευταίου επιπέδου ιεραρχίας της κρυφής μνήμης αντίστοιχα. Ακόμη μια παρατήρηση είναι επίσης η μείωση της τυπικής απόκλισης της απόδοσης του συνόλου των εφαρμογών, γεγονός που την καθιστά περισσότερο προβλέψιμη.

Άσκηση τεχνητής πίεσης πολλαπλών πόρων και στις δύο ομάδες επεξεργαστών

Όπώς φαίνεται στο σχήμα 11c ,όπου παρουσιάζονται τα συνολικά αποτελέσματα και από τα 4 σενάρια που δοκιμάστηκαν. Τοποθετήθηκε διαφορετικός αριθμός από εφαρμογές πίεσης πόρων και στις δύο ομάδες επεξεργαστών και στη συνέχεια μετρήσαμε την απόδοση των επόμενων 25 εφαρμογών όταν αυτές θα δρομολογούνταν από τη δική μας προσέγγιση αλλά και από την ήδη υπάρχουσα. Η διάμεσος των κατανομών ήταν στην δική μας προσέγγιση από 4.9% έως και 80% υψηλότερη. Η μέση τιμής της διαμέσου από τις 4 διαφορετικές τιμές, ήταν κατα 16.8% υψηλότερη στη δική μας προσαρμοσμένη υλοποίηση.

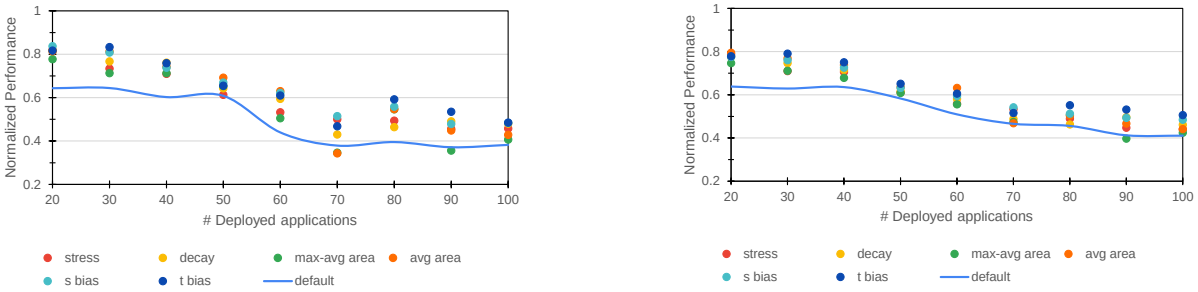
Δρομολόγηση εφαρμογών χωρίς τεχνητή άσκηση πίεσης

Στη συνέχεια, η επόμενη αξιολόγηση και σύγκριση της δικής μας προσέγγισης με την ήδη υπάρχουσα, περιλαμβάνει τη δοκιμή δρομολόγησης εφαρμογών χωρίς να προϋπάρχει κάποιο φορτίο εκ των προτέρων στο σύστημα. Δοκιμάσαμε λοιπόν τη δρομολόγηση ομάδων ερφαρμογών οι οποίες διέφεραν σε πλήθος. Συγκεκριμένα χρησιμοποιήσαμε εφαρμογές από: scikit-learn, cloudsuite και spec2006. Οι ομάδες αυτές αποτελούνταν από τυχαίο αριθμό εφαρμογών από τις παραπάνω βιβλιοθήκες και χωρίζονται σε υποομάδες, οι οποίες καταφθάνουν ανά χρονικό διάστημα κυμαινόμενο από 10 έως 50 δευτερόλεπτα. Ο συνολικός αριθμός των εφαρμογών ανα ομάδα που δοκιμάστηκαν ήταν από 20 έως και 100. Στα διαγράμματα 12a και 12b παρουσιάζονται οι διάμεσοι και οι μέσες τιμές από τις ακόλουθες υλοποιήσεις για τοποθέτηση 20 μέχρι και 100 εφαρμογών:

- Default(Kubernetes)
- Only Stress (Stress)
- S bias (s_bias)
- t bias (t_bias)
- Time decay (decay)
- Average and max area (avg_max_area)
- Average area (avg_area)

Όπως φαίνεται και από τα σχήματα, η πληθώρα των προσαρμοσμένων υλοποιήσεων του δρομολογήτη επιφέρει καλύτερα αποτελέσματα στην απόδοση των εφαρμογών, επιτυγχάνοντας μικρότερους χρόνους εκτέλεσης τόσο κατά μέσο όρο, όσο και επί του συνόλου της κατανομής των δοκιμαζόμενων εφαρμογών.

Από τα πειραματικά αποτελέσματα καλύτερη προσέγγιση για τη συγκεκριμένη περίπτωση φαίνεται να είναι η bias,η οποία παρουσιάζει καλύτερο διάμεσο σε



(a) Διάμεσος (b) Μέση τιμή

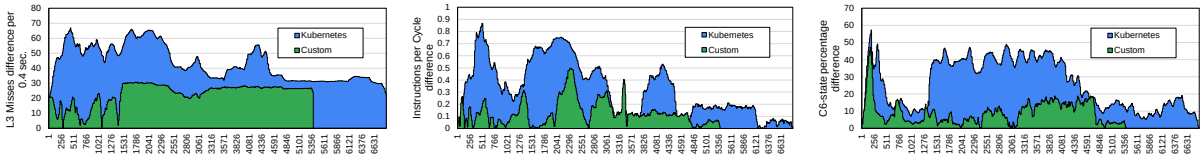
Figure 12: Σύγκριση διαφορετικών προσεγγίσεων δρομολόγησης

σχέση με την υπάρχουσα υλοποίηση κατά 26.9%, 29.3%, 25.9%, 7.6%, 38.8%, 23.4%, 49.7%, 44.0% και 26.9% για την τοποθέτηση 20, 30, 40, 50, 60, 70, 80, 90 και 100 εφαρμογών αντίστοιχα. Ανάλογα αποτελέσματα προκύπτουν και για τη μέση τιμή, η οποία είναι υψηλότερη από 10.7% μέχρι και 28.9%.

Τα καλύτερα αποτελέσματα που προκύπτουν από τη προσθήκη του παράγοντα χρόνου $DF(t)$ ενθαρρύνουν την περεταίρω μελέτη σχετικά με την καλύτερη πρόβλεψη εκτιμώμενου χρόνου, αλλά και το σχεδιασμό ενός βελτιστοποιημένου μοντέλου διαχείρισης του.

Ισορροπία χρήσης διαθέσιμων πόρων συστήματος

Οι μεγάλες συστοιχίες υπολογιστών σήμερα στα κέντρα δεδομένων όπως προείπαμε υποφέρουν από υποεκμετάλλευση των διαθέσιμων πόρων τους. Στο σχήμα 13 παρουσιάζουμε μερικές μετρήσεις του συστήματος που εξάγονται κατα τη διάρκεια μιας από τις προαναφερθέισες δοκιμές. Τα διαγράμματα παρουσιάζουν την χρήση πόρων κάθε ομάδας επεξεργαστών. Στο πρώτο διάγραμμα η ανισορροπία χρήσης του τελευταίου επιπέδου της κρυφής μνήμης διαφαίνεται από τη διαφορά των μη αποτελεσματικών ευρέσεων δεδομένων σε αυτή. Αυτό σημαίνει πώς η μια από της δύο ομάδες ήταν υπερφορτωμένη, υποβαθμίζοντας σημαντικά την απόδοση των δρομολογημένων σε αυτή εφαρμογών λόγω μεγάλης συγκέντρωσης, ενώ η άλλη είχε τον συγκεκριμένο πόρο διαθέσιμο και μη χρησιμοποιούμενο το ίδιο χρονικό διάστημα. Από την άλλη πλευρά, στη δική μας υλοποίηση, ο δρομολογητής είναι ενήμερος σχετικά με την πίεση που επικρατεί στη κρυφή μνήμη με αποτέλεσμα να κατανέμει τις εφαρμογές προς δρομολόγηση με έναν περισσότερο δίκαιο τρόπο, προσπαθώντας να διαμοιράσει το φόρτο μεταξύ των διαφορετικών μερών του συστήματος. Αντίστοιχα στα διαγράμματα 13c και 13b φαίνεται η περισσότερο ισορροπημένη χρήση των πόρων κατα τη διάρκεια της εκτέλεσης των εφαρμογών που τοποθετήθηκαν στο σύστημα από το προσαρμοσμένο δρομολογητή. Τόσο η ποσοστιαία χρήση των πυρήνων, όσο και οι εντολές που εκτελούν αυτοί ανα κύκλο ρολογιού παρουσιάζουν επίσης έναν περισσότερο ισότιμο διαμοιρασμό μεταξύ των ομάδων.



(a) L3 cache (b) IPC (c) C6-state

Figure 13: Ανισορροπία χρήσης διαθέσιμων πόρων

5.2 Ετερογενές Σύστημα

Τέλος, το τελευταίο πείραμα σύγκρισης και αξιολόγησης του δρομολογητή που σχεδιάσαμε, εξετάζει την ετερογένεια, αφού καλείται να κατανείμει εφαρμογές μεταξύ δύο διαφορετικών συστημάτων.

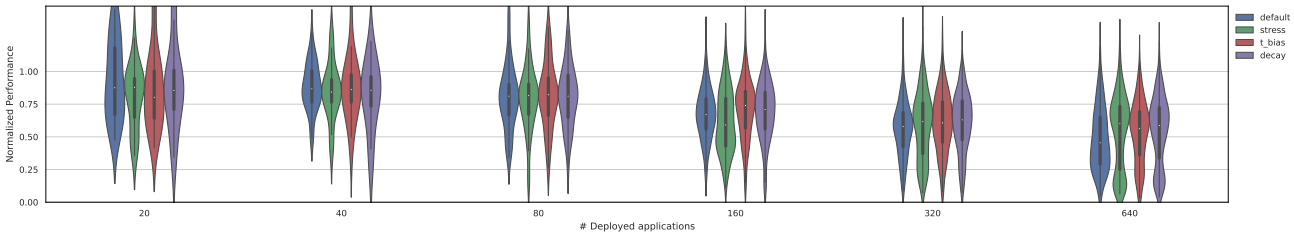


Figure 14: Σύγκριση της κατανομής της κανονικοποιημένης απόδοσης εφαρμογών μεταξύ διαφορετικών προσεγγίσεων σε μια ετερογενή ομάδα συστημάτων.

Σε σενάρια όπου η δρομολόγηση εφαρμογών ήταν λιγότερο έντονη, με αποτέλεσμα το σύστημα να είναι ικανό να τις εξυπηρετεί χωρίς να δημιουργείτε πίεση στους πόρους του, παρατηρούμε να μην υπάρχει κάποια βελτίωση στην απόδοση των εφαρμογών εν συγκρίσει με την υπάρχουσα υλοποίηση του Kubernetes. Ωστόσο, στο τελευταίο σενάριο (δρομολόγηση 640 εφαρμογών), οι προτεινόμενες απο εμάς υλοποιήσεις επιτυγχάνουν απο 23.8% μέχρι και 32% υψηλότερο διάμεσο, και απο 9.6% μέχρι και 12.2% υψηλότερη μέση τιμή.

6 Σύνοψη και Μελλοντική Δουλειά

6.1 Σύνοψη

Στη διπλωματική αυτή εργασία, αρχικά συζητήσαμε τις επιπτώσεις που επιφέρει ο ανταγωνισμός μεταξύ των εφαρμογών σε κοινόχρηστους πόρους στην απόδοση ενός συστήματος. Επιπλέον, περγράψαμε τις νέες τάσεις στα κέντρα δεδομένων σχετικά με την εικονικοποίηση και ενορχήστρωση, αλλά και την ανάγκη που αναδύεται σχετικά με την επίγνωση τόσο της συμφόρησης των κρίσιμων πόρων του συστήματος, όσο και την ετερογένεια που χαρακτηρίζει τις σύγχρονες ομάδες υπολογιστών.

Μετά απο πολλές δοκιμές, με χρήση των κατάλληλων εφαρμογών πίεσης, διαπιστώσαμε την επίδραση της πίεσης πόρων στην απόδοση των εφαρμογών και

ακολούθως προτείναμε μια μετρική που χαρακτηρίζει την κατάστση που βρίσκεται το σύστημα. Στη συνέχεια, αφού περιγράψαμε τον Κυβερνήτη, προτείνουμε μια νέα προσέγγιση δρομολογητή, την οποία και αξιολογούμε συγκρίνοντας την με την ήδη υπάρχουσα υλοποίηση. Τα αποτελέσματα που προκύπτουν είναι ενθαρρυντικά καθώς βελτιώνεται η απόδοση των δρομολογούμενων εφαρμογών σε διάφορα σενάρια.

6.2 Μελλοντική δουλειά

Σαν μελλοντική δουλειά, προτείνουμε από προγραμματιστικής πλευράς, την μελέτη και την ανάπτυξη μιας βάσης δεδομένων που θα επιταχύνει την αποκόμιση μετρικών του συστήματος. Τέλος, από ερευνητικής πλευράς, προτείνουμε τη μελλοντική χρήση νευρωνικού δικτύου για την αποτελεσματικότερη εύρεση της συνάρτησης αξιολόγησης του δρομολογητή. Επιπλέον, τεχνολογίες όπως το Cache Allocation Technology και το Running Average Power Limit θα επέτρεπαν τη διαχείριση των πόρων του συστήματος κατά τη διάρκεια της εκτέλεσης, με σκοπό τόσο την αποτελεσματικότερη χρήση τους, όσο και την βελτίωση της συνολικής ενεργειακής απόδοσης του εξεταζόμενου συστήματος.

Chapter 1

Introduction

1.1 Cloud Computing

Cloud Computing has transformed a large part of the IT industry and has radically changed the way millions of users and organizations are using the Internet. Moreover, companies with large batch-oriented tasks can get results as quickly as their programs can scale. The adoption of cloud computing has seen explosive growth, both at consumer and enterprise levels and will continue to rise in the future [1]. The evolution and endorsement of container-based virtualization technology, as well as the advantages that the Cloud computing offers both to users and operators, have acted as enablers towards this direction. Nowadays, it is easier than ever to deploy any application in the Cloud, running in the operating system of your choice. Also, this revolutionary technology made available to users the pay as you go feature while economies of scale are enabled for data-center operators who are sharing their across several users. In addition, an upfront commitment by cloud users is eliminated, thereby allowing companies to start small and increase hardware resources only when there is an increase in their needs.

Cloud provides service-oriented architecture which advocates "everything as a service". Cloud-computing providers offer their "services" according to different models, of which the three standard models are Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS). "Infrastructure as a service" (IaaS) refers to online services that provide high-level APIs used to deference various low-level details of underlying network infrastructure like physical computing resources, location, data partitioning, scaling, security, backup etc. A hypervisor runs the virtual machines as guests. Pools of hypervisors within the cloud operational system can support large numbers of virtual machines and the ability to scale services up and down according to customers' varying requirements. On the other hand, PaaS is the capability provided to the consumer to deploy onto the cloud infrastructure consumer-created or acquired applications created using programming languages, libraries, services,

and tools supported by the provider. The consumer does not manage or control the underlying cloud infrastructure including network, servers, operating systems, or storage, but has control over the deployed applications and possibly configuration settings for the application-hosting environment. Finally, Saas is the service provided to the consumer to use the provider's applications running on a cloud infrastructure. The applications are accessible from various client devices through either a thin client interface, such as a web browser (e.g., web-based email), or a program interface. Again the consumer does not manage the underlying cloud infrastructure, or even individual application capabilities, with the possible exception of limited user-specific application configuration settings.

1.2 Data Centers concerns: shared resources, Interference, under-Utilization and Heterogeneity

What we call cloud, is the combination of hardware relying inside data centers and the appropriate software to make use of this hardware, provided by Cloud providers. Thanks to virtualization, someone can create useful IT services using resources that are traditionally bound to hardware [2]. That makes resources more agile, allowing them to serve multiple functions. It allows user to use a physical machine's full capacity by distributing its capabilities among many users or environments. The legacy servers, dedicated to only one task are no longer a problem for any organization or business. Hypervisors offer this functionality, which can sit on top of an operating system or get installed directly onto hardware (like a server), which is how most enterprises virtualize.

In the history of computing, the transition from single-core Central Processing Units (CPUs) to multi-core processors was definitely a breakthrough. This new technology made it possible to run multiple applications or even multiple threads of a process in the same machine concurrently. That innovation had a great impact on the performance of computing tasks. However, a new issue has started to arise. This issue is the shared resources competition between the different consumers, called interference. While cloud is becoming more and more popular, the amount of applications deployed and executed, competing for shared resources usage, was also radically increased. The increment in the amount of workloads uploaded and executed on the cloud, have forced data-center (DC) operators and cloud providers, such as Google Cloud Platform [3] and Amazon EC2 [4], to embrace workload co-location and multi-tenancy as first class system design concern. Although hypervisors and virtualization provide a higher level of abstraction between the physical machines and the users, it also makes them unaware of critical information about the hardware, which

will lead in suboptimal use of the physical resources provided.

The current trend in computer architecture design today is an isolated memory space for Level 1 and Level 2 cache for every physical core. Due to Hyper-Threading,a core is consisted of two separate hardware threads sharing those two resources spaces. Furthermore, physical cores belonging to the same socket are sharing the same Last Level Cache (LLC) or Level 3 cache.

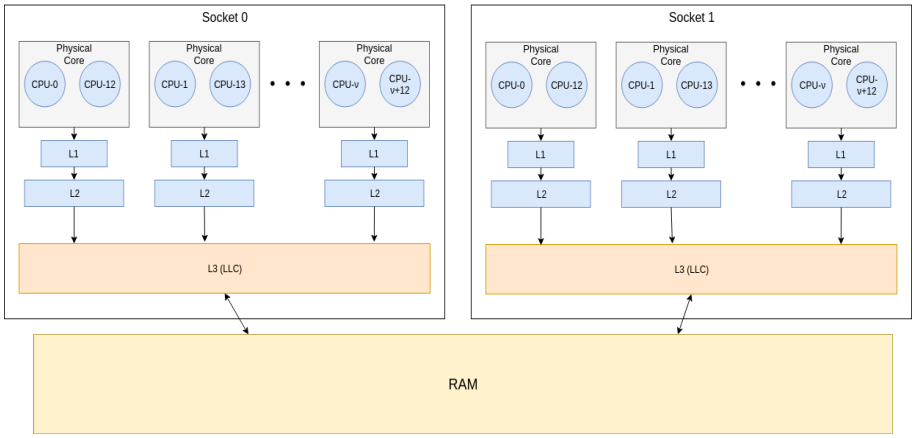


Figure 1.1: Memory System Architecture

In general, interference describes such phenomena in any shared resources such as cache and memory occupancy, memory and network bandwidth and others. Today, system performance is increasingly coupled to cache hierarchy design. Cache interference describes the phenomenon when two or more processors request data from the cache. Cache in those cases in order to serve both competitors requests, due to its restricted space, fetches and writes back data from/to the main memory. Because of the finite amount of cache in every level, when a cache miss is happening due to an application's demand in data, useful blocks that are used by other application(s) in the same cache space are written back to the main memory (RAM). The read/write operations from/to the RAM are expensive in terms of energy and delay, slowing down the whole system. Because of the interference, those operations happen more frequently, causing a great reduction in performance. Cache tends to be a performance bottleneck because of high network and memory latency [5], and a better resource management is a promising, worth to be researched topic.

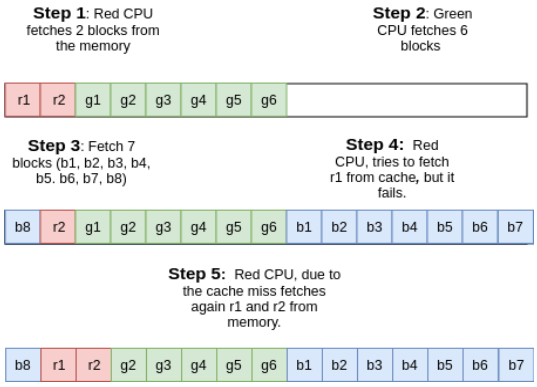


Figure 1.2: LLC Interference

In figure 1.2, cache interference is illustrated. When the cache space is filled up, new blocks that are needed, will replace the old ones based on the cache's replacement policy (e.g. fifo, lifo, lru and others). Because of that, these data that were written back to memory will be needed to be fetched again, even if the application requiring them does not use any other data than that. For example, as it is illustrated in figure 1.2, in Step 1, red application needs data from memory, so it fetches the required data r1 and its subsequent block r2 from memory. Afterwards, green application fetches blocks g1-h6. When the blue application is executed, it requests data, they are fetched from memory and they write r1 block back to memory. Next, when red application tries to use the recently fetched r1, another cache miss is taking place. This is a miss caused by interference, because while red application was keeping using the already fetched data, co-scheduled applications caused contention competing for cache space. This situation becomes even intensive when more and more applications are running using the same shared resource. Such competition may carry on for great amount of time, thus causing great performance degradation.

In an attempt to elevate the performance of the systems, in other works to keep users satisfied, cloud providers currently provide users with elasticity and resizability of their computing capacity, leading to a dynamic provisioning of resources. This flexibility in resources scalability has led users to request more and more resources in order to satisfy the Quality-of-Service (QoS) requirements of their latency-sensitive workloads. However, even at large companies like Microsoft and Google the average utilization is typically under 50% [6]. Moreover, Mozilla's and VMWare's data centers were reported to operate at 6% and at 20-30% utilization respectively [7]. Underutilized servers contribute to expenses and limit the scaling of the data-center. Web service providers have identified as a critical design goal the improvement of utilization in modern warehouse-scale computers in order to reduce the total cost of ownership [8]. On the other hand, things get worse in cluster managers and orchestrators that enable cluster sharing between workloads. In a production cluster at Twitter, CPU utilization

is below 20% while resource reservations reach up to 80% of total capacity [9]. Cluster managers fail to reserve the right amount of resources. Finally, the mature Google cluster manager Borg [10], achieves 25-35% and 40% CPU and memory utilization respectively, while reserved resources are 75% and 60% at the same time.

On top of that, the continuous evolution of hardware technologies and generations forces data-center operators to repeatedly upgrade their underlying infrastructure, in order to keep up with the latest advancements in technology and also allow cloud providers to supply the best (QoS) to their users, leading to clusters with various different server configurations. From the above, it is evident that, multi-sharing and multi-diversity of resources can cause serious degradation on the performance of running applications, thus the need for interference- and heterogeneous-aware scheduling of incoming workloads on a cluster is indispensable.

1.3 Container Orchestration with Kubernetes

The current trend, at enterprise level, for the scheduling of arriving workloads on a pool of available resources is through container orchestrators, such as Kubernetes [11] or Mesos [12]. The latest advancements and performance improvements of container-based virtualization [13] have driven companies to transform the way they develop and deploy their applications, converting them to "cloud-native", containerized microservices. Even though container orchestrators provide major benefits, such as ease of use and deployment, abstraction of resources, scaling and others, the scheduling policies they follow are naive relying on simple metrics, like CPU or memory utilization, neglecting interference effects, overlooking the specifications of the underlying infrastructure and the nature of the imposed stress on the shared resources, offering a *reservation-centric* approach in cluster management.

Although Kubernetes is a very complex project, being continuously improved by its large community, is currently unaware of some critical low-level resources. As Kubernetes manages virtual machines (VMs), it is only (sometimes) aware of the virtual CPU and memory usage. While, it uses those information so as to make scheduling decisions for the incoming applications (pods), it fails to identify low-level contention in resources like cache and memory bandwidth.

1.4 Thesis Overview

In this thesis, we explore the impact of *interference-awareness* in scheduling process. We identify an inefficient and trivial pod placement decision mak-

ing conducted by Kubernetes scheduler, lacking from useful information about system state. In chapter 4, we measure and analyze the impact of different low-level resource contention on the performance of various applications. Based on our observations, we propose a *score function*, attempting to reflect the condition of a server system lying under a virtualized environment. In chapter 5, we present our custom framework, which uses the previous proposed score as a node prioritization function in order to schedule jobs from different benchmark classes on a Kubernetes cluster. We also present an attempt to introduce the time factor in application initial scheduling and heterogeneity both in intra and inter-server level. With this approach we are trying to tackle the problem of under-utilization of cluster resources due to contention-agnostic schedulers. Finally, in chapter 6, we evaluate our proposed framework, being compared with default Kubernetes scheduler, across different stressing scenarios and workloads.

Chapter 2

Related Work

2.1 Metrics Collection

2.1.1 Rusty

Rusty [14] is a predictive monitoring system, able to address the aforementioned challenges using Long Short-Term Memory networks to enable fast and accurate runtime forecasting of key performance metrics and resources stresses of cloud-native applications under interference. Rusty is lightweight and achieves high prediction accuracy, i.e. average R^2 value of 0.98, thus forming a promising solution for runtime predictive resource allocation.

2.1.2 Bubble-Flux

Bubble-Flux [15] is an integrated dynamic interference measurement and online QoS management mechanism that provides accurate QoS control and maximizes server utilization. It is consisted of two parts. The first one, Dynamic Bubble, measures the instantaneous pressure on the shared hardware resources and predict how the QoS of a latency-sensitive job will be affected by potential co-runners. Secondly, using an online Bubble Flux Engine, monitors the QoS of the latency-sensitive applications and controls the execution of batch jobs to adapt to load changes, in order to deliver satisfactory QoS.

2.1.3 Other approaches

Several approaches have been discussed regarding to hypervisor-based monitoring [16]. Open-sourced services like Prometheus [17], and the Elasticsearch, fluentd and Kibana (EFK) stack [18] provide well-organized systems for metrics logging, aggregation and querying. Nevertheless, metrics acquired from such collectors are naive, not able to reveal the real system state. They extract metrics mostly used for alert generation, security insight, providing a brief overview of the system's condition. As a result the resource under contention

cannot be identified and the root cause of application degradation remains unmanageable. Contrarily, low-level metrics, which describe micro-architectural events, are capable of providing useful information regarding to the resource under contention, namely the origin of system's inability to serve workloads' needs efficiently.

2.2 Application Scheduling

2.2.1 Kubernetes

State-of-the-art orchestrators like Kubernetes [11] and Mesos [12] rely on unsophisticated metrics, like CPU and memory utilization, to manage node availability and workload scheduling. Using agents, Kubernetes extracts aggregated metrics from the worker nodes of the cluster. Afterwards it uses a two-level approach. After examining all available nodes, it selects only the feasible ones. The nodes remaining are evaluated through a variety of priority functions which determine their viability. However, those naive metrics are not able to indicate the condition of a system experiencing interference phenomena.

2.2.2 Mage

Mage [19] is an interference- and heterogeneity-aware runtime that leverages online data mining to explore the space of application placements, and determine the one that minimizes interference between co-resident applications. In addition, it continuously monitors and determines whether alternative placements would prove beneficial, taking into account the overhead of migration.

2.2.3 Medea

Medea [20] is a cluster scheduler designed for the placement of long- and short-running containers. It captures interactions among containers within and across applications, following a two-scheduler design: (i) for long-running applications, it applies an optimization-based approach that takes into consideration constraints and global objectives; (ii) for short-running containers, it uses a traditional task-based scheduler.

2.2.4 Paragon

Paragon [21] is an online interference- and heterogeneity-aware, scalable datacenter scheduler. It is derived from analytical methods and instead of profiling each application in detail, it leverages information already known by applications previously see. Using collaborative filtering techniques, to quickly classify

an unknown workload with respect to heterogeneity and interference, by identifying similarity to previously scheduled applications. Afterwards Paragon greedily schedules aiming to maximize the performance of the applications and the server utilization.

2.3 Resource Allocation

2.3.1 Quasar

Quasar [9] is a cluster management system that increases resource utilization, while providing high application performance. Quasar does not rely on resource reservations, but instead it determines the right amount of resources to meet QoS constraints expressed by the user. Second, Quasar uses classification techniques to determine the imact of the type and the amount of resources and interference on performance of each workload. Moreover, it adjusts resource allocation and assignment as needed during execution.

2.3.2 Other approaches

In addition various hardware-related techniques, including cache partitioning have been explored to decrease inter-application contention. [22, 23, 24]

2.4 Our Approach

Our approach focuses on global optimization objectives. Using a universal approach for every kind of workload behavior and duration aiming to maximize resource utilization and minimize application execution delays provoked by interference phenomena. The node condition and prioritization is based on micro-architecture metrics extraction from the physical, underlying server, surpassing virtualization resources abstraction level.

Chapter 3

Kubernetes, a Container Orchestrator

The framework we will analyze in the next pages is called Kubernetes (κυβερνήτης, Greek for "governor", "helmsman" or "captain")[1]

3.1 Docker containers and Orchestration

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

[1] "What is Kubernetes?". Kubernetes. Retrieved 2017-03-31.

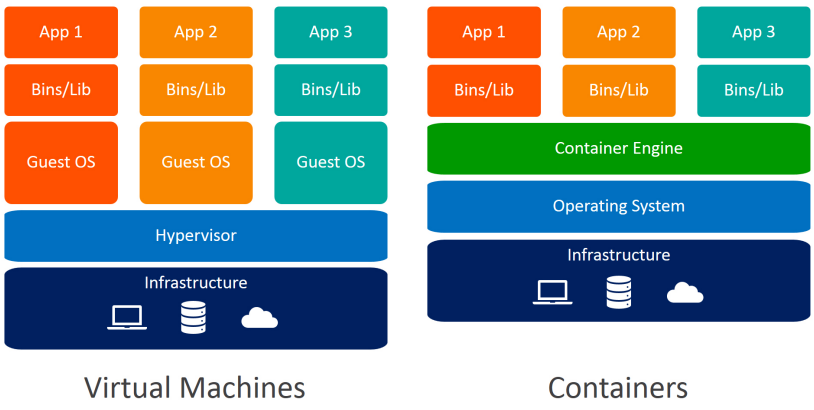


Figure 3.1: Virtual Machines and Containers [2]

3.1.1 Virtual Machines

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

3.1.2 Containers

A container is a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another. A Docker container image is a lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries and settings.

Container images become containers at runtime and in the case of Docker containers - images become containers when they run on Docker Engine. Available for both Linux and Windows-based applications, containerized software will always run the same, regardless of the infrastructure. Containers isolate software from its environment and ensure that it works uniformly despite differences for instance between development and staging. Containers technology is:

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

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

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

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

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

3.1.3 Orchestration

The answer to the previous question about which virtualization technology is better to use is that they should be used both. They are in fact complementary technologies. Containers support VM-like separation of concerns but with far less overhead and far greater flexibility. As a result, containers have reshaped the way people think about developing, deploying, and maintaining software. In such a hybrid containerized architecture, the different services that constitute an application are packaged into separate containers and deployed across a cluster of virtual machines as illustrated in figure 3.2 [25].

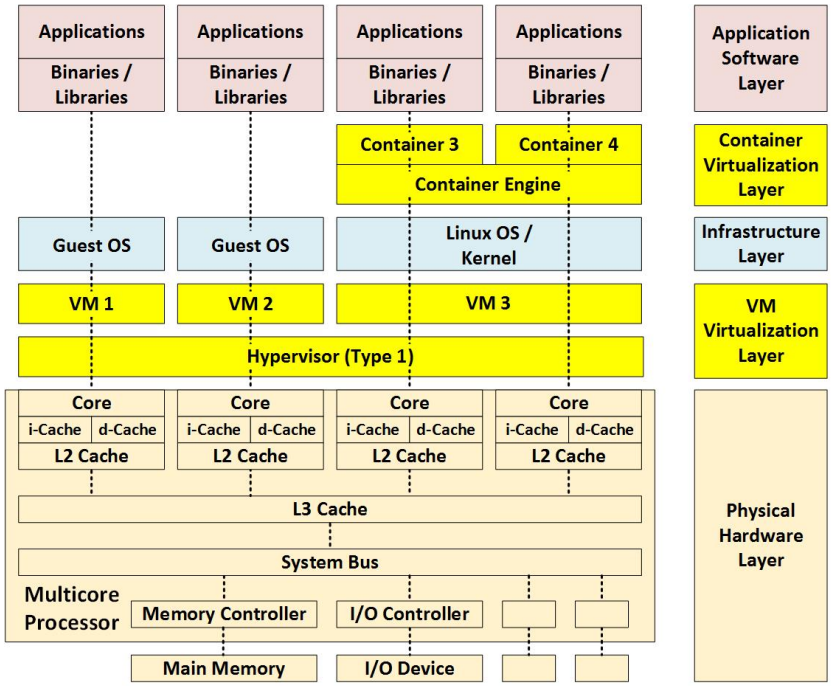


Figure 3.2: Hybrid Containerized Architecture [3]

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

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

3.2 Kubernetes Master Node(s) Components

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

□ *kube-apiserver:* Component on the master that exposes the Kubernetes

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

API. It is the front-end for the Kubernetes control plane.

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

- *kube-scheduler:* Component on the master that watches newly created pods that have no node assigned, and selects a node for them to run on. It will be discussed more thoroughly later on.

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

3.3 Kubernetes Worker Node(s) Components

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

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

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

- *Container Runtime:* The container runtime is the software that is responsible for running containers. (Docker in our case)

3.3.1 Other Important Addons

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

3.4 Kubernetes Architecture

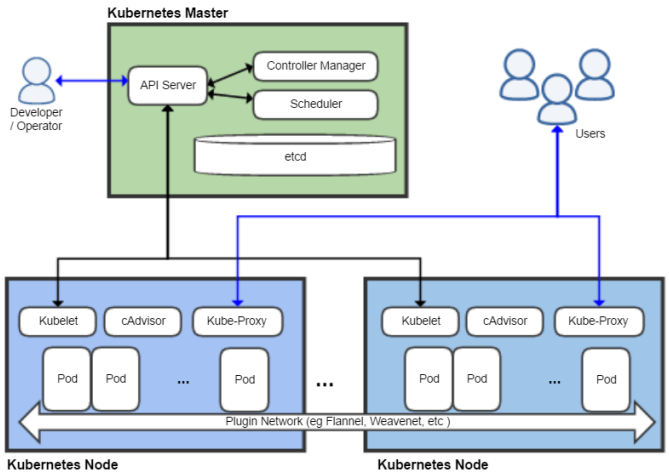


Figure 3.3: Kubernetes Architecture [4]

Kubernetes's architecture makes use of various concepts and abstractions. Some of these are variations on existing, familiar notions, but others are specific to Kubernetes. As illustrated in 3.3 and described before, a Kubernetes cluster is consisted of Nodes. Those nodes are separated into two groups, either Master or Worker nodes. Workloads are executed in Worker Nodes. The entities sorted by their abstraction level are containers, pods and services, deployments and nodes.

3.4.1 Cluster

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

3.4.2 Nodes

Each cluster contains Kubernetes nodes. Nodes might be physical machines or VMs. Again, the idea is abstraction: Whatever the application is running on,

[4]https://en.wikipedia.org/wiki/Kubernetes
[5]https://kubernetes.io/fr/docs/tutorials/kubernetes-basics/create-cluster/cluster-intro/

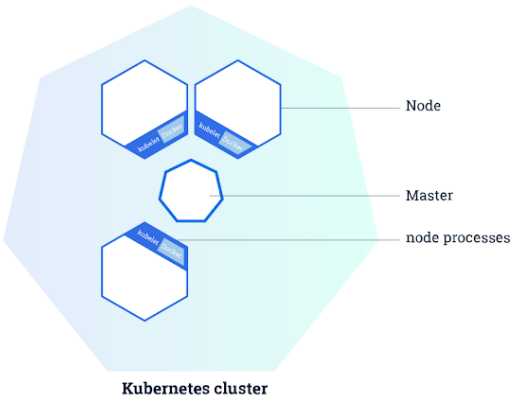


Figure 3.4: Cluster-Node abstraction level [5]

Kubernetes handles deployment on that substrate. These Nodes can be either Master Nodes or Worker Nodes. Worker nodes are the machines where the applications (containers) run on. An node with its components is presented in figure 3.5.

3.4.3 Deployment

As it is described in Kubernetes documentation, a desired state is described in a Deployment, and the Deployment controller changes the actual state to the desired state at a controlled rate. Deployments are defined to create new `ReplicaSets`, or to remove existing Deployments and adopt all their resources with new Deployments. This object offered the easily manageable scalabilty, so as to increase or decrease accordingly the required stress levels, just by changing the replicas of the pods created. In the following snippet a sample deployment is presented:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: l3
spec:
  selector:
    matchLabels:
      app: l3
  replicas: 1
  template:
    metadata:
      labels:
        app: l3
    spec:
      containers:
      - name: l3-container
        image: iwita/microlab:x1
        command: ["/bin/sh","-c"]
        args:
          - <bash-command>
        ports:
        - containerPort: 80
        imagePullPolicy: Always
      imagePullSecrets:
```

```
25        - name: regcred
```

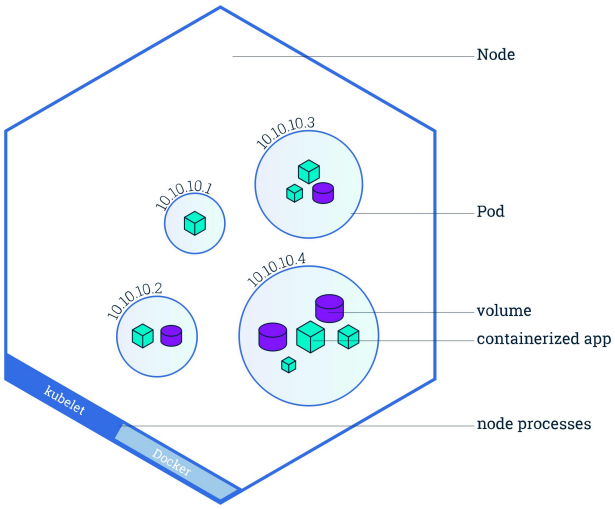


Figure 3.5: Node-Pod-Container abstraction levels [6]

3.4.4 Pods

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

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

3.4.5 Service

Kubernetes Pods are mortal. They are born and when they die, they are not resurrected. If you use a Deployment to run your app, it can create and destroy

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

Pods dynamically (e.g. when scaling out or in). Each Pod gets its own IP address, however the set of Pods for a Deployment running in one moment in time could be different from the set of Pods running that application a moment later. This leads to a problem: if some set of Pods (call them "backends") provides functionality to other Pods (call them "frontends") inside your cluster, how do those frontends find out and keep track of which IP address to connect to, so that the frontend can use the backend part of the workload? A Service is an abstract way to expose an application running on a set of Pods as a network service. Kubernetes gives pods their own IP addresses and a single DNS name for a set of pods, and can load-balance across them.

3.5 Kubernetes Resources

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

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

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

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

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

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

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

```
apiVersion: v1
kind: Pod
metadata:
  name: frontend
spec:
  containers:
  - name: db
    image: mysql
    env:
    - name: MYSQL_ROOT_PASSWORD
      value: "password"
    resources:
      requests:
        memory: "64Mi"
        cpu: "250m"
      limits:
        memory: "128Mi"
        cpu: "500m"
  - name: wp
    image: wordpress
    resources:
      requests:
        memory: "64Mi"
        cpu: "250m"
      limits:
        memory: "128Mi"
        cpu: "500m"
```

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

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

Quality of Service (QoS)

Compressible Resources Guarantees (CPU) [8] Pods are guaranteed to get the amount of CPU they request, they may or may not get additional CPU time (depending on the other jobs running). This isn't fully guaranteed today because cpu isolation is at the container level. Pod level cgroups will be introduced soon to achieve this goal. Excess CPU resources will be distributed based on the amount of CPU requested. For example, suppose container A requests for 600 milli CPUs, and container B requests for 300 milli CPUs. Suppose that both containers are trying to use as much CPU as they can. Then the extra 100 milli CPUs will be distributed to A and B in a 2:1 ratio Pods will be throttled if they exceed their limit. If limit is unspecified, then the pods can use excess CPU when available.

Incompressible Resources Guarantees: Pods will get the amount of memory they request, if they exceed their memory request, they could be killed (if some other pod needs memory), but if pods consume less memory than requested, they will not be killed (except in cases where system tasks or daemons need more memory). When Pods use more memory than their limit, a process that is using the most amount of memory, inside one of the pod's containers, will be killed by the kernel.

QoS Classes

- If limits and optionally requests (not equal to 0) are set for all resources across all containers and they are equal, then the pod is classified as **Guaranteed**.
- If requests and optionally limits are set (not equal to 0) for one or more resources across one or more containers, and they are not equal, then the pod is classified as **Burstable**. When limits are not specified, they default to the node capacity.
- If requests and limits are not set for all of the resources, across all containers, then the pod is classified as **Best-Effort**.

Pods will not be killed if CPU guarantees cannot be met (for example if system tasks or daemons take up lots of CPU), they will be temporarily throttled. Memory is an incompressible resource and so let's discuss the semantics of memory management a bit.

[8] https://github.com/kubernetes/community/blob/master/contributors/design-proposals/node/resource-qos.md

- *Best-Effort* pods will be treated as lowest priority. Processes in these pods are the first to get killed if the system runs out of memory. These containers can use any amount of free memory in the node though.
- *Guaranteed* pods are considered top-priority and are guaranteed to not be killed until they exceed their limits, or if the system is under memory pressure and there are no lower priority containers that can be evicted.
- *Burstable* pods have some form of minimal resource guarantee, but can use more resources when available. Under system memory pressure, these containers are more likely to be killed once they exceed their requests and no Best-Effort pods exist.

3.6 Kubernetes Scheduling

The Kubernetes Scheduler is a core component of Kubernetes: After a user or a controller creates a Pod, the Kubernetes Scheduler, monitoring the Object Store for unassigned Pods, will assign the Pod to a Node. Then, the kubelet, monitoring the Object Store for assigned Pods, will execute the Pod. [9] For each unscheduled Pod, the Kubernetes scheduler tries to find a node across the cluster according to a set of rules. There are two steps before a destination node of a Pod is chosen. The first step is filtering all the nodes and the second is ranking the remaining nodes to find a best fit for the Pod.

3.6.1 Node Filtering

First, the Scheduler determines the set of feasible placements, which is the set of nodes that meet a set of given constraints. All filter functions must yield true for the Node to host the Pod. The following constraints called predicates are set active by default:

NoDiskConflictPred: NoDiskConflict evaluates if a pod can fit due to the volumes it requests, and those that are already mounted. If there is already a volume mounted on that node, another pod that uses the same volume can't be scheduled there.

General Predicates: In this general predicate are included some of the major predicates of kubernetes scheduler. *PodFitsPorts* is a default predicate, where fit is defined based on the absence of port conflicts. Furthermore, *PodFitsResourcesPred*, according to which, fit is determined by resource availability. The

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

HostNamePred determines the fit by the presence of the Host parameter and a String match.

MatchNodeSelectorPred: In case a Node Selector is defined in the pod creation, fit is determined at this stage based on this node selector query.

NoVolumeConflictPred: Fit is determined by volume zone requirements.

Max{EBS,GCEPD,AzureDisk,CSI}VolumeCountPred: Fit is determined by whether or not there would be too many {EBS,GCEPD,AzureDisk,CSI} volumes attached to the node.

MatchInterPodAffinity: In this predicate, fit is determined by inter-pod affinity. Inter-pod affinity and anti-affinity allow you to constrain which nodes your pod is eligible to be scheduled based on labels on pods that are already running on the node rather than based on labels on nodes. The rules are of the form "this pod should (or, in the case of anti-affinity, should not) run in an X if that X is already running one or more pods that meet rule Y". Y is expressed as a LabelSelector with an optional associated list of namespaces; unlike nodes, because pods are namespaced (and therefore the labels on pods are implicitly namespaced), a label selector over pod labels must specify which namespaces the selector should apply to.

CheckNode{DiskPressure, MemoryPressure, PIDPressure, Condition}: Fit is determined by node {disk pressure,memory pressure,pid pressure, conditions}. Node conditions include: not ready state, network unavailable or out of disk.

PodToleratesNodeTaintsPred: Fit is determined based on whether a pod can tolerate all of the node's taints. Taints and tolerations work together to ensure that pods are not scheduled onto inappropriate nodes. One or more taints are applied to a node; this marks that the node should not accept any pods that do not tolerate the taints. Tolerations are applied to pods, and allow (but do not require) the pods to schedule onto nodes with matching taints.

However there are a few more of them ready to be set for usage at any Kubernetes custom deployment such as *CheckServiceAffinityPredicate*.

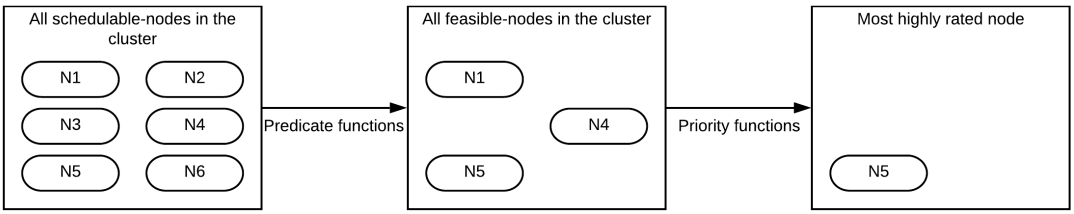


Figure 3.6: Node filtering and ranking

As someone can see in the picture above, in the first step the nodes are filtered, and only the feasible ones, that satisfy the predicates are proceeded in the latter steps.

Kubernetes Scheduler uses this technique for 2 reasons. Firstly, it needs to make sure that no pod/deployment will be scheduled in a Node that is unable to handle it taking into account its Quality of Service as well which can be declared with a few configurations in the yaml file that creates the pod. Secondly, that way it will run the second part of the algorithm (prioritization functions) across a much less set of nodes, which will consume less system resources and less time.

There are 2 types of conditions that describe those predicates

- Schedulability and Node Conditions: These conditions are accounted for via taints and tolerations
- Resource Availability: These types of functions deem a Node feasible based on the Pod's resource requirements compared with the Node's resource available resources.

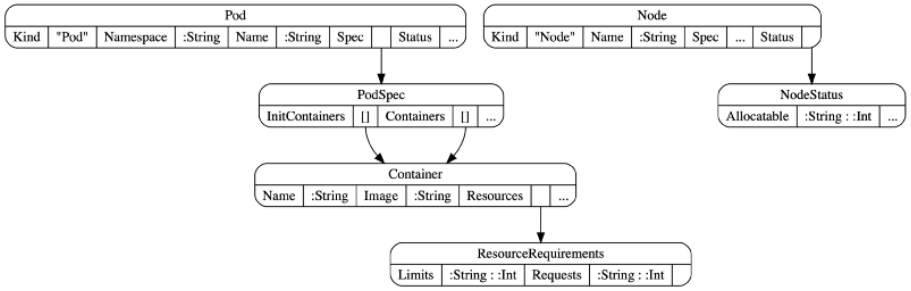


Figure 3.7: Check for resource availability [10]

3.6.2 Node Prioritizing

After the filtering, with only the feasible Nodes remaining, Kubernetes scheduler using a set of predefined rating functions, determines the viability of each Node. The Pod will be scheduled in the one with the highest viability.

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

The rating for each Node derives from the summation of the weighted scores of each priority function:

Summary of Key Notations for Final Score Calculation	
Notation	Definition
p_i	Priority function i
w_i	Weight of Priority function i
k	Number of feasible nodes to host the current pod
n_j	Node j being evaluated

$$Score(n_j) = \sum_{i=1}^{k} w_i \times p_i, \ \forall j \in feasibleNodes$$

Summary of Key Notations for Priority Calculation	
Notation	Definition
$Cmem_j$	Total Memory Capacity of node j
$Rmem_j$	Total Memory requested from node j
$Ccpu_j$	Total Milli CPUs Capacity of node j
$Rcpu_j$	Total Milli CPUs resource requested from node j
s_j	Score of node j being evaluated

The priority functions that are set by default in a Kubernetes installation are the following:

ServiceSpreading: *ServiceSpreadingPriority* is a priority that spreads pods by minimizing the number of pods belonging to the same service on the same node.

MostRequested: This a cluster autoscaler-friendly function. It gives used nodes higher priority, based on their declared resources usage.

$$s_i = \frac{memScore_j + cpuScore_j}{2}$$

$$memScore_j = \frac{Rmem_j}{Cmem_j}, cpuScore_j = \frac{Rcpu_j}{Ccpu_j}$$

RequestedToCapacityRatio: *RequestedToCapacityRatioPriority* is a configurable priority function that assigns different scores in the node based in his resources usage. By default, its behaviour is similar to the *LeastRequestedPriority* as it assigns 1.0 to resource when all capacity is available and 0.0 when requested amount is equal to capacity. Concluding, this function converts the scoring assignments to a linear function and returns node's score.

SelectorSpread: This function spreads the pods across hosts, considering pods belonging to the same service, replica set or StateFulSet. It favors nodes that have fewer existing matching pods. Trying to satisfy kubernetes principles, it spreads replicate pods across difference nodes so as to achieve greater availability when a node fails.

InterPodAffinity: *InterPodAffinityPriority* computes a sum by iterating through the elements of *weightedPodAffinityTerm* and adding the *weight* variable to the sum if the corresponding PodAffinity is satisfied for that node. The node(s) with the highest sum are the most preferred.

LeastRequested: It is a function that favors nodes with fewer requested resources. It calculates the percentage of memory and CPU requested by pods scheduled on the node, and prioritizes based on the minimum of the average of the fraction of requested to capacity. It's the opposite function from the *MostRequested*.

$$s_i = \frac{memScore_j + cpuScore_j}{2}$$

$$memScore_j = 1 - \frac{Rmem_j}{Cmem_j}, cpuScore_j = 1 - \frac{Rcpu_j}{Ccpu_j}$$

NodeAffinity: Node affinity is conceptually similar to nodeSelector – it allows you to constrain which nodes your pod is eligible to be scheduled on, based on labels on the node.

There are currently two types of node affinity, called requiredDuringSchedulingIgnoredDuringExecution and preferredDuringSchedulingIgnoredDuringExecution. You can think of them as "hard" and "soft" respectively, in the sense that the former specifies rules that must be met for a pod to be scheduled onto a node (just like nodeSelector but using a more expressive syntax), while the latter specifies preferences that the scheduler will try to enforce but will not guarantee. The "IgnoredDuringExecution" part of the names means that, similar to how nodeSelector works, if labels on a node change at runtime such that the affinity rules on a pod are no longer met, the pod will still continue to run on the node.

Thus an example of requiredDuringSchedulingIgnoredDuringExecution would be "only run the pod on nodes with Intel CPUs" and an example preferredDuringSchedulingIgnoredDuringExecution would be "try to run this set of pods in a team of nodes sharing a specific tag, but if it's not possible, then allow some to run elsewhere".

The specified function prioritizes nodes according to node affinity scheduling preferences indicated in *PreferredDuringSchedulingIgnoredDuringExecution*.

Each time a node matches a *preferredSchedulingterm*, it will get the corresponding weight added to its score. Thus, the more *preferredSchedulingTerms* the node satisfies and the more those terms weight, the higher score the node gets.

TaintToleration: This function calculates the score for every node based on the number of intorelable taints (*preferNoSchedule*) on the node.

ImageLocality: *ImageLocalityPriority* favors nodes that have already requested pod container's image. First it detects the presence of the image and then calculates the score from 0 to 10 based on the total size of those images. It takes also into account the number of nodes this image is spread, trying to prevent *node heating* phenomena, i.e., pods get assigned to the same or a few nodes due to image locality..

NodePreferAvoidPods: This is the priority function default kubernetes scheduler weighs more that anything else. Actually, if any node prefers to avoid the pod being currently scheduled, this node's score due to this function will be zero. As a result, due to the high weight of this function, the final score of this node will be also low. In other words, the reason behind this choice is making sure that any node with the annotation *scheduler.alpha.kubernetes.io/preferAvoidPods* will get to the bottom of the priority list.

BalancedResourceAllocation: This function should not be used alone, but should be used together with *LeastRequestedPriority*. It favors nodes with balanced resource usage rate. It calculates the difference between the cpu and memory fraction of capacity, and prioritizes the host based on how close the two metrics are to each other. This algorithm was partly inspired by [26].

$$\{cpu, mem, volume\}fraction_j = \frac{\{cpu, mem, volume\}requested_j}{\{cpu, mem, volume\}capacity_j}$$

$$mean_j = \frac{cpuFraction_j + memFraction_j + volumeFraction_j}{3}$$

$$variance_j = \frac{(cpuFraction_j - mean_j)^2 + (memFraction_j - mean_j)^2 + (volumeFraction_j - mean_j)^2}{3}$$

$$s_j = 1 - variance_j$$

Kubernetes native code is open-sourced. As a result, it is available for anyone to download, configure and deploy an altered version. In this thesis, a custom scheduler is used. This custom scheduler was created by changing the

native kubernetes project code, building the new scheduler executable, creating the image, and finally using that image to create the customized deployment representing the new scheduler.

After reviewing the whole Kubernetes scheduler package in the native Kubernetes git repository and understanding its functionality, we tried to observe any different functionality that a change in the code would result to. Our custom Kubernetes scheduler, includes all the default predicates, so as to serve the first level of scheduling, the feasible nodes selection. We ensured to include those predicates so as to avoid any pod placement into a non feasible node. Regarding to the priority functions, we used the *NodeAffnitity* priority giving it a large weight. Furthermore, we also kept the *LeastRequestedResources* priority so as to deal with edge cases, where our picked node, was not a feasible one. As the core decision making process is happening before actually kubernetes scheduler is used, decreasing the number of priority fuctions kubernetes scheduler uses, we also decrease the overhead of our custom system.

Chapter 4

Motivational Analysis and Observations

Both academia [27] and industry [28] have identified that contention on the low-level shared resources of a system, i.e. low-level caches and bus bandwidth, can lead to unpredictable performance variability and degradation, which highly reduces the QoS of applications [29]. 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 [30, 31]. Moreover, hardware heterogeneity can have significant impact on the performance of applications, especially for Latency-Critical (LC) applications [32, 33].

In this chapter, we analyze and verify the impact of interference on different shared resources of the system, on the performance of applications. Firstly, we describe our experimental setup and, then, we demonstrate how resource contention affects the performance and low-level metrics on a variety of cloud workloads. Also we showcase the impact of heterogeneity in workload's performance. Furthermore, we reveal Kubernetes scheduler shortcoming regarding to the pod placement in available nodes.

4.1 Experimental Infrastructure

4.1.1 System setup

For the rest of the thesis, we consider two multi-processor systems as shown in tables 4.4 and 4.5, henceforth referenced as H1 and H2. To simulate a cloud environment, all the referenced workloads running on the cluster have been containerized, utilizing the Docker platform [34]. In addition, we have deployed 5 virtual machines (VMs) on top of the physical machines with various configuration serving as the nodes of our cluster, where each VM's cores range from 4 up to 16 and RAM size from 8192(MB) up to 65536(MB) and, we use

KVM as our hypervisor. Each VM's characteristics are described in table 4.1.

Virtual Machines				
VM-Name	**Server**	**Socket ID**	**Cores**	**RAM (GB)**
kube-master	H2	0	2	8
kube-01	H2	1	4	8
kube-02	H2	0	8	16
kube-03	H2	0	16	16
kube-04	H2	1	32	32
kube-05	H1	0	4	8
kube-06	H1	1	8	16
kube-07	H1	0	16	32
kube-08	H1	1	16	64

Table 4.1: Virtual Machines Characteristics

The combination of VMs with containers is currently the common way of deploying cloud clusters at scale, since it establishes the perfect catalyst for reliability and robustness [35]. The virtual cores of each VM have been mapped on physical cores of the servers using the CPU pinning options of the `libvirt` library, to eliminate context-switching and also be able to monitor VM-specific metrics. On top of the VMs, we have deployed Kubernetes [11] as our container orchestrator, one of the most popular and most used platforms nowadays. The system as a whole is illustrated in figure 4.1. We used a single-master node cluster with the VM serving as master deployed in a separate physical machine, without affecting the testing results.

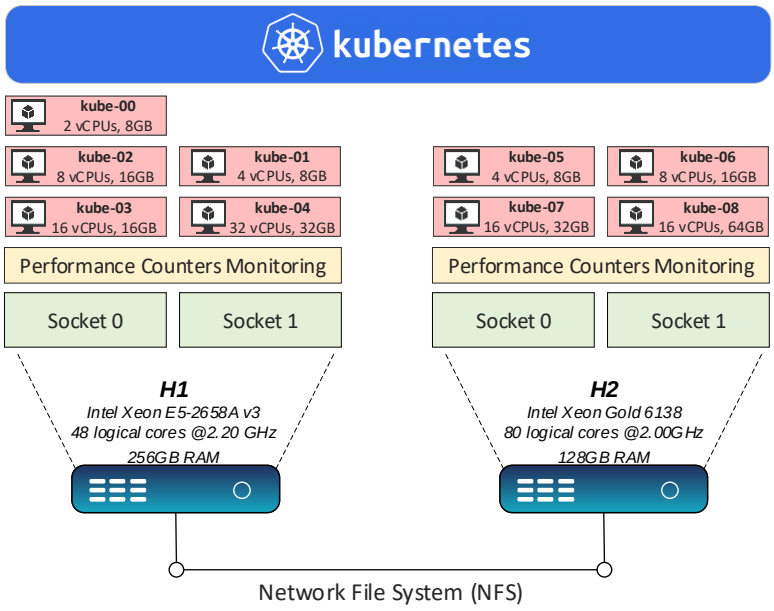


Figure 4.1: Stress Level and duration

4.1.2 Monitoring and Communication

As a first step, we need to get insight about the real system metrics. Those metrics that describe the underlying infrastructure, residing on the host machine. The previous layer of the one where Kubernetes is set on. For this purpose we use the Performance Counter Minitor(PCM). PCM is a tool developed by Intel. It is used as an agent, extracting a big variety of metrics from the system it is running on. The Intel® Performance Counter Monitor [36] provides sample C++ routines and utilities to estimate the internal resource utilization of the latest Intel® Xeon® and Core™ processors.

The CPU utilization does not tell you the utilization of the CPU. CPU utilization number obtained from operating system (OS) is a metric that has been used for many purposes like product sizing, compute capacity planning, job scheduling, and so on. The current implementation of this metric (the number that the UNIX* "top" utility and the Windows* task manager report) shows the portion of time slots that the CPU scheduler in the OS could assign to execution of running programs or the OS itself; the rest of the time is idle. For compute-bound workloads, the CPU utilization metric calculated this way predicted the remaining CPU capacity very well for architectures of 80ies that had much more uniform and predictable performance compared to modern systems. The advances in computer architecture made this algorithm an unreliable metric because of introduction of multi core and multi CPU systems, multi-level caches, non-uniform memory, simultaneous multithreading (SMT), pipelining, out-of-order execution, etc.

Using PCM we were able to extract system, socket and core metrics. Most of the metrics provided useful information about the state of the system. The available metrics are the following:

- □ Instructions per Cycle
- □ L3 Misses
- □ C-States (C0,C1,C6)
- □ L2 Misses
- □ L3 Occupancy
- □ Reads/Writes

Instructions Per Cycle (IPC): Regarding to the core metric, IPC describes the instructions required to execute a piece of code divided by the number of hardware cycles done at this time. For the socket and the system, the IPC is calculated by the following equation.

$$IPC_j = \frac{1}{o_j} \times \sum_{c=0}^{o_j - 1} (IPC_c \times C_0 State_c)$$

L2 & L3 Misses: These misses are the number of misses in the L2 and L3 cache respectively. Regarding to the core, they describe the misses occurred

in a predefined time interval. For the system and the sockets, misses are the aggregation of all the misses occurred in the cores belonging to them.

C-States In order to save energy when the CPU is idle, the CPU can be commanded to enter a low-power mode. Each core has three idle states, C0, C1 and C6. They are numbered starting at C0, which is the normal CPU operating mode, i.e. the CPU is 100% turned on. The higher the C number is, the deeper is the CPU sleep mode, i.e. more circuits and signals are turned off and the more time the CPU will take to go back to C0 mode, i.e. to wake-up. C1 state (Halt) stops CPU main internal clocks via software; bus interface unit and APIC are kept running at full speed. Finally C6 state (deep power down) reduces the CPU internal voltage to any value, including 0 Volts.

Read & Writes These metrics describe the number of reads and writes from and to the memory. They are provided only at socket/system level and they are extracted on a set time interval.

After the appropriate metrics extraction, we communicate them using a Network File System, which enables file sharing between server(s) and Kubernetes master node, where the scheduling procedure takes place. The NFS Server is on Kubernetes Master Node (kube-00). The two NFS clients, are set in the two servers, where the metrics extraction is taking place.

4.2 Description of Cloud workloads and Interference micro-benchmarks

Modern data-center server machines accommodate a 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. scikit-learn [37] and SPEC2006 [38] (as BE) and Cloudsuite [39] (as LC) suites.

4.2.1 iBench

In order to add constant artificial pressure on our tested machines, we utilized the iBench suite [40]. iBench provides contentious micro-benchmarks which can simulate various stress of resources in different intensities and for different shared resources, ranging from core up to memory levels, able to press resources that span the CPU, cache hierarchy, memory, storage and networking subsystems. We utilize those benchmarks as a mean to stress specific shared resources of the system and observe the impact of interference on the perfor-

mance of our running applications. More specifically, in this thesis we use the following micro-benchmarks:

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

In our tests, we used the `Deployment` object in Kubernetes, so as to deploy those benchmarks.

4.2.2 Scikit-Learn

Scikit-learn [41] is a a free software machine learning library for the Python programming language. It includes a wide range of state-of-the-art machine learning algorithms. Using a general purpose language, this package aims to close the gap between machine-learning and non-specialists, enabling them to learn and use it in order to provide solutions to their problems. Emphasis is put on ease of use, performance, documentation, and API consistency. It has minimal dependencies, it is open-sourced, encouraging its use in both academic and commercial settings. Workloads brief description is provided in table 4.2. More specifically:

- **Lasso:** Linear Model trained with L1 prior as regularizer (aka the Lasso).
- **Linear Discriminant Analysis:** A classifier with a linear decision boundary, generated by fitting class conditional densities to the data and using Bayes' rule. The model fits a Gaussian density to each class, assuming that all classes share the same covariance matrix.
- **Linear Regression:** Ordinary least squares Linear Regression.
- **AdaBoost Classifier:** An AdaBoost [42] classifier is a meta-estimator that begins by fitting a classifier on the original dataset and then fits additional copies of the classifier on the same dataset but where the weights of incorrectly classified instances are adjusted such that subsequent classifiers focus more on difficult cases.
- **Random Forest Regressor:** A random forest is a meta estimator that fits a number of classifying decision trees on various sub-samples of the dataset and uses averaging to improve the predictive accuracy and control over-fitting.
- **Random Forest Classifier:** A random forest is a meta estimator that fits a number of decision tree classifiers on various sub-samples of the dataset

and uses averaging to improve the predictive accuracy and control over-fitting.

The datasets used in the training phase of these workloads are comprised of 40,000 instances, with 784 features per instance. For the purposes of our thesis, we dockerized those applications. After that, they were used from within the Kubernetes cluster by creating the appropriate `yaml` files. A sample `yaml` file for a *scikit* application deployment is the following:

```
apiVersion: v1
kind: Pod
metadata:
  name: scikit-pod-lasso
spec:
  restartPolicy: OnFailure
  containers:
  - name: scikit-container
    image: registry.hub.docker.com/iwita/scikit:no_entry
    env:
    - name: CLF
      value: "Lasso"
    command: ["/bin/bash","-c" ]
    args:
      - time /workloads/fit_${CLF}.py "/784x40000.data" "/784x40000.labels" "1"
    ports:
    - containerPort: 80
    imagePullPolicy: Always
  imagePullSecrets:
  - name: regsecret
```

4.2.3 Spec CPU® 2006

Spec 2006® [38] is another suite, which includes both integer and floating point benchmarks. SPEC designed this suite to provide a comparative measure of compute-intensive performance using workloads developed from real user applications, as well as everyday operations deployed in cloud environments. The SPEC CPU® 2006 benchmark is able to measure both the time of completition of a single task a in a machine and the the amount of tasks a machine can accomplish in a certain amount of time, called throughput. For the purposed of this thesis we used the following benchmarks:

- **473.astar:** This benchmark uses language C++ and its type is integer. Astar is derived from a portable 2D path-finding library that is used in game's AI. This library implements three different path-finding algorithms: First is the well known A* algorithm for maps with passable and non-passable terrain types. Second is a modification of the A* path finding algorithm for maps with different terrain types and different move speed. Third is an implementation of A* algorithm for graphs. This is formed by map regions with neighborhood relationship. The input file is a map in binary format. The program also accepts typical map region size which

is used in regionbased path finding algorithm and density for randomly created forest-style test maps. The program also reads the number of ways to simulate. The program outputs the number of existing ways and the total way length to validate correctness.

- **437.leslie3d:** It is a floating point benchmark and it uses Fortan 90. leslie3d is derived from LESlie3d (LargeEddy Simulations with Linear-Eddy Model in 3D), a researchlevel Computational Fluid Dynamics (CFD) code used to investigate a wide array of turbulence phenomena such as mixing, combustion, and acoustics. LESlie3d uses a strongly-conservative, finite-volume algorithm with the MacCormack Predictor-Corrector time integration scheme. The accuracy is fourth-order spatially and second-order temporally. For CPU2006, the program solves a test problem using the temporal mixing layer. This type of flow occurs in the mixing regions of all combustors that employ fuel injection (which is nearly all combustors). The benchmark version, 437.leslie3d, performs limited file I/O using a theoretically exact problem. Input parameters include the grid size, flow parameters and boundary conditions. The output includes analysis information that tracks the momentum thickness through time.

- **436.cactusADM:** This is a floating point benchmark which is coded in Fortran 90 and ANSI C. CactusADM is a combination of Cactus, an open source problem solving environment, and BenchADM, a computational kernel representative of many applications in numerical relativity (ADM stands for ADM formalism developed by Arnowitt, Deser and Misner). CactusADM solves the Einstein evolution equations, which describe how spacetime curves as response to its matter content, and are a set of ten coupled nonlinear partial differential equations, in their standard ADM 3+1 formulation. A staggered-leapfrog numerical method is used to carry out the update. The input file defines the grid size, as well as the number of iterations which the code will run. The outputs are the iteration, time, and gxx and gyz components of the metric which are coordinate-dependent descriptions of the space time.

- **482.sphinx3:** Sphinx-3 is a widely known speech recognition system from Carnegie Mellon University (CMU). It uses C language and it is a floating point benchmark. CMU supplies a program known as livepretend, which decodes utterances in batch mode, but otherwise operates as if it were decoding a live human. The benchmark focuses on the CPU-intensive portions of the task, thus it reads all the inputs during initialization and then processes them repeatedly with different settings for the "beams" (the probabilities that are used to prune the set of active hypotheses at each

recognition step). The AN4 Database from CMU is used as input. The raw audio format files are used in either big endian or little endian form (depending on the current machine). Correct recognition is determined by examination of which utterances were recognized , as well as a trace of language and acoustic scores.

The deployment of those benchmarks in our Kubernetes cluster is similar to the previous ones:

```
apiVersion: v1
kind: Pod
metadata:
  name: spec2006-astar
spec:
  restartPolicy: OnFailure
  containers:
  - name: spec2006-astar-container
    image: registry.hub.docker.com/iwita/spec2006
    env:
    - name: BENCHMARK
      value: "473.astar"
    ports:
    - containerPort: 80
    imagePullPolicy: Always
  imagePullSecrets:
  - name: regsecret
```

4.2.4 Cloudsuite

Finally, the `Cloudsuite` [39] benchmarks are based on real-world online services hosted in modern data-centers. It consists of eight applications that have been selected based on their popularity in today's datacenters. The benchmarks are based on real-world software stacks and represent real-world setups. Cloud computing is emerging as a dominant computing platform for providing scalable online services to a global client base. Today's popular online services (e.g., web search, social networking, and business analytics) are characterized by massive working sets, high degrees of parallelism, and real-time constraints. These characteristics set cloud services apart from desktop (SPEC), parallel (PARSEC), and traditional commercial server applications (TPC). Those benchmarks stimulate research into the field of cloud and data-centric computing.

- **In-Memory Analytics** utilizes Apache Spark [43] and runs a collaborating filtering algorithm on a movie ratings dataset. This dataset is consisted of 21,000,000 ratings applied to 30,000 movies, by 230,000 users. Its size is 144MB.
- **Data-Serving** relies on the Yahoo! Cloud Serving Benchmark [44] and the Cassandra data store [45]. This framework comes with appropriate interfaces to populate and stress many popular data serving systems. We also increased the amount of operations to 300,000.

Table 4.2: Summary of workloads(BE=best effort and LC=latency critical workloads)

Suite	Benchmark	Abbreviations	Type	Input	Description
scikit-learn [41]	Lasso	lasso	BE	784 x 4000.data 784 x 4000.labels	Linear Model trained with L1 prior as regularizer (aka the Lasso)
	Linear Discrinimant Analysis	lda			Classifier with a linear decision boundary, generated by fitting class conditional densities to the data and using Bayes' rule.
	Linear Regression	linregr			Ordinary least squares Linear Regression
	AdaBoost Classifier	ada			This class implements the algorithm known as AdaBoost-SAMME [51]
	Random Forest Classifier	rfc			
	Random Forest Regressor	rfr			
Spec 2006 [38]	473.astar	astar	BE	a map in binary format	Pathfinding library for 2D maps, including the well known A* algorithm.
	437.leslie3d	leslie		grid size, flow parameters and boundary conditions	Computational Fluid Dynamics (CFD) using Large-Eddy Simulations with Linear-Eddy Model in 3D. Uses the MacCormack Predictor-Corrector time integration scheme.
	436.cactusADM	cactus		grid:120x120x120, 1000 iterations	Solves the Einstein evolution equations using a staggered-leapfrog numerical method
	482.sphinx3	sphinx		The AN4 Database from CMU [52] is used	A widely-known speech recognition system from Carnegie Mellon University
Cloudsuite [39]	in-memory analytics	in-mem	LC	21k ratings, 30k movies by 230k users (size 144MB)	a collaborative filtering algorithm in-memory on a dataset of user-movie ratings
	data-serving	data-serving		OperationCount=30000	relies on the Yahoo! Cloud Serving Benchmark (YCSB)
	web-serving	web-serving		Load Scale:100	social networking engine

- **Web-Serving** is consisted of 3 servers, an NGINX [46] web-server, a Memcached [47] caching server and a MySQL [48] database server, simulating modern services hosted in the cloud. More specifically, this benchmark includes the social network engine Elgg [49] and a client implemented using the Faban [50] workload generator. In the Load-Scale input parameter in the client container creation, we used the value of 100 instead of the 7 which is used by default.

In order to deploy those benchmarks in our Kubernetes cluster we needed to create the appropriate yaml files. However, as the presence of containers does not make Kubernetes and Docker identical to each other in any way, regarding to the `cloudsuite` we modified the Docker container logic into understandable by Kubernetes objects.

4.3 Kubernetes scheduler Inefficiency

Taking into account all the information provided before, Kubernetes scheduler is highly dependent on Cluster Resources. As it is illustrated in figure 4.2, scheduler follows a reservation-centric approach, trying to satisfy incoming applications requirements in high level resources. In addition, these resources deviate from the real ones. This is happening because Kubernetes is considering as allocated resources the ones the user has already requested. The resources

request can be set in `yaml` config file as it is presented in section 3.5.

Strong declared resources dependability is something that can downgrade the whole infrastructure's performance. Kubernetes API which is keeping all the necessary information, is only aware of the requested resources the user asked for, in the pod's or deployment's creation. However the real resources the application uses may differ. If they are less from the asked ones, this prevents those extra resources to be accessed by another application leading to under-utilization. In case the real resource usage is more than the declared one there may be several results depending on the limits declaration. If there is no limit declaration, Kubernetes API, is only aware of the requested resources. The extra resource usage is not reported anywhere, and it will scale greedily according to applications demands. This may lead to slowdown in the Node's already running applications.

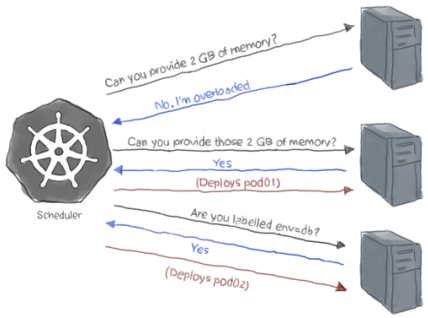


Figure 4.2: Pod Scheduling

On the other hand, another problem we observed is the kind of resources according to which Kubernetes scheduler makes scheduling decisions. Node prioritization depends on Memory (RAM) usage CPU utilization. However, we observed that those resources are not representative for the condition of the node, in many kinds of today's applications. The perfect candidate to host an incoming application is not the node with the minimum CPU and/or RAM usage. In case two or more nodes have more than enough resources to host the new application why should the one with the most space get selected? What if the physical servers in which this node is set as a Virtual Machine suffers from interference effects? What if the infrastructure suffers from memory interference caused by workload scheduled by Kubernetes Scheduler? Those questions are tested in the following sections.

Moreover, Kubernetes platform as it was referred before, has access to Virtual Machine(VM) resources such as Disk, Memory and CPU Usage. In fact those 3 types of resources are all virtual. They are part of a bigger pie of shared resources pinned from the host machine to the VMs. As a result, the observability Kubernetes has is limited. The container orchestrator is completely unaware of any co-scheduled tasks in the host machine, that may affect VMs'

performance. For example in a typical architecture, each core belonging to the same socket, share the same L3 cache. L3 cache is a limited and really critical resource for a system's performance. Processor's data missing from Level 1 (L1) cache are fetched from Level 2 (L2) cache. Furthermore if those blocks are missing from L2, they are fetched from L3. Those operations are taking place in the cache exclusively. However, if the data are also missing from the L3 cache, they are fetched from Memory (RAM). The latter operation has a great cost and also a great impact on nowadays computational performance. Delays due to LLC misses appear to be the bottleneck for any trial of delay minimization. As memory capacity, CPUs frequency, and multicore systems are being rapidly increasing over the years, the data I/O bandwidth between different memory levels is unable to follow those rapid changes. As a result different algorithms and techniques are being researched and developed for various computer architectures aiming to take advantage of any feasible cache usage improvement for delay minimization.

Returning to Kubernetes (k8s), our problem in this thesis, is the fact that Kubernetes worker nodes (kube-nodes) which are VMs created from 2 different servers share resources for which Kubernetes API is completely unaware of. As a result, k8s scheduler does not have in our opinion enough information before its scheduling decision making. Kube-scheduler is connected with kube-apiserver, where node information are exposed. We claim that those data are not representative enough about nodes' condition. As a matter of fact, we conducted some tests to prove our claim.

As it was also referred before, cache interference has a great impact on the performance of workload. Applications that are interacting more with cache are more sensitive to such cache usage intensity. As a result, it is very critical applications with intense cache usage not to be placed in stressed shared resources.

We tried to examine, what would Kubernetes scheduling decision be, when an application is ready to be scheduled over pre-existing workloads. We used two different Virtual Machines (VMs) where, each one's cores are pinned in a different socket of the server, a fact that Kubernetes is unaware of at the moment.

- kube-01 : 4 CPUs, 8GB of RAM
- kube-02 : 8 CPUs, 16GB of RAM

The decision Kubernetes scheduler has to make is to select the best matching node (VM) for the incoming application.

Firstly, we placed into our system, 9 applications from iBench. The 3 of them were cpu-ibench intensive workloads and the other 6 were LLC intensive

(l3-ibench). We put the 3 CPU intensive applications in kube-01 and the rest in kube-02. Those 2 nodes' cores belong to different sockets, with the last residing in the same server.

Stress among VMs		
VM name	Kind of Stress	pods/available cores
kube-01	CPU Bound	3/4
kube-02	L3 cache intensive	6/8

For our first test, we used the Lasso workload from the scikit-learn benchmark suite, as described in section 4.2. After scheduling the iBench benchmarks in the proper nodes, we tried to schedule the scikit application. We repeated this experiment 5 times. Kubernetes scheduled the application every time in kube-02 (the one with the 6 x l3 intensive benchmarks). Then, we repeated the process above, but this time we forced the application to be scheduled in kube-01, by using a `nodeSelector`.

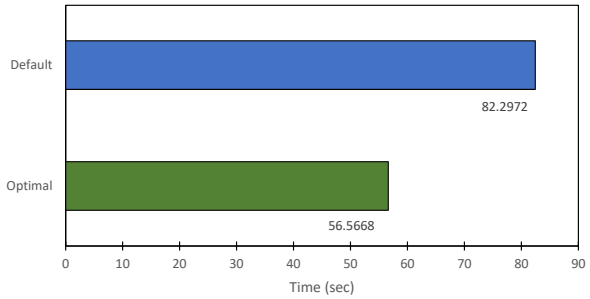


Figure 4.3: Average application completition time

As it is presented in the graph above, the delay is calculated as:

$$delay = \frac{optimal\_time}{default\_time} = 0.6873$$

The delay seems to be remarkable. As a result in this thesis, we are trying to analyze interference phenomena, performance degradation of co-located applications and to implement an interference (and heterogeneity) - aware scheduler able to be integrated with a Kubernetes cluster set on Virtual Machines over heterogeneous Servers.

4.4 Impact of interference on the performance of applications

Data-centers operators are trying to increase utilization, by accommodating multiple applications in shared resources. Those resources are prone to contention, causing destructive interference and leading to unpredictable performance. Shared resources like last level cache, memory and network bandwidth,

core utilization and others usually suffer from such interference phenomena. Imposing interference in the Last-Level cache (LLC) and memory pressure has been proven to induce high performance degradation [5]. Moreover, latest benchmarking of known cloud providers has shown that memory is reported to be the new bottleneck that destroys applications performance [28]. Contention is a state data-center operators should avoid, as a stressed resource may characterize the current system unable to serve QoS requirements. As a result, while one resource is over utilized, others may remain both under utilized and unusable. Best-practices in such cases suggest balanced resources exploitation so as to maximize performance per used server. Our first challenge is to observe applications' sensitivity to a variety of resources stress varying in nature and intensity.

As interference has an impact on the performance of applications, in this section, we profiled our target workloads, while stressing different shared resources, with various levels of interference intensity. In System/Server level the shared resources we are investigating are CPU, L2 cache, L3 cache and memory bandwidth. Our servers are consisted of 2 sockets with isolated per socket Level 3 caches. However those sockets' cores are distributed in different VMs. An interesting fact there is that by stressing one VM, also another VM is highly impacted. This is a common issue in Data Centers nowadays. Users in different VMs may observe performance degradation due to another VM, many times owned by another party.

In the following subsections, we stress different resources of our system, and derive some useful observations regarding to the performance degradation of various applications deployed. We denote the performance of application i as:

$$performance_i = \frac{1}{elapsedTime}$$

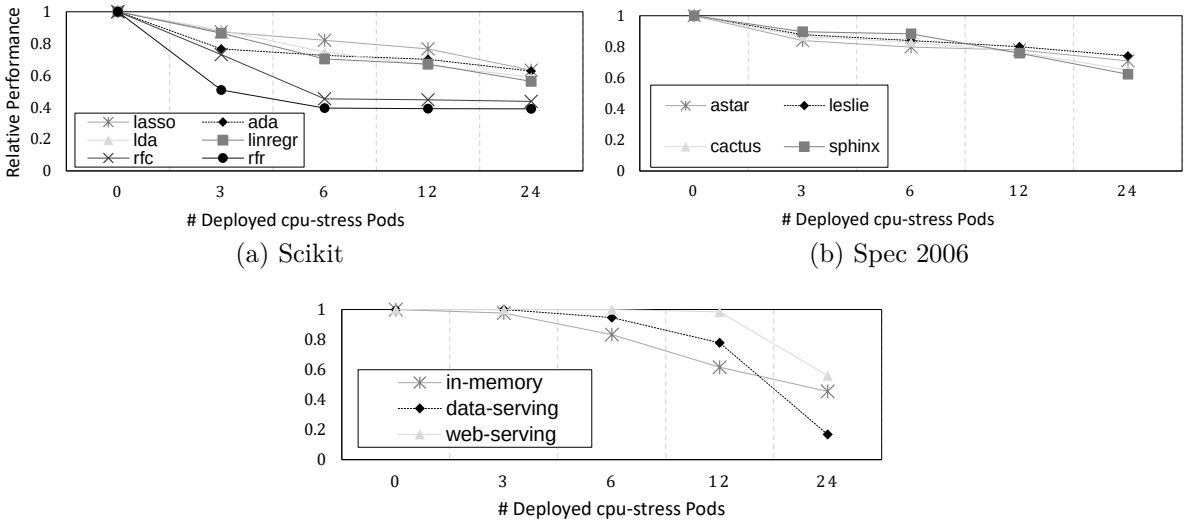
As $performance(A, S)$ we define the performance of application A under stress S. For now we measure the stress level using the amount of pods deployed. So $performance(lasso, 8 \times l3 - ibench)$ is the performance of scikit $lasso$ application, co-scheduled with 8 pods of $l3 - ibench$. Therefore $performance(A, 0)$ is the performance of the application when executed isolated. Using the latest variable, we define normalized performance.

$$normalizedPerformance(A, S) = \frac{performance(A, S)}{performance(A, 0)}$$

4.4.1 Stressing the Cores

The first resource we stressed was core utilization. Using `cpu-ibench` we measured the performance and different metrics of the system in different intensities.

We stressed one of our system sockets with 3, 6, 12 and 24 pods utilizing 12.5%, 25%, 50% and 100% of the available socket's cores belonging to the cluster respectively. As figure 4.4 depicts, the utilization of the cores of the socket has negligible impact on the performance of our target workloads, compared to L3 and memory bandwidth stress (described in next paragraphs). Specifically, for the scikit-learn benchmarks, the behavior of the workloads varies. While Linear Discriminant Analysis (`lda`), Lasso (`lasso`), AdaBoost Classifier (`ada`) and Linear Regression (`linregr`) have not significant impact on their performance, Random Forest Regressor (`rfr`) and Random Forest Classifier (`rfc`) are more affected by increased cpu utilization. On the other side, SPEC 2006 benchmarks behaviour is pretty similar. They are not greatly impacted by high core utilization. Last but not least, cloudsuite benchmarks performance is downgraded in the late test cases, where the number of deployed ibench pods had been increased. This is probably happening due to multi-threading. Cloudsuite applications use multi-threading and as a result cpu resource starvation affects more and more threads when there are not cores in idle state left.



(a) Scikit

(b) Spec 2006

(c) Cloudsuite

Figure 4.4: Impact of CPU stress on the performance of target applications.

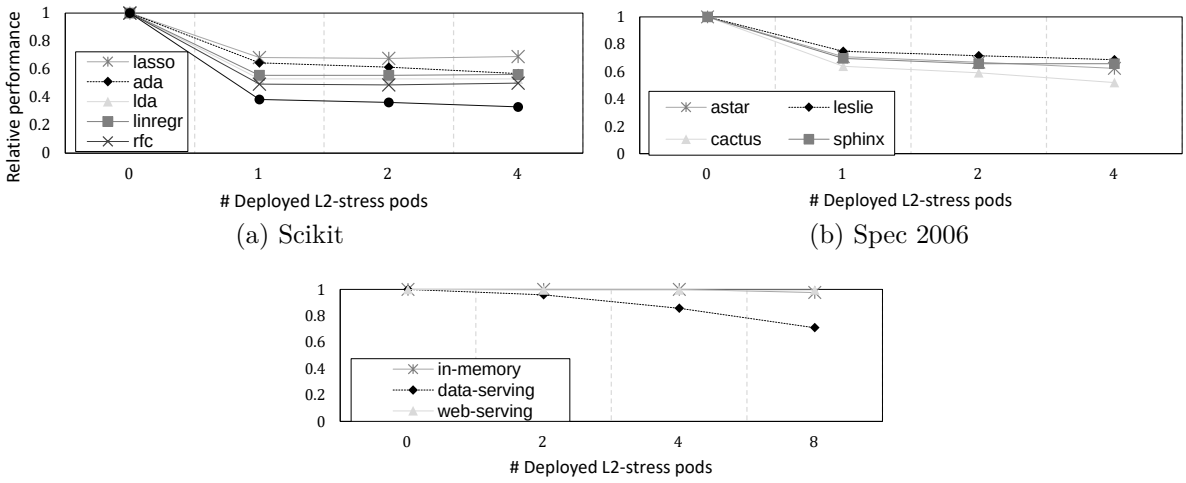
4.4.2 Stressing L2 cache

Attempting to go deeper in the cache hierarchy, we tested the L2 cache stress impact on scikit, spec 2006 and cloudsuite applications' performance. In order to stress the L2 cache we used the `L2-ibench` benchmarks, which were scheduled in the appropriate hardware thread that is sharing the same L2 cache with the thread where the tested application would be placed. As Kubernetes deployment objects do not support application scheduling in a specific core, we configured the ibench, scikit and SPEC 2006 images. Moreover, in the pod

configuration file, we added some additional commands using *taskset* without affecting container's execution. In order to test scikit and SPEC 2006 applications, we co-scheduled 1,2 and 4 `L2-ibench` pods on the one hardware thread and our tested application on the other. This way we were able to test L2 cache resource stress, by providing an idle hardware thread for the application to be executed at the same time. It is also notable there that the previously referred threads are pinned to different VMs.

Figures 4.5a and 4.5b show the impact of L2 stress on various scikit and spec 2006 applications. The performance degradation the tested applications suffer from, is remarkable still from the first `L2-ibench` pod addition. However, there is no addition degradation after the first L2-ibench pod. This behavior occurred as the L2 cache of the hardware thread where our application was placed, was already continuously clearing all the needed blocks. Beyond that, any addition iBench pod did not affect further the performance of the application, as L2 cache was already experiencing misses in every access endeavor. On the other hand, for the multi-threaded cloudsuite benchmarks, we tested a different scenario. Before, scheduling each application on a 8-cpu VM vm_1, we had placed 2,4 and 8 `L2-ibench` pods in the respective hardware threads(sharing L2 cache with vm_1's cores) which are pinned to another VM. As illustrated in figure 4.5c the impact on performance was negligible.

As a result, scheduling taking into account the L2 cache stress in individual cores on the scheduling of incoming applications seems to be promising for some benchmarks. However, this is a practice that would be better applied on a online scheduler able to migrate applications in different cores, nodes or sockets at runtime.



(a) Scikit

(b) Spec 2006

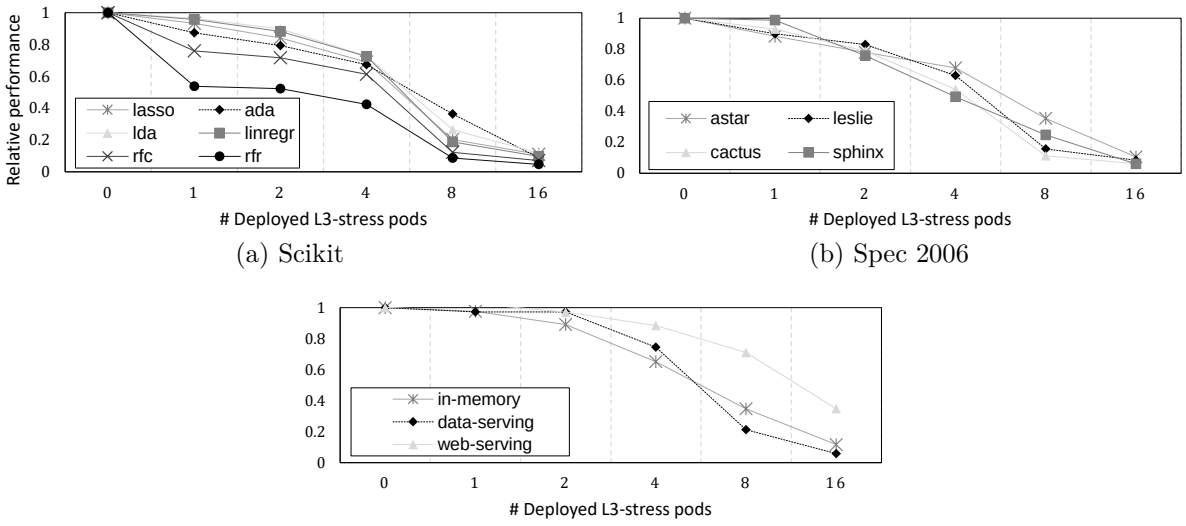
(c) Cloudsuite

Figure 4.5: Impact of L2 Cache stress on the performance of target applications.

4.4.3 Stressing L3 Cache (LLC)

Next, we containerized the `L3-ibench` and deployed it in our Kubernetes cluster selecting a VM pinned in socket1 and setting it running indefinitely. Each `L3-ibench` pod accesses the 50% of available LLC continuously. Figure 4.6, shows the normalized performance of different applications due to LLC stress in various intensity levels (1,2,4,8,16).

While `SPEC 2006` benchmarks seem to suffer from a linear degradation of their performance, `scikit` ones have the 4 pods as a boundary of devastating impact on their performance. The difference in normalized performance between 4 and 8 pods is up to 79.5% and seems to be a contention threshold, especially in `cloudsuite` benchmarks which performance has a great impact only beyond that. Different applications in the same group also vary in their LLC stress sensibility. Random Forest Regressor (`rfr`) has negative impact on his performance earlier than other benchmarks, proving to be more sensitive in this kind of pressure. In addition, `sphinx` from SPEC 2006 benchmarks although has no impact on its performance co-scheduled with 1 pod, when more pods are placed, the performance is linearly downgraded with a high incline.



(a) Scikit

(b) Spec 2006

(c) Cloudsuite

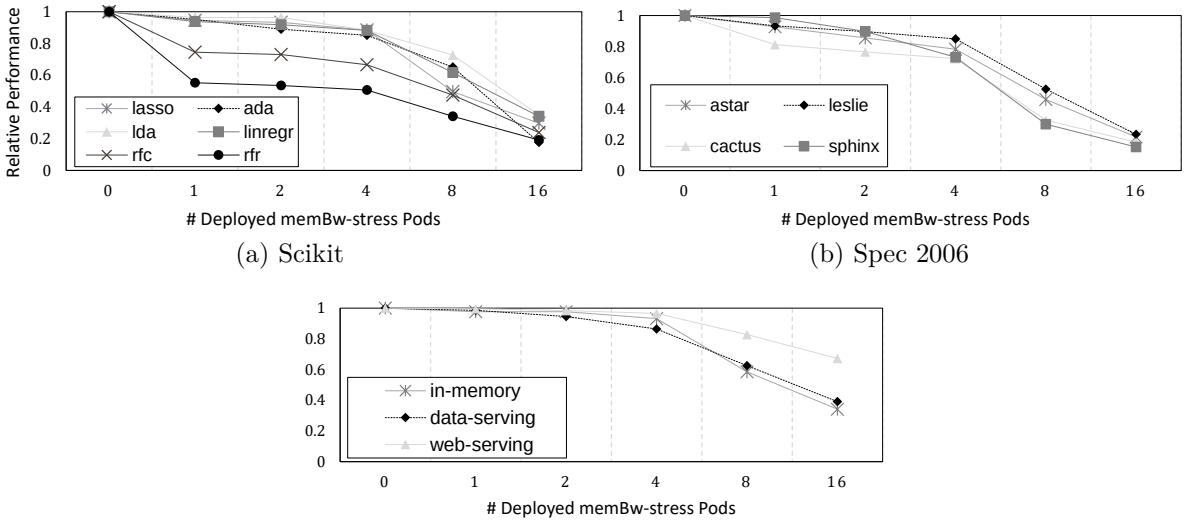
Figure 4.6: Impact of L3 Cache stress on the performance of target applications.

4.4.4 Stressing Memory Bandwidth

Next, using the `memBw-ibench` benchmark we were able to stress another shared resource, the memory bandwidth. Following the same procedure as before, we extracted stressing results, showed in figure 4.7. Memory bandwidth is the next resource after LLC hierarchically. All of the benchmarks have a much smoother gradient than the ones co-existing with LLC pressure. Cloudsuite and SPEC

2006 applications are insensitive in memory bandwidth before the threshold of 4 pods scheduling.

On the other hand, scikit benchmarks performance varies. While `rfr` suffers from 44.7% performance degradation from the 1st pod's addition, `linregr`, does not get as affected even in the 4th pod's scheduling.



(a) Scikit

(b) Spec 2006

(c) Cloudsuite

Figure 4.7: Impact of Memory Bandwidth stress on the performance of target applications.

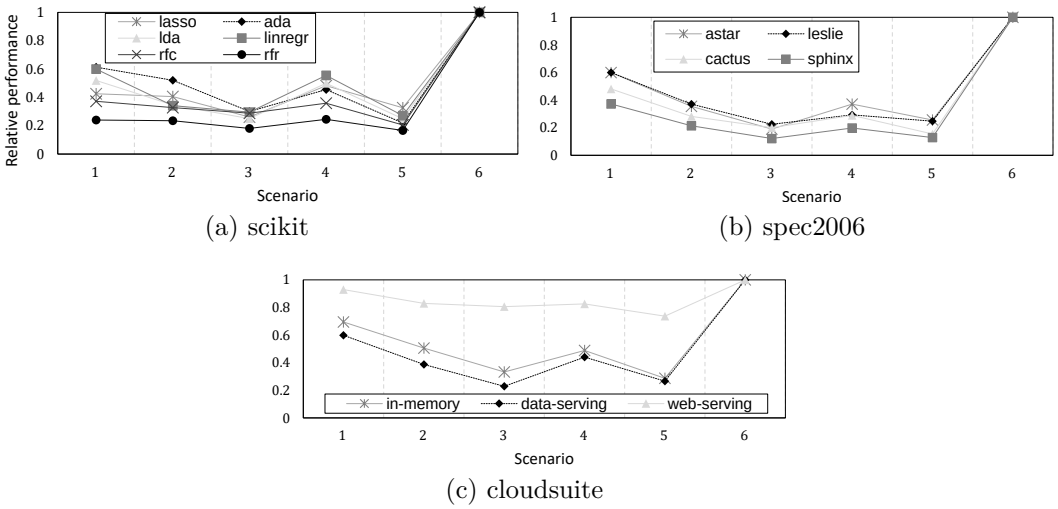
4.4.5 Mixed Stressing Scenarios

Trying to simulate real world case workload scenarios, we used different intensities of `ibench` pods for stressing cpu, L3 cache and memory bandwidth at the same time. We created 6 scenarios, which will be the pre-existing workload, before scikit and spec applications get co-scheduled. Those scenarios are described in table 4.3. The different benchmarks behavior can be observed in figure 4.8a between `ada` and `linregr`. While their normalized performance seems to be equal in most scenarios, in the scenario 2, where L3 cache stress is more than scenario 1 and less than scenario 3, `ada` performance degradation is much smoother compared to `linregr` and the other benchmarks. Contrarily, in scenario 4, `linregr` achieves greater relative performance than others do.

Roughly speaking, in all the scenarios of pressure, scikit, SPEC 2006 and cloudsuite applications experienced analogous impact on their performance. That means that performance was determined by system's condition relatively. The sorted scenarios according to applications relative performance is 6,1,4,2,3. Interestingly that sorting remains the same for all benchmarks and the need for an indicator of performance that would select a socket/system from a pool of available ones, so as to maximize any incoming application's performance is revealed.

Table 4.3: Stressing Scenarios

No. of Test	L3 (pods)	memBw (pods)	CPU (pods)
#1	2	3	8
#2	5	1	4
#3	8	1	1
#4	2	8	2
#5	10	0	6
#6	0	0	0



(a) scikit

(b) spec2006

(c) cloudsuite

Figure 4.8: Impact of mixed resources stressing scenarios on the performance of target applications.

4.4.6 Quantifying Stress Levels

While ibench adoption can offer a great perspective over co-scheduled applications performance by stressing and considering individual low-level resources, for the real bottleneck of the system detection we need to inspect and examine the low-level performance counters as a whole [14].

For this purpose we needed a more general way of measuring the stress level. Trying to find an appropriate metric to measure socket performance, we also monitored socket metrics using PCM. We extracted L3 cache misses number, C0-state percentage, reads and writes from and to the memory and IPC. Furthermore, we calculated the average of the values extracted. In the following figures 4.9a and 4.9b, the variation of different metrics during different levels of L3 cache stress is presented. In figure 4.9a, are displayed the L3 misses and L3 hits of the socket. The radical increase after the second pod addition is happening because more and more minor processes of the system are experiencing misses in every LLC access. The decrease of the L3 misses after the 4th pod addition is an observation worth to be reviewed. L3 misses

intensity beyond that point is so high that it cannot get served by the available memory bandwidth, thus processes requesting data missing from L3 cache are in a wait state, their IPC is decreased and the total amount of additional memory requests per timeframe is also decreased. Regarding to the figure 4.9b, a plateau in reads and writes from and to the memory is illustrated starting from the addition of the 4th pod. IPC is increased during the zero to one L3-iBench pod transition. While, one L3 cache-iBench pod flushes continuously half of the cache, useful blocks used by minor processes in the cluster are not written back to memory. As a result, this process addition, contributes more in the average IPC of the socket increment than the total L3 misses count.

(a) L3 Misses & Hits

(b) Reads, Writes & IPC

Figure 4.9: Impact of L3 Cache stress on low-level metrics of the socket.

A conclusion derived from those figures is the dependability of the metrics on stress level. As figures 4.6 and 4.7 previously also show, the pressure on L3 cache misses and memory bandwidth downgrades performance radically. LLC is basically the border between the cores and the main memory. Any operation taking place beyond that border generates additional delays in higher order of magnitude than the operations happening within (L1 instruction misses, L1 data misses, L2 misses). Memory reads and writes are the requests for data in behalf of L3 cache misses, and provide a low level performance counter able to depict the number of memory access.

Additionally, following the previous metaphor, memory bandwidth represents the width of the pipe connecting operations within and beyond that aforementioned border. As LLC misses are increased, the number of operations that will happen in memory side is increased as well. If the ratio of such increments of operations is higher than the one memory bus can serve, memory contention occurs. In this case, different processes are competing for memory access [53], and as the available bandwidth is not able to support all requests at the same time, neither memory reads and writes number nor L3 cache misses are valuable indicators of contention beyond this point. In this case, `IPC`, another low level hardware metric, is an indicator of further slowdown caused by delays in process execution due to memory bus competition. A custom metric

we propose that is promising regarding to the stress level calculation is:

$$Custom\ Metric\,(S) = \frac{Reads + Writes}{IPC} \tag{4.1}$$

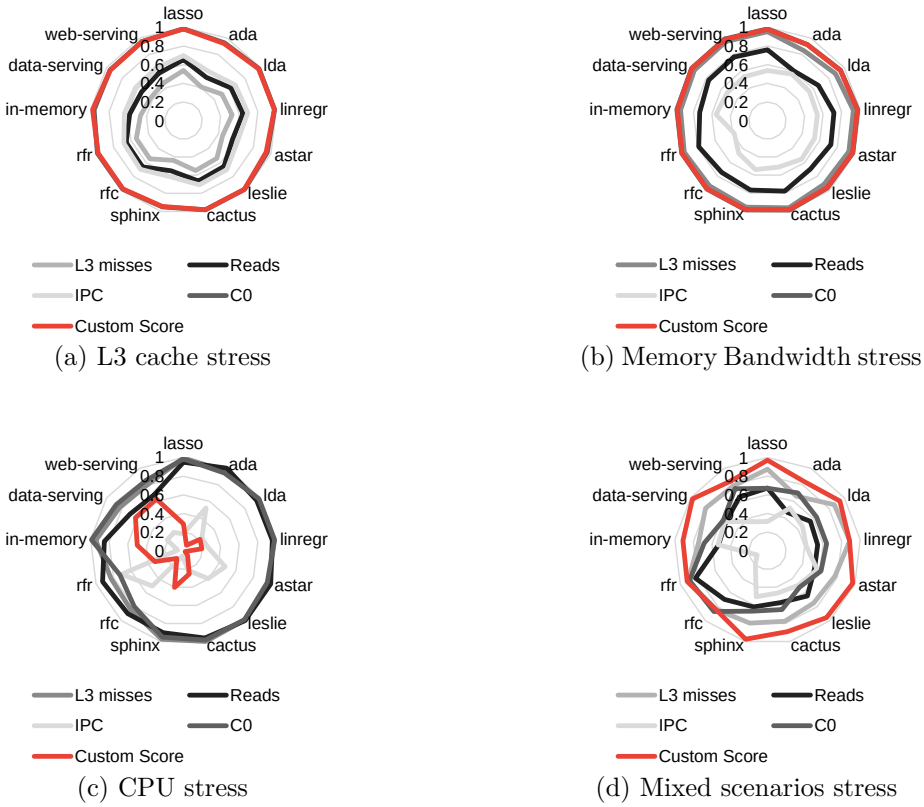
where reads, writes and IPC are the average of the measured values. This metric takes into account the reads and writes from and to the memory, since the delay caused moving data from and to the memory is decisive for the system's performance. Additionally, in cases when reads and writes reach a plateau, IPC is a valuable metric indicating the condition of the system.

System's condition estimation through low-level metrics

In order to compare the accuracy of different metrics on reflecting the condition of the system, we computed the Pearson correlation between the performance degradation of each application executed under interference and the average of the socket's low-level metrics sampled prior to the scheduling of the applications. Figure 4.10 shows the correlation between the normalized performance of the application under different levels of stress as described in the previous section and the corresponding metrics values. Our custom metric seems to be highly correlated with the performance of application in most scenarios. In `L3-ibench`, the custom score and the C0-state of the sockets are competing for the first place. Furthermore, in the `memBw-ibench` stress, L3 misses seem to be correlated with applications performance too. In those two previous scenarios, the high value of C0-state in performance correlation is disorienting. As C0 depicts the percentage of physical cores in executing state (not being idle), it is expected to get increased when the number of deployed pods is also increased. As a result, due to the fact that those pods are stressing the system, C0 seems to be highly correlated with system's contention. However, as figure 4.10d illustrates, when different pods (`cpu,L3,memBw`) according to the table 4.3 are deployed, C0 is not a reliable system state indicator anymore. Finally, in core utilization stress, our custom metric fails to indicate the isolated case and this is depicted in correlation as shown in figure 4.10c. This is happening because in the case of an empty system, the IPC metric is low enough to increase the value of our scoring function.

4.5 Impact of heterogeneity on the performance of applications

Nowadays, data centers are consisted of systems differing in hardware characteristics. As a result, clusters provided to clients are also heterogeneous, with



(a) L3 cache stress

(b) Memory Bandwidth stress

(c) CPU stress

(d) Mixed scenarios stress

Figure 4.10: Correlation between applications performance degradation and system metrics.

Virtual Machines residing in diverse systems. In such cases, when an orchestrator like Kubernetes is responsible for nodes over different underlying systems management, those different characteristics in resources and their usage must be used properly in order to maximize the performance of the system as a whole.

The design of such a heterogeneity-aware scheduler is a research subject. In the next paragraphs, we conducted the respective tests we did earlier, this time on another server with different characteristics. Those two different machines are described in table 4.4 (H1) and table 4.5 (H2)

Table 4.4: Host-1 (H1) specifications

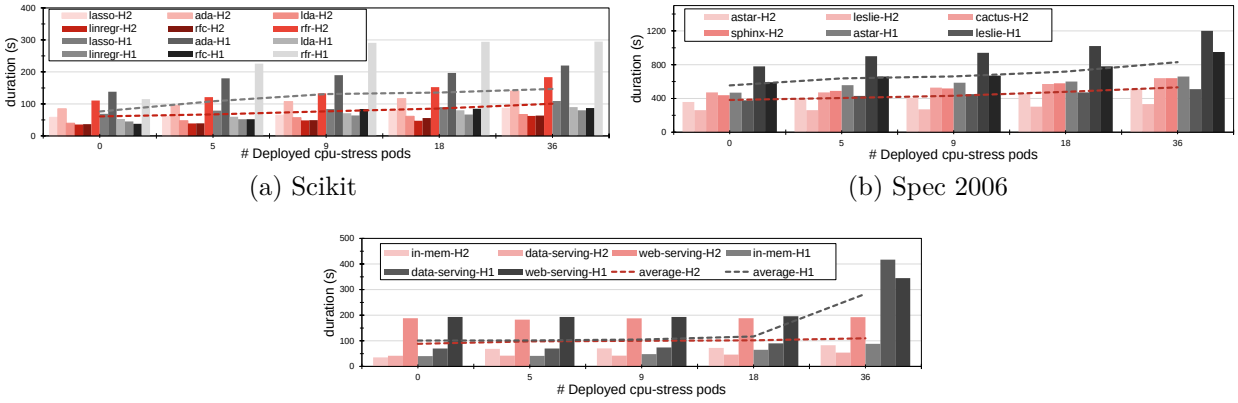
Processor Model	Intel® Xeon® E5-2658A v3
Cores per socket	12 (24 logical) @2.20GHz
Sockets	2
L1 Cache	32KB instr. & 32KB data
L2 Cache	256KB
L3 Cache	30MB, 20-way set-associative
Memory	256GB @2133MHz
Links	2 x QuickPath Interconnect
Operating System	Ubuntu 16.04, kernel v4.4

Table 4.5: Host-2 (H2) specifications

Processor Model	Intel® Xeon® Gold 6138
Cores per socket	20 (40 logical) @2.00GHz
Sockets	2
L1 Cache	32KB instr. & 32KB data
L2 Cache	1024KB
L3 Cache	28MB, 11-way set-associative
Memory	128GB @2666MHz
Links	3 x Ultra Path Interconnect
Operating System	Ubuntu 18.04, kernel v4.15

4.5.1 Stressing the Cores

As it is illustrated in figure 4.11, H2 is constantly performing better than H1. CPU stress does not seem to radically impact the performance of deployed applications. Furthermore, the deviation between the performance of the two hosts remains constant between 31% and 36% across all the different scenarios for spec 2006 benchmarks. This aforementioned behavior characterizes also the scikit-learn applications. On the other hand, in the 100% cpu utilization scenario, H2 due to its greater cores availability was able to schedule in a more spacy VM the multi-threaded cloudsuite benchmarks. While difference in performance is below 10% in the first four scenarios, in the fifth the respective deviation is 61&.



(a) Scikit

(b) Spec 2006

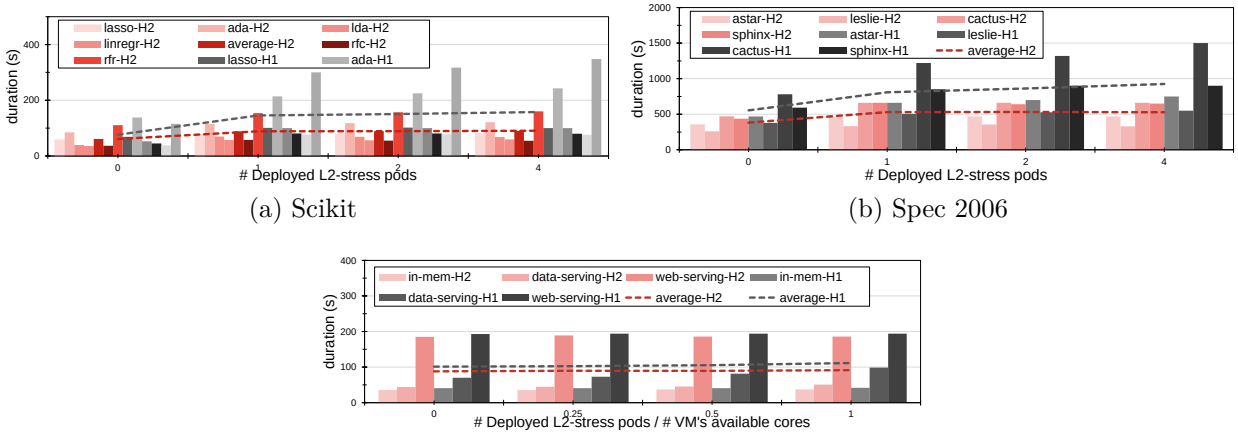
(c) Cloudsuite

Figure 4.11: Comparative performance analysis between H1 and H2, under CPU-stress

4.5.2 Stressing L2 cache

In L2 cache stress, as was also described in the single server analysis in section 4.4 only one (L2 cache-iBench) pod is enough to flush the whole L2 cache and force any other process trying to access it into a miss. Beyond that further pod addition have no impact in any benchmarks performance. As figure

4.12a depicts, H2 appears to be tolerant enough. The performance deviation is increased after one pod addition and remains constant during the latter scenarios. It is important to note at that point that H2 is consisted of larger L2 cache capacity as described in table 4.5. While scikit-learn (figure 4.12a) and spec 2006 (4.12b) benchmarks seem to be affected by heterogeneity, cloudsuite ones (figure 4.12c) are not. More specifically, scikit benchmarks performance deviation during the zero to one pod transition is altered from 20% to 40%.



(a) Scikit

(b) Spec 2006

(c) Cloudsuite

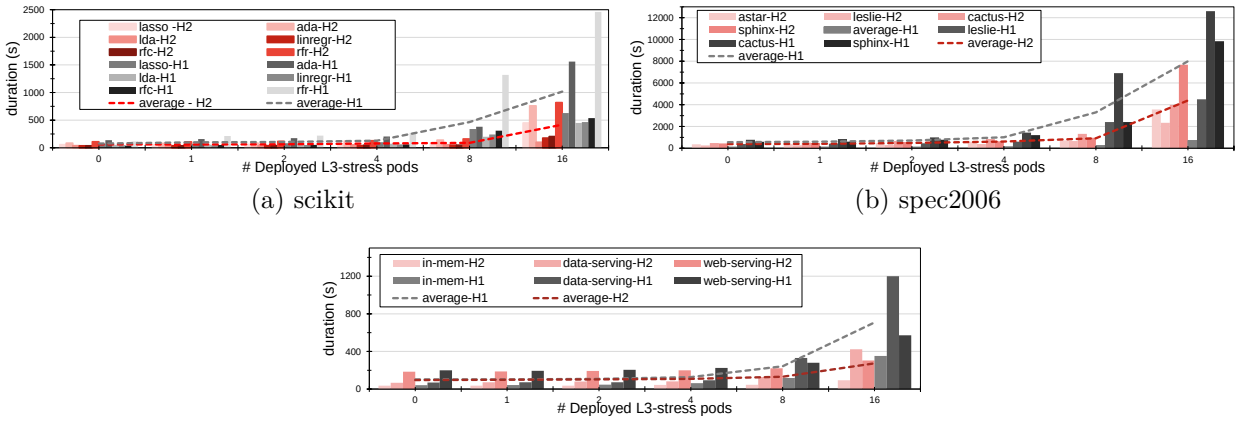
Figure 4.12: Comparative performance analysis between H1 and H2, under L2 cache-stress

4.5.3 Stressing L3 cache (LLC)

Scikit (figure 4.13a) and SPEC 2006 (figure 4.13b) workloads present similar behavior until the addition of the fourth pod. H1 and H2 performance is almost the same with the isolated execution of the corresponding benchmark. Compared with H1, H2 presents a greater tolerance in L3 cache stress. While H1 is impacted by interference in L3-cache from the fourth pod addition, H2 is affected only after the eighth pod respectively. However both Hosts performance is greatly deprecated in 16 L3 cache stress pods placement. On the other hand, cloudsuite applications (figure 4.13c), performance is being degraded with a slower ratio when compared with H1. Both hosts are only greatly impacted after the addition of the 8th pod.

4.5.4 Stressing Memory Bandwidth

As tables 4.4 and 4.5 describe, the two hosts differ in the number of links from/to the memory, as well as in the corresponding bandwidth. H2 which is consisted of three Ultra Path Interconnect links (instead of two QuickPath Interconnect links in H1) is evident that handles memory transactions in a much more efficient way as shown in figure 4.14. Comparing the two hosts, H2 is only

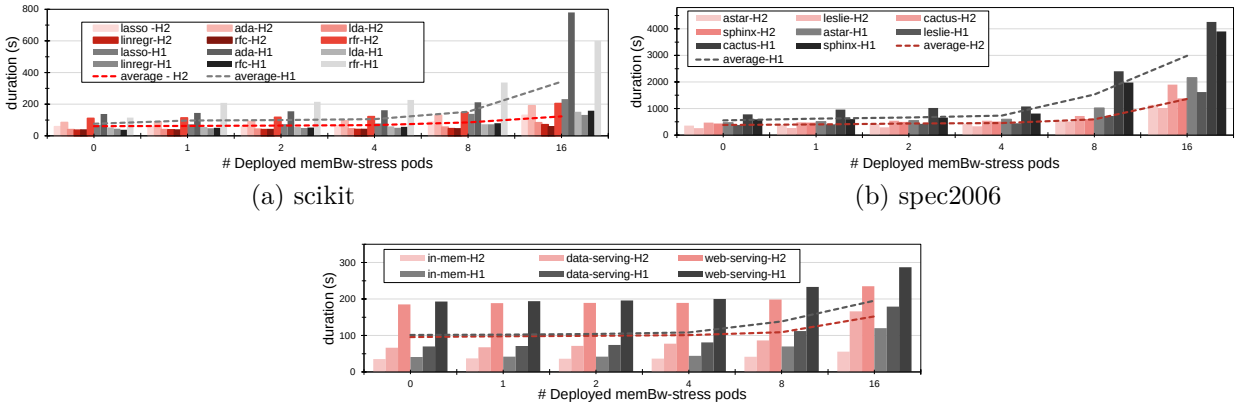


(a) scikit (b) spec2006

(c) cloudsuite

Figure 4.13: Comparative performance analysis between H1 and H2, under L3 cache-stress.

greatly impacted by memory bandwidth pressure in the more intensive scenario (16 pods). Thus, the variation in H1 and H2 performance is kept constant until the 8th pod addition, when H1 experiences contention and the performance of the deployed application is significantly degraded. While H2 performs better even in low pressure scenarios in scikit and SPEC 2006, the two servers operate in a similar manner in the cloudsuite benchmarks. Subfigures 4.14a, 4.14b and 4.14c illustrate the performance of the corresponding benchmark suite, as well as the average value for each suite over the different stressing scenarios.



(a) scikit (b) spec2006

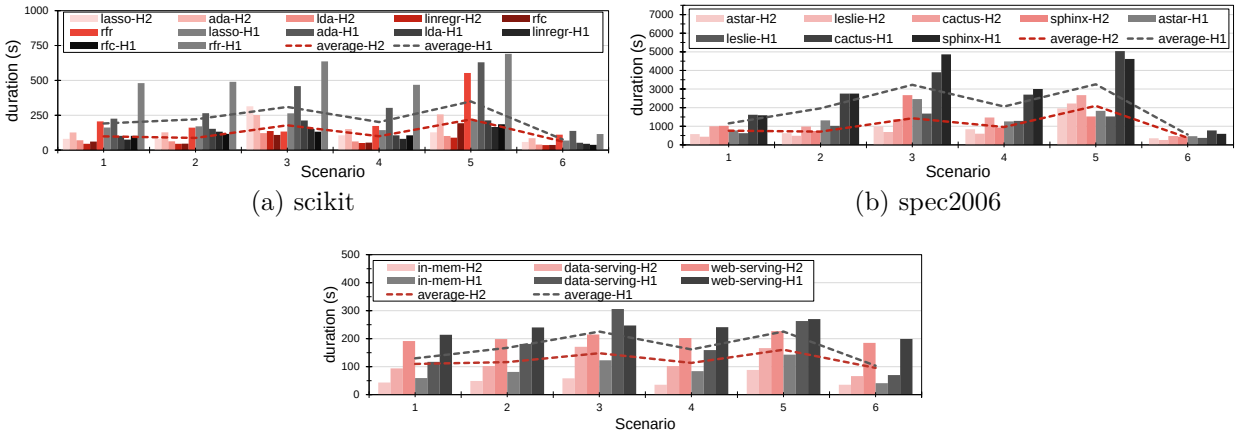
(c) cloudsuite

Figure 4.14: Comparative performance analysis between H1 and H2, under Memory Bandwidth-stress

4.5.5 Mixed Stressing Scenarios

Finally, we deployed the different stressing scenarios described in 4.3. As the dashed lines depict in figure 4.15, the average elapsed time across all the different scenarios behavior is similar for the three benchmark suites. The performance deviation between H1 and H2 is observed in scenario 3. We would expected

the scenarios with the most delay to have the greatest difference between the two hosts performance. However, the average performance of H1 in scenario 3 is worse that scenario 5. On the other hand, H2 achieves better performance in scenario 5 than scenario 3. One explanation for this behavior is the greater memory bandwidth of H2 (table 4.5) and the larger L3 cache of H1 (4.4) at the same time. While in scenario 5 there are 10 x L3cache and 0 x Memory Bandwidth stress pods deployed, in scenario 3 the pods are 8 and 1 respectively.



(a) scikit (b) spec2006

(c) cloudsuite

Figure 4.15: Comparative performance analysis between H1 and H2, under different stressing scenarios

4.6 Stress Duration and Stress Level Pareto

Scheduling is a process, where the optimal candidate should be selected for executing the application in the beginning of the 'ready to be scheduled' queue. Regarding to the makespan scheduling problem, where we consider having m identical machines, and n jobs, with processing time $p1, p2, p3, ...pn$, the objective is to minimize the makespan by scheduling each job in the appropriate machine. One algorithm for accomplishing this goal is the Longest Processing Time algorithm (LPT). However, such an algorithm cannot be applied in an on-line scheduler, as it needs the duration of the applications that are waiting to be scheduled in the first place. Another problem arising for real-world scheduling is the execution time slowdown of an application due to interference happening in the shared resources. A question occurred during the development of the current thesis was the competition between 2 candidate machines with the following corresponding states. The first machine is under stress $S1$ and the application running has an approximate duration time $t1$. The second node has stress $S2 < S1$ and approximate duration time $t2 > t1$. Trying to simplify the problem definition, we assume each node has one application running.

Trying to figure out this question we conducted the following test. We co-scheduled SPEC 2006 applications with L3 cache stress benchmarks in 3 scenarios. We deployed 4, 8 and 16 L3 cache-ibench pods for a duration equal to $t^a_{isolated}$, $\frac{t^a_{isolated}}{2}$ and $\frac{t^a_{isolated}}{4}$ respectively, and on top of them we placed application a.

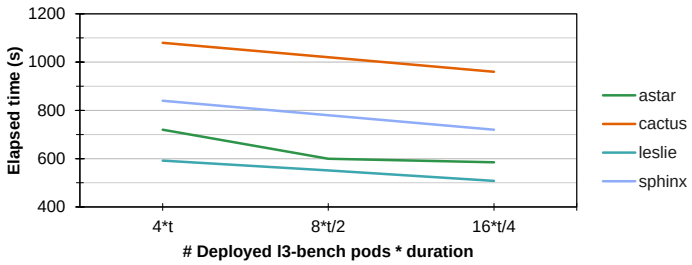


Figure 4.16: Stress Level and duration

In figure 4.16, the results are displayed. As we can see, the product of stress and its duration does not indicate the same degradation in the performance of applications. We denote as c_1 the case with smaller stress and longer duration and as c_2 the case with the greater stress and the shorter duration. We can see that c_1 leads to greater performance degradation of the evaluated application compared to c_2. As a result cases with the same product ($stress \times duration$) differ in performance degradation, with the duration parameter being the one with the most impact.

In the following sections we discuss and compare experimentally different naive approaches that could offer a better modeling for the score occurred by the product of stress level and duration.

Chapter 5

Interference-aware Kubernetes Scheduler

We target conventional data center environments, where applications are arriving on the cluster and an orchestrator is responsible for scheduling them on the available pool of VMs lying on top of the server systems, as shown in figure 5.1.

5.1 Mathematical Modeling & Problem Definition

Our problem in practice is a Partition problem or Number Partitioning generalization called Multiprocessor Scheduling Problem. Number Partitioning is the task of deciding whether a given multiset S of positive integers can be partitioned into two subsets $S1$ and $S2$ such that the sum of the numbers in $S1$ equals the sum of the numbers in $S2$. Multiprocessor scheduling algorithms are static or dynamic. A scheduling algorithm is static if the scheduling decisions as to what computational tasks will be allocated to what processors are made before running the program. An algorithm is dynamic if the decision is made at run time. For static scheduling algorithms, a typical approach is to rank the tasks according to their precedence relationships and use a list scheduling technique to schedule them onto the processors. We have $n = number\ of\ sockets$ sets, as many as the unique L3 caches in our infrastructure.

Another reduction is to a Bin Packing problem variation called minTotal Dynamic Bin Packing (DBP) [54]. In the bin packing problem, items of different volumes must be packed into a finite number of bins or containers each of volume V in a way that minimizes the number of bins used. Dynamic Bin Packing (DBP) is a variant of classical bin packing, which assumes that items may arrive and depart at arbitrary times. The minTotal DBP is a new version of the DBP problem, and it targets at minimizing the total cost of the bins used over time. In our case we only know the items' departure time when executed isolated. The arrival time and the item size are only known when the item arrives. The items are not allowed to move from one bin to another once they

have been assigned upon arrivals.

Decription: However, the problem we are trying to solve is an online job assignment in a finite amount of bins. Our objective is the minimization of scheduled application a performance degradation d_a, $\forall a \in B$, where B is the set of the total jobs to get assigned.

Target HW model: Each server is uniquely identified by an identifier $i \in \mathbb{N}^{\leq n}$, where n is the total number of servers available on the cluster. We denote the $j^{th}, j \in \mathbb{N}^{\leq m_i}$ socket of server i as s_j^i, where m_i is the total number of sockets of server i. Every socket $s_j^i \forall i, j$ is characterized by its attributes $\langle C6, IPC, Reads, Writes, \#Links, LinksBandwidth \rangle$ and is consisted of o_j number of cores. Furthermore, each node (VM) in the cluster is consisted of vcpus pinned in the same socket and is denoted as $n_l^{j,i}$. Every node is characterized by the total number of cores $p_l \leq o_j$ and their average C6 $\langle C6 \rangle$. In addition, each core of socket s_j^i and node's $n_l^{j,i}$ is denoted as $c_k^{j,i,l}, k \in \mathbb{N}^{\leq_i}$.

Application model: We consider that each workload arriving on the cluster is characterized by a tuple $\mathcal{A} = \langle Writes_{iso}, Reads_{iso}, C0_{iso} \rangle$, where $Writes_{iso}$, $Reads_{iso}$, $C0_{iso}$ refer to the mean values of the respective low-level performance counters (as described in section 4). As $Writes_{iso}$ and $Reads_{iso}$ refer to socket metrics, $C0_{iso}$ is related to core level metrics and is the average value of the socket's cores belonging to the examined cluster.

Summarizing our problem modeling and variables for the hardware model are the following:

Summary of Key Notations	
Notation	Definition
sv_i	i_{th} server
w	total number of servers in the cluster
s_j^i	j^{th} socket, part of server i
m_i	total number of sockets of server i
o_j	total number of cores per socket in server i
$n_l^{j,i}$	l_{th} node (VM) of server i and socket j
p_l	total number of cores in l_{th} node.
$c_k^{i,j,l}$	core in l_{th} node, part of the s^j socket
d_a	performance degradation of application a

The degradation of application i, d_i is defined as:

$$d_a = f_1(\mathcal{A}_a | S_1^t, S_2^t, ..., S_m^t, S_1^{t+1}, ..S_m^{t+1}, .., S_1^{t_n})$$

The degradation of the performance of each job depends on job's behaviour $\mathcal{A}$,

as well as on each candidate bin score S_j^t, $\forall t$ after the job placement until it departs from the bin. Moreover, the score of each socket j in time t, is:

$$S_j^t = S_j^{t-1} + f_2(d_{k,j}),\ \forall j$$

Any job can be placed in exactly one bin. As it can be observed, S_j^{t+1} and $d_{i,j}$ have a circular relationship, with the one being dependent on the value of the other. This fact makes our problem more difficult to get reduced into a known one. Also it is evident that the performance degradation d_a of job a depends on future incoming jobs placement.
The objective of our problem is the total minimization of jobs performance degradation.

$$\min \sum d_a, \forall a \in B$$

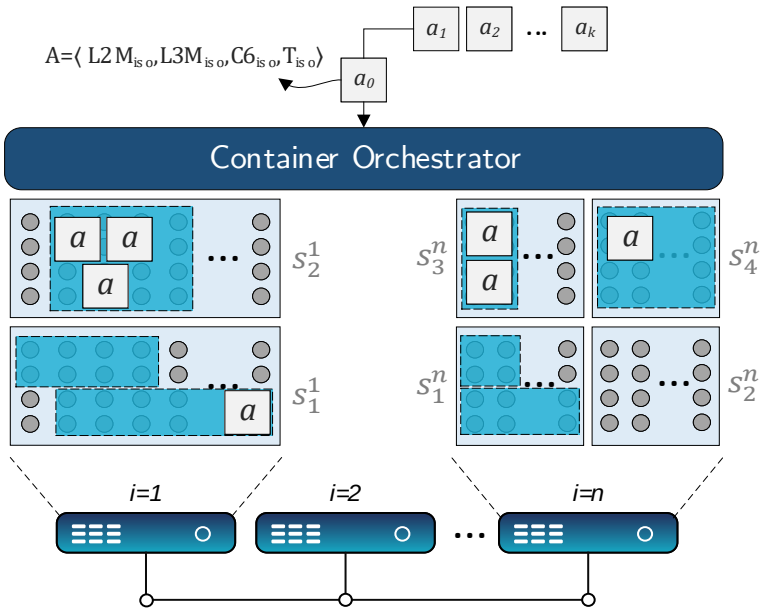


Figure 5.1: Cluster Architecture

5.2 Proposed Solution and Heuristic Algorithm Approach

In this section, we discuss our approach in the previous problem. Firslty, as we tested and evaluated in chapter 4 and illustrated in figure 4.10, the performance degradation of the applications scheduled, is highly correlated with our custom metric defined in equation 4.1. Taking this into account, when a placement decision should be made, we consider the bin with the less score as the bin where the incoming application will suffer from less degradation. This dynamic

scoring across the bins (sockets) is proportional to the already attached jobs' left execution time t_k, the bin's stress value $Stress_j$, as well as servers heterogeneity. Also, because of the application profiling, we are able to recognize different characteristics between the different applications we need to schedule. Those characteristics can make interference related delays more intensive. In other words, applications may have a very diverse performance depending on the other applications they are co-scheduled with.

Our approach solving this problem is an heuristic one, using a Best Fit algorithm. We create a scoring function for every bin and for every incoming item we pick the best one. The scoring function is the following:

$$S_j = Stress_j \times DF(t) \times HF$$

where DF (Duration Factor) is the impact of the approximate running applications duration which we need to take into account too. HF is the Heterogeneity variability. We discuss different functions of duration factor calculation in the next sections.

5.2.1 Parameter 1: Stress Score

Using PCM, we extract system, socket and core metrics. We calculate the weighted average of the metrics extracted over the last 20 seconds, with the latest metrics weighing the most. Metrics are divided into socket and core metrics. From the metrics provided we use the memory `reads` and `writes`, the cores' `C6-state` and `IPC` extracted in a 0.4 seconds interval. The stress score will be calculated by using the custom metric described in equation 4.1. Also, the need of core availability, should also been taken into account. So we divide this scoring function into two cases.

In the first case, no more than one core in average is idle in the nodes belonging in a specific socket.

$$C6_j \times o_j < 1$$

The Stress score there will be:

$$Stress_j = \frac{Reads + Writes}{IPC_j \times C6_j \times o_j}$$

In the second case, when there are more than one cores in average available in C6-state:

$$C6_j \times o_j >= 1,$$

the stress score will be:

$$Stress_j = \frac{Reads_j + Writes_j}{IPC_j \times 1 \times o_j}$$

where

$$IPC_j = \sum_{1}^{o_j} IPC_k, \; \forall c_k \in j$$

5.2.2 Parameter 2: Duration Factor

A complete interference-aware scheduler except for the node condition, needs to take into account the existing workload's duration too. However, this duration is currently not able to be calculated.

Regarding to the Duration Factor, which is a function depending to duration (t) of running applications, we tested and evaluated several approaches using the approximate values of average and max duration of scheduled applications.

Area Calculation using the average duration

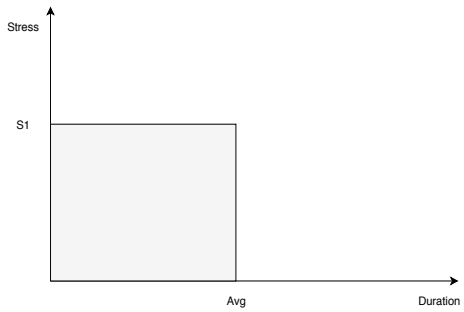


Figure 5.2

As it is illustrated in 5.4, in this approach, we multiply the average approximate calculated value of running applications duration with the stress of the current bin(socket/node). The final score in this case is gray area in ($\int_0^{avg} s1dx$).

$$DF = t_{avg}$$

Area Calculation using both average and maximum duration

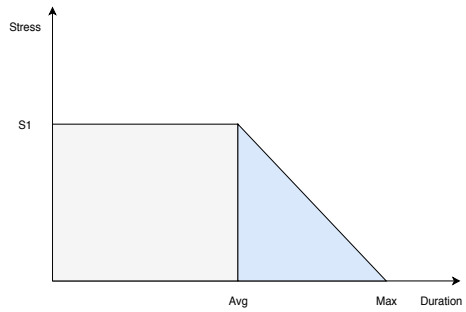


Figure 5.3

In this approach, we use both the average and the maximum value. For a duration until $t = average$ we multiply with the corresponding stress. Beyond that for $average < t < max$, we assume a linear degradation of the stress level, until $t = max$. The integral indicating the covered area is:

$$\int_0^{avg} S1dx + \int_{avg}^{max} (\frac{S1}{avg - max} \times x - \frac{S1 \times max}{avg - max})dx$$

$$DF = t_{avg} + \frac{t_{max} - t_{avg}}{2}$$

Duration - Stress Tendency

A question occurred while observing testing results, was what should the preferable node be if the product of $t \times S$ is the same between 2 or more nodes. In this case should the lowest stress level or the lowest duration be preferred. (e.g $node1\langle stress = s, average\_t = t\rangle, node2\langle stress = 10 \times s, average\_t = \frac{t}{10}\rangle$

We tried 2 different naive approaches using $d > 1$ as an exponent.

Duration Tendency Firstly we add some more value - bias to the average duration of the running applications. Regarding to the example above, $node2$ will get a lower score and will be selected.

$$DF = t^d \times S_j$$

Stress Tendency Next, we configured our scoring function in order to bias the stressing factor of the equation. Subsequently, $node1$ is going to result into a lower score an be the preferred one.

$$S_j = t \times S_j^d$$

Decay

Another factor that we thought should effect final node selection is the relationship between the average and the maximum duration of the scheduled applications. A big difference between the average and the maximum value of duration, actually indicates a bigger amount of applications with only a small amount of time remaining. Taking this into account we used the decay function in order to calculate the duration factor this time.

$$DF(t) = t_{max} \times (1 - e^{\frac{-t_{avg}}{t_{max}}})$$

Actually the more the average and the maximum value vary, the less final score should be given in the current socket (lesser score meaning greater viability).

Figure 5.4

5.2.3 Parameter 3: Heterogeneity Factor

Hardware heterogeneity can have significant impact on the performance of applications, especially for Latency-Critical (LC) applications [32, 33]. In this thesis, our cluster as a whole is heterogeneous. The two servers consisting the system differ in L2, L3 (LLC) cache sizes and way of associativity. The number of available cores, cache sizes, their base and maximum frequency and memory also varies. Last but not least the number and the width of Quick Path Interconnect (QPI) links or memory buses serving the memory requests are also different.

Trying to identify the state of each socket in order to prioritize them, we added one more factor, the HF, Heterogeneous factor. This one takes into account, the capability of each server to serve memory requests.

$$HF_j = \frac{1}{b_i \times l_i}$$

, where b is the bandwidth of each link in and l is the number of QPI links in server i.

5.3 Algorithm

We decided to use our proposed custom metric in our custom scheduler's implementation. We designed a 2-level approach regarding to the node selection. This leveled approach would be more efficient in large scale systems, because this way we filter out a great amount of candidate nodes. For example, our cluster is consisted of 2 servers, each server is consisted of 2 sockets and each socket's cores are pinned across 2 different VMs. As a result after selecting the most appropriate socket we cut the amount of possible nodes to one quarter. That is an approach able to scale in clusters with hundreds of sockets.

For the purposes of our design, we pinned each physical thread in the corresponding VM, in the appropriate core. (E.g. we mapped Host1/Socket0 to the kube-02).

```
server_socket_kubeNode =
{
    #(serverName, socketNumber):(vmName, coreNum)

  ("Host1",0,2):("kube-02",0),
  ("Host1",0,3):("kube-02",1),
  ("Host1",0,4):("kube-02",2),
  ("Host1",0,5):("kube-02",3),
  ("Host1",0,6):("kube-02",4),
  ("Host1",0,7):("kube-02",5),
  ("Host1",0,8):("kube-02",6),
  ("Host1",0,9):("kube-02",7),
    ...
    ...
}
```

The skeleton of the scheduler's implementation is the following. The separate parts and functions are discussed later in more detail.

initializeSockets();
for *a in applicationsQueue* **do**
  $\mathcal{A}_a \leftarrow$ Application(a);
  $s_j^i \leftarrow$ selectSocket($\mathcal{A}_a$);
  $n_l^{i,j} \leftarrow$ selectNode($s_j^i, \mathcal{A}_a$);
  schedule($n_l^{i,j}$);
end

Algorithm 1: Scheduler's main function

5.3.1 1st Level - Socket Selection

The First Level Filtering is conducted between all the candidate sockets of the cluster. In our infrastructure, we have to select one of the four available sockets. For the purposes of this stage, we extract socket metrics for every socket included in the cluster.

Also, another important fact here, which is also Kubernetes scheduler unaware of is that any Kubernetes cluster, especially the ones provided by Cloud Providers where the physical resources are not owned by the user, is that VMs resources are interfering with the residing physical resources. Usually in cloud environments cores belonging to the same physical machine may be shared between two or more different clusters. In this case, Kubernetes is unaware of any interference happening because of shared physical resources (such as socket).

First of all we gather the information we need for each socket. Socket is the first level of abstraction where resources such as L3 cache and Memory Bandwidth are shared between their sub components. In previous section, we presented the impact on the performance of applications due to interference into those shared socket resources.

Socket Initialization

The function called first is the `initializeSockets()` (Algorithm 2). In this function pcm extracted metrics through the installed NFS are read. It is called every time a new batch of applications, ready to get scheduled, arrives.

for $i \leftarrow 0$ *to* w **do**
  $s^i \leftarrow$read(),$\forall j \in i$;
  for $j \leftarrow 0$ *to* m_i **do**
    $s^i_j\langle C6\rangle \leftarrow initCores(server_i, s^i_j)$;
    calculateSocketScore(s^i_j,timer);
  end
end

Algorithm 2: Sockets initialization

The C6-State we set on the `Socket` object is not the socket level metric provided by `PCM`. The reason we use this metric is to take into account the cores availability. As we need a more accurate number, instead of using the C6-state average of the socket, we calculate a new C6-state average using only the cores' of each socket being part of our cluster. This is conducted in the `initCores()` function described in algorithm 3. It intializes also the nodes metrics, according to the pinned cores of the system.

$c_k \leftarrow$ read() $\forall k$;
for $k \leftarrow 0$ *to* o_j **do**
  if $c_k^{i,j} \in n_l^{i,j}$, $\forall l \in$ *cluster* **then**
    update $n_l^{i,j}\langle C6\rangle$;
    update $s_j^i\langle C6, IPC\rangle$;
  end
end
return $s_j^i\langle C6\rangle$

Algorithm 3: Cores initialization

Socket Score Calculation

The sockets after being intialized with the metrics extracted, they need to be prioritized based on a scoring function. There are three parts which consist this function: stress score, duration factor and heterogeneity factor.

At first place in the `calculateSocketScore()` function, we abstract from each scheduled application the time passed since its scheduling and calculate the average and the max duration for every socket.

for $\forall \mathcal{A}_a \in s_j^i\langle apps\rangle$ **do**
  tRemaining $\leftarrow \mathcal{A}_a\langle duration\rangle - \mathcal{A}_a\langle arrival\rangle -$timer;
  if *tRemaining* > 0 **then**
    avg $\leftarrow$ tRemaining + avg;
    max $\leftarrow$ max(tRemaining,max)
  else
    $s_j^i\langle apps\rangle.remove(\mathcal{A}_a)$;
  end
end

Algorithm 4: Running Applications' Duration

Finally, we select the socket with the minimum total score. Next, we update the reads and writes count temporarily by adding the scheduled application's metrics and calculating a new approximate value until real metrics are read again. After that, we get to the next level of our process, selecting a node among the ones being part of the selected socket.

5.3.2 2nd Level - Node Selection

After selecting the most appropriate socket according to our priority function, the L3 cache and memory bandwidth are not prioritizing factors anymore as their are shared between the cores within the socket. The next level of abstraction in our system, are the nodes (VMs). Owing to node heterogeneity, in

terms of virtual cores and memory capacity in this stage of our approach we need to choose the most appropriate of the nodes whose cores are pinned to the selected socket. We use the number of cores of each node in combination with the average c6-state of those cores and we select the predominant node (with the least score) to schedule the current application.

$$Score_n^{i,j}{}_l = \sum_{k=1}^{k=p_l} c_k^{i,j,l} \langle C6 - State \rangle$$

Next we update again the node's c6-state using application's average value.

5.3.3 Pod Placement

Finally, after the most viable node is selected from our algorithm, we use *node affinity* functionality in order to schedule the pod to the desired node (VM). Using the appropriate yaml file which describes the pod's characteristics and containers image, we create a copy for each application ready to get scheduled, changing the pod's name with a unique id, pod's most desired node (affinity) and the scheduler to be used (my-scheduler). Those pod's yaml file is presented below:

```
apiVersion: v1
kind: Pod
metadata:
  name: scikit-pod-lasso
spec:
  restartPolicy: OnFailure
  schedulerName: my-scheduler
  containers:
  - name: scikit-container
    image: registry.hub.docker.com/imageX
    resources:
      requests:
        memory: "1648Mi"
    env:
    - name: CLF
      value: "Lasso"
    command: ["/bin/bash","-c" ]
    args:
      - time /workloads/fit_${CLF}.py "/784x40000.data" "/784x40000.labels" "1"
    ports:
    - containerPort: 80
    imagePullPolicy: Always
  imagePullSecrets:
  - name: regsecret
  affinity:
    nodeAffinity:
      preferredDuringSchedulingIgnoredDuringExecution:
      - weight: 100
        preference:
          matchExpressions:
          - key: kubernetes.io/hostname
            operator: In
            values:
            - kube-08
```

Chapter 6

Evaluation

In this chapter, we evaluate the results of our proposed approach in various scenarios. Firstly, by using a single server we evaluate our custom stress function and various duration factor approaches. Next, by executing our workloads in our two heterogeneous servers, H1 and H2 as described in 4.4 and 4.5, we evaluate the efficacy of our proposed Heterogeneity Factor (HF - sec. 5).

6.1 Single Server

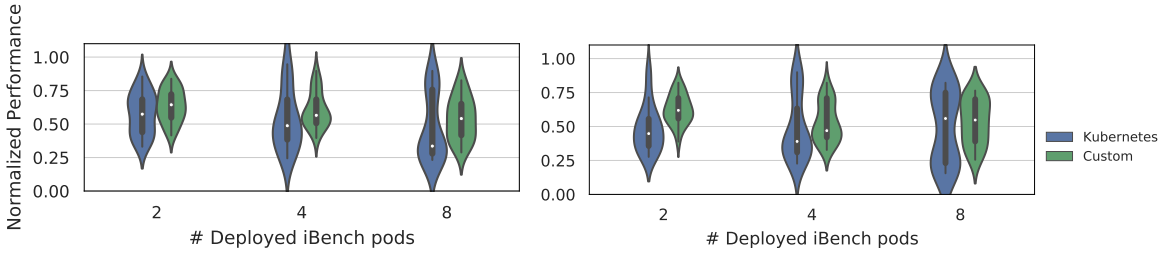
In this first part of the current chapter, we evaluate the performance of applications scheduled by Kubernetes scheduler and our custom solutions on VMs residing on the top of a single server. We evaluate pod placement under different scenarios, using our proposed custom scheduler.

6.1.1 Stressing one Socket

To begin with, we tested our proposed scheduler awareness regarding to the already scheduled workload. Using the ibench micro-benchmarks we stressed different shared resources of one of the two sockets consisting our infrastructure. So as to avoid server heterogeneity, we used only one of our two servers. However the VMs were of different resources capacity.

Virtual Machines				
VM-Name	**Server**	**socket**	**Cores**	**RAM (GB)**
kube-05	s2	0	4	8
kube-06	s2	1	8	16
kube-07	s2	0	16	32
kube-08	s2	1	16	64

In order to succinctly showcase the tests results, we used the normalized performance, by dividing the elapsed time in isolated execution with the re-



(a) Memory Bandwidth stress (b) L3 stress

Figure 6.1: Applications relative performance after being co-scheduled with pre-existing stressing workload.

spective time occurred when co-scheduled. In addition, we used violin plot to better illustrate applications' elapsed time distribution.

$$performance = \frac{isolated\_time}{elapsed\_time}$$

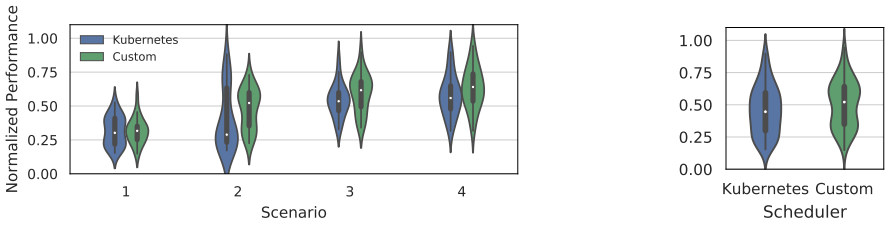
Memory Bandwidth

Memory pressure impacts system's performance, as due to the limited bandwidth, application competition and interference occurs. As shown in Fig. our scheduler achieves a higher median in the y-axis which presents applications' normalized performance. This is an indicator of performance improvement as median values are 12.2%, 15.9% and 61.2% greater respectively. Moreover, the results' standard deviation is 21.88%, 47.27% and 35.77% smaller respectively in our proposed scheduler than Kubernetes. As it is also shown in the violin plots, the density of applications relative performance is accumulated in higher values. This compact variation leads to a more predictable workload performance.

L3 Cache

Similarly to the previous testcases, we stressed this time one of the sockets with different intensities of L3 cache pressure using `L3-ibench` pods. The median of the workloads performance is higher in our approach when scheduling 2 and 4 pods by 38.3% and 20.4% respectively. In 8 pods case, the two medians are identical between Kubernetes and our custom scheduler, 0.5588 and 0.5476 respectively (2% lower). The standard deviation of the workload in this c is 21.9%, 47.3% and 35.8% less in our approach in 2,4 and 8 pods deployment respectively.

What we observed is that Kubernetes unawareness about system metrics, lead to completely suboptimal application scheduling causing a great degradation in performance. While Kubernetes was choosing almost naively the winner node, without taking into account the great interference `l3-ibench` was cre-



(a) Different Scenarios (b) Aggregated

Figure 6.2: Applications relative performance after being co-scheduled with pre-existing stressing workload.

ating, our custom approach was avoiding scheduling pods in a stressed shared resource. As long 2 different VMs were pinned to physical cores residing in the same socket, it was critical both to know about the stress of the current resource and the awareness of the system about the core pinning across the virtual nodes.

6.1.2 Stressing both sockets

In this section, after scheduling random ibench workload in our system, consisted of `cpu,L3` and `membw` pressure , we tried to schedule 25 random applications arriving in batches in different time intervals. We compared again our custom approach with Kubernetes scheduler and both the individual and aggregated results are illustrated in figure 6.2. Among the different tests, the median occurred by our scheduler was from 4.9-80.8% higher than the default and the overall average was 16.8% greater as well.

Scenario	Socket0			Socket1		
	CPU-pods	**L3-pods**	**Mem-Bw pods**	**CPU-pods**	**L3 pods**	**Mem-Bw pods**
1	1	3	2	3	4	3
2	3	1	0	2	3	4
3	2	0	3	4	1	0
4	4	1	1	2	1	0

6.1.3 Scheduling in the absence of artificial stress

Previously, we tested how our proposed scheduler would behave in a pre-existing pressure, and its ability to share the incoming workload between the available resources. In this subsection, we schedule workloads consisted of different amount of applications fluctuating from 20 to 100. The workloads included applications referred in section 4.2. Those workloads were also consisted of numerous batches arriving in our system on an interval fluctuating from 30 to 50 seconds. Fig. 6.3 illustrates the results of scheduling workloads varying

from 20 to 100 applications in a 44-core Kubernetes cluster consisted of Host 1 presented in table 4.4.

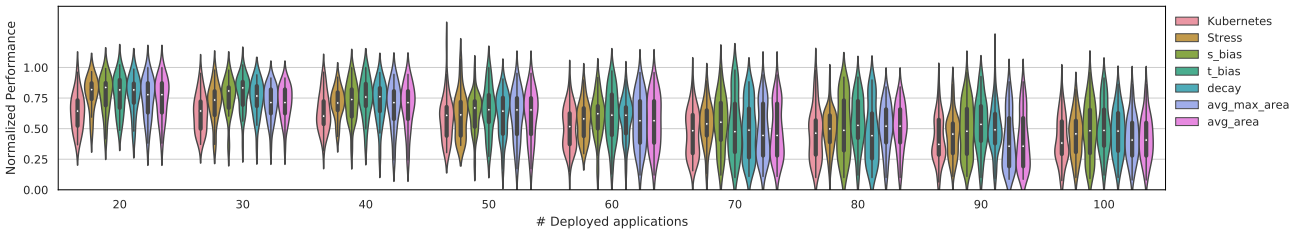
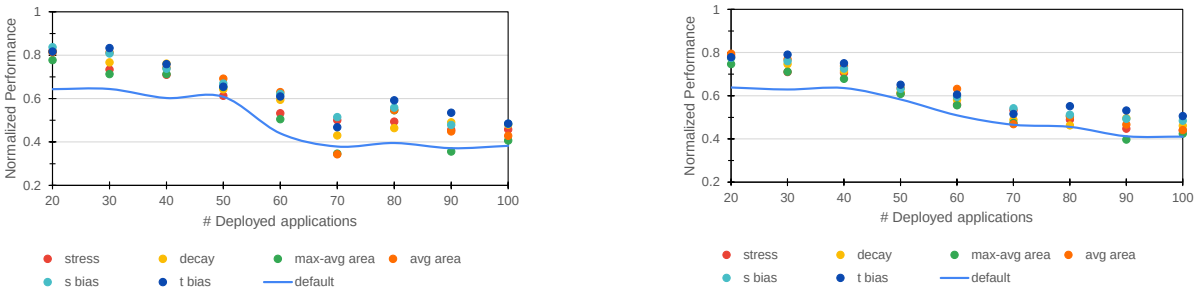


Figure 6.3: Applications normalized performance distribution across multiple scheduler design approaches.

Violin plots illustrate the distribution of applications' elapsed times. In y axis is the normalized performance of those applications. From this figure, it is evident that for smaller number of deployed workloads our scheduler achieves much higher performance over the default scheduler, with 24.5% higher performance on average achieved in average area approach. For higher number of workloads we see that the performance gap between our proposed scheduler and Kubernetes one shrinks. This is due to the fact that this huge amount of workloads force our system to be saturated and, therefore, proper scheduling does not affect the overall performance of workloads. However, even in such over-stressed scenarios, we can still see that there is a clear advantage of our proposed approach over the naive one.



(a) Median

(b) Average

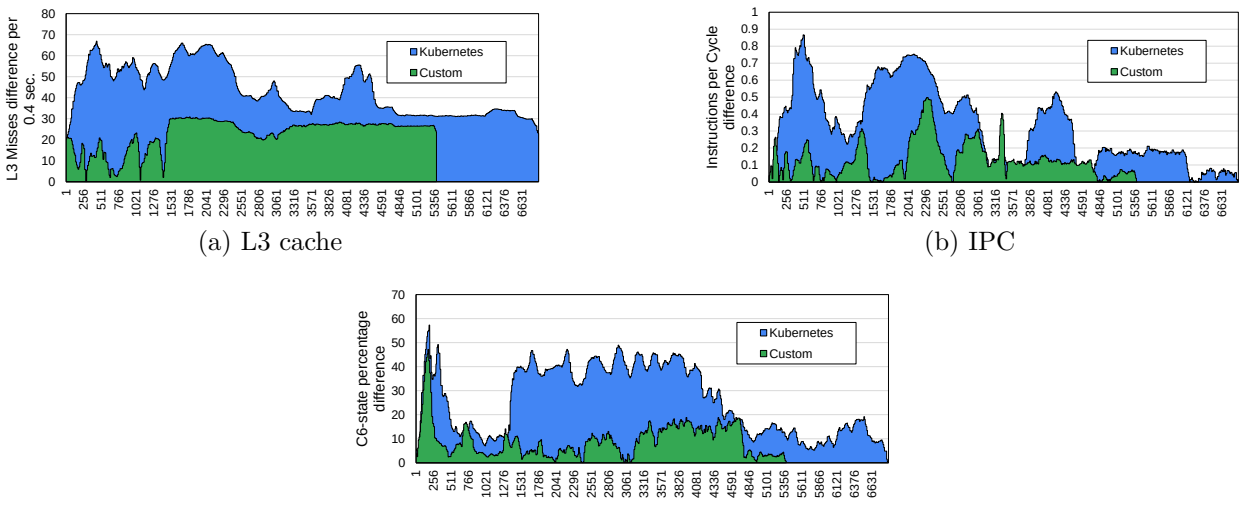
Figure 6.4: Different models medians and average values comparison.

A more clear intuition can be obtained through figures 6.4a and 6.4b where the median and average values are illustrated respectively. The majority of the tested approaches overtake default Kubernetes scheduler results. From the tests conducted, t bias as the duration factor ($DF(t)$) approach seems to achieve the higher values both in medians and averages. More specifically, it achieves higher median by 26.9%, 29.3%, 25.9%, 7.6%, 38.8%, 23.4%, 49.7%, 44.0% and 26.9% for the 20, 30, 40, 50, 60, 70, 80, 90 and 100 applications placement respectively. Similar results occur regarding to the average value which from 10.7% to 28.9% higher than the default one.

6.1.4 Available Resources Usage

Data-centers today suffer from under-utilization. Due to high level resources allocation, a low-level resource contention is either not manageable or it is prevented by reserving more and more resources. However, a more even share of the workload between isolated resources belonging to the same or different systems, is able restrict this phenomenon.

In Fig. 6.5, we present some hardware system metrics extracted during one of the above tests. More specifically, each figure describes resource usage from each socket during the execution of our workloads. In Fig. 6.5a, the imbalance between each socket's cache misses in Kubernetes scheduler is significantly greater than our proposed one. Consequently, one of the two sockets was over-utilized, degrading running applications performance due to contention, while the other one had that specific resource available and unused. On the other side, our proposed scheduler is aware of the L3 cache interference and distributes applications in a more evenhanded way, trying to share the load between separate components of the system. Similarly in fig. 6.5b and 6.5c is displayed the more balanced resource usage during the execution of the workload our proposed approach scheduled. C6-state percentage which represents the inactive cores and the Instructions per Cycle seem also to come up against a more equitable sharing in our custom approach.



(a) L3 cache

(b) IPC

(c) C6-state

Figure 6.5: Resources Usage imbalance between the sockets.

6.2 Heterogeneous System

Finally, we evaluated the heterogeneous scheduling, and how our proposed approach schedules incoming pods compared to Kubernetes native scheduler.

Combining Host 1 and Host 2 in a Kubernetes cluster and using a catholic scheduler approach we schedule 40, 80, 160, 320 and 640 applications. The distribution of the different approaches applications relative performance is illustrated using violin plots in figure 6.6.

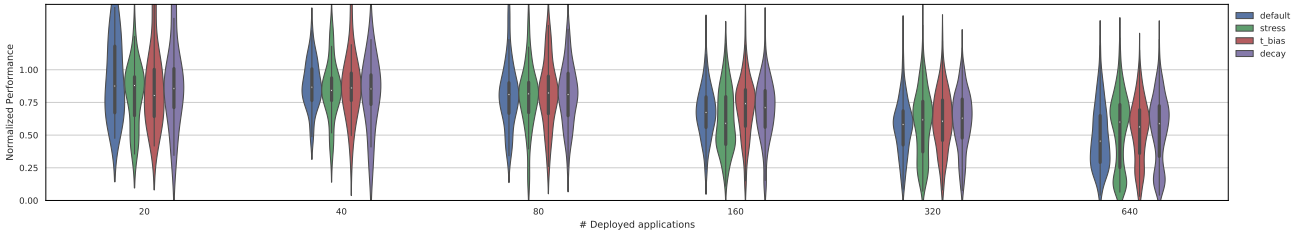


Figure 6.6: Comparison between different approaches in applications' relative performance distribution in Heterogeneous Kubernetes cluster.

In lower workload density, scheduler performance varies between the different approaches. Cluster resources are more than enough to consume this small amount of incoming applications and the performance varies. The median value of distribution of our approaches is 1%-3% lower than the default. However, when more and more applications get scheduled and the system is under contention, our approaches average value increases. More specifically the median value of the distribution is 6.5% and 28.4% higher in average than the default in the 320 and 640 applications scheduling respectively.

Regarding to the `t bias` approach which stood out among all the other approaches in the previous section, it achieved -8%, -0.7%, +0.1%, +10%, +4.6%, +23.8% lower/higher median compared to the default scheduler in the 20,40,80,160,320 and 640 application placement scenarios respectively.

Chapter 7

Conclusion and Future Work

7.1 Summary

In this thesis, we discussed data-centers dual optimization goal of maximizing performance and resource utilization at the same time. Moreover, we described virtualization and containerization technologies as well as the trend for their complementary use. Regarding to the orchestrator of the previously referred containers, we presented Kubernetes.

Interference and heterogeneity are two major concerns for data operators nowadays. Attenpting to observe such phenomena, we analyzed the performance of workloads from different scientific benchmarking libraries, under pressure on various resources in different intensities. Thus, we identified resources contention and we proposed a highly correlated with application slowdown indicator able to depict system's condition. In addition we executed the same workloads with the same resources stress specification for another system, and we analyzed the deviation in performance occurred. After observing differences between heterogeneous systems behavior, we recognized the need for heterogeneous aware scheduling.

Furthermore, we designed an integrated with Kubernetes interference-aware scheduler and implemented it on the top of a virtualized environment. We tested different approaches regarding to the selection of our scoring function, which is consisted of three different parameters (stress, duration factor, heterogeneity factor). Moreover, we evaluated the pod placement of our proposed scheduler on a single server, using different scenarios and compared it with kube-scheduler. We showed that in most of the scenarios, our custom approach improves the average performance of the deployed workloads by 20% ,the median of distribution by 30.3% and achieves a more balanced resource utilization at the same time. Finally, we evaluated pod placement of our proposed approach when it comes to heterogeneous clusters. While we approached the heterogeneity factor with a primitive factor, we observed that our proposed scheduler achieved better results in high workload densities, when the system

was under contention.

7.2 Future Work

The analysis, observations and proposals described in this thesis were an immature attempt to identify, describe and quantify interference and heterogeneity phenomena. In the following subsections, future work is suggested. We categorize those suggestions into two groups, the ones related to development optimizations and the ones related to further research opportunities.

7.2.1 Development Scope

Regarding to the development of the system, future work could include the use of another framework in order to extract the needed metrics from the system. This framework may extract metrics in a less frequent ratio, process them wasting less resources and finally using a "smart" database, enabling faster querying.

Furthermore, such a system could be plugged-in Kubernetes project. Metrics extracted from the system will be communicated to the Kubernetes API, as custom resources. Also, proposed scoring functions can be implemented in kube-scheduler native code. Regarding to heterogeneity, the code can be extended to take into consideration modern system resources such as GPU.

7.2.2 Research Scope

On the other hand, we also suggest some research subjects as proposed future work. First of all, a Neural Network could be used to better estimate the condition of each candidate server and node. Instead of relying on an observation-based custom score, the scoring function could be the prediction of a well trained Neural Network.

Moreover, runtime control of resources could be implemented using PCM metrics extracted. CPU frequency and RAM usage can be manipulated depending on workload scheduled on pinned cores intensity. Additionally, alongside with the scheduler, Cache Allocation Technology (CAT) [55] can also be used. CAT enables privileged software such as an OS or VMM to control data placement in the last-level cache (LLC), enabling isolation and prioritization of important threads, apps, containers, or VMs. Using CAT, latency-critical tasks running on the cluster can be prioritized as illustrated in figure 7.1.

Last but not least, energy efficiency is a concern being discussed a lot over the years. Running Average Power Limit (RAPL) provides a way to set power limits on processor packages and DRAM. This will allow a monitoring and control program to dynamically limit max average power, to match its expected

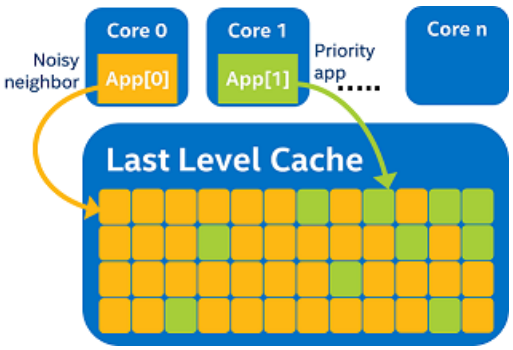


Figure 7.1: A "noisy neighbor" on core zero over-utilizes shared resources in the platform, causing performance inversion (though the priority app on core one is higher priority, it runs slower than expected).[1]

power and cooling budget. In addition, power limits in a rack enable power budgeting across the rack distribution. By dynamically monitoring the feedback of power consumption, power limits can be reassigned based on use and workloads. Because multiple bursts of heavy workloads will eventually cause the ambient temperature to rise, reducing the rate of heat transfer, one uniform power limit can't be enforced. RAPL provides a way to set short term and longer term averaging windows for power limits. These window sizes and power limits can be adjusted dynamically.

Bibliography

[1] C. V. networking Index, "Forecast and methodology, 2016-2021, white paper," *San Jose, CA, USA*, vol. 1, 2016.

[2] *RedHat official website*, https://www.redhat.com/.

[3] *Google Cloud Platform*, https://cloud.google.com/.

[4] *Amazon Elastic Compute Cloud*, https://aws.amazon.com/ec2/.

[5] 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*. ACM, 2010, pp. 557–558.

[6] L. A. Barroso and U. Hölzle, "The case for energy-proportional computing," 2007.

[7] R. McMillan, "Data center servers suck-but nobody knows how much," *Wired magazine, www. wired. com/2012/10/data-center-servers*, 2012.

[8] 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.

[9] C. Delimitrou and C. Kozyrakis, "Quasar: resource-efficient and qos-aware cluster management," in *ACM SIGARCH Computer Architecture News*, vol. 42, no. 1. ACM, 2014, pp. 127–144.

[10] 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*. ACM, 2012, p. 7.

[11] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, omega, and kubernetes."

[12] 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, no. 2011, 2011, pp. 22–22.

[13] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and linux containers," in *2015 IEEE international symposium on performance analysis of systems and software (ISPASS)*. IEEE, 2015, pp. 171–172.

[14] D. Masouros, S. Xydis, and D. Soudris, "Rusty: Runtime system predictability leveraging lstm neural networks," *IEEE Computer Architecture Letters*, vol. 18, no. 2, pp. 103–106, 2019.

[15] 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.

[16] 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.

[17] P. Authors, "Prometheus-monitoring system & time series database," 2017.

[18] K. Yang, "Aggregated containerized logging solution with fluentd, elasticsearch and kibana," *International Journal of Computer Applications*, vol. 150, no. 3, 2016.

[19] F. Romero and C. Delimitrou, "Mage: Online interference-aware scheduling in multi-scale heterogeneous systems," *arXiv preprint arXiv:1804.06462*, 2018.

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

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

[22] 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 *ACM Sigplan Notices*, vol. 45, no. 3. ACM, 2010, pp. 335–346.

[23] F. Guo, Y. Solihin, L. Zhao, and R. Iyer, "A framework for providing quality of service in chip multi-processors," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2007, pp. 343–355.

[24] S. Kim, D. Chandra, and Y. Solihin, "Fair cache sharing and partitioning in a chip multiprocessor architecture," in *Proceedings. 13th International Conference on Parallel Architecture and Compilation Techniques, 2004. PACT 2004.* IEEE, 2004, pp. 111–122.

[25] D. Firesmith, *Virtualization via Containers*, carnegie Mellon University Blogs.

[26] X. Li, Z. Qian, S. Lu, and J. Wu, "Energy efficient virtual machine placement algorithm with balanced and improved resource utilization in a data center," *Mathematical and Computer Modelling*, vol. 58, no. 5-6, pp. 1222–1235, 2013.

[27] S. Zhuravlev, S. Blagodurov, and A. Fedorova, "Addressing shared resource contention in multicore processors via scheduling," in *ACM Sigplan Notices*, vol. 45, no. 3. ACM, 2010, pp. 129–142.

[28] 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*. ACM, 2019, p. 39.

[29] 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, no. 3. ACM, 2015, pp. 450–462.

[30] 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, ISCA 2019, Phoenix, AZ, USA, June 22-26, 2019*, 2019, pp. 462–473. [Online]. Available: https://doi.org/10.1145/3307650.3322234

[31] M. Ferdman, C. Kaynak, and B. Falsafi, "Proactive instruction fetch," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-44. New York, NY, USA: ACM, 2011, pp. 152–162. [Online]. Available: http://doi.acm.org/10.1145/2155620.2155638

[32] J. Mars and L. Tang, "Whare-map: heterogeneity in homogeneous warehouse-scale computers," in *ACM SIGARCH Computer Architecture News*, vol. 41, no. 3. ACM, 2013, pp. 619–630.

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

[34] D. Merkel, "Docker: Lightweight linux containers for consistent development and deployment," *Linux J.*, vol. 2014, no. 239, Mar. 2014. [Online]. Available: http://dl.acm.org/citation.cfm?id=2600239.2600241

[35] V. Inc., "Containers on virtual machines or bare metal?" *Deploying and Securely Managing Containerized Applications at Scale, White Paper*, Dec. 2018.

[36] *Processor Counter Monitor (PCM)*, https://github.com/opcm/pcm.

[37] 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.

[38] J. L. Henning, "Spec cpu2006 benchmark descriptions," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 4, pp. 1–17, 2006.

[39] 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.

[40] C. Delimitrou and C. Kozyrakis, "ibench: Quantifying interference for datacenter applications," in *2013 IEEE international symposium on workload characterization (IISWC)*. IEEE, 2013, pp. 23–33.

[41] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[42] Y. Freund and R. E. Schapire, "A decision-theoretic generalization of on-line learning and an application to boosting," *Journal of computer and system sciences*, vol. 55, no. 1, pp. 119–139, 1997.

[43] 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.

[44] 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*. ACM, 2010, pp. 143–154.

[45] A. Cassandra, "Apache cassandra," *Website. Available online at http://planetcassandra. org/what-is-apache-cassandra*, p. 13, 2014.

[46] *NGINX official website*, https://www.nginx.com/.

[47] B. Fitzpatrick, "Distributed caching with memcached," *Linux journal*, vol. 2004, no. 124, p. 5, 2004.

[48] *MySQL official website*, https://www.mysql.com/.

[49] *Elgg official website*, https://www.elgg.org/.

[50] *Faban official website*, http://www.faban.org/.

[51] T. Hastie, S. Rosset, J. Zhu, and H. Zou, "Multi-class adaboost," *Statistics and its Interface*, vol. 2, no. 3, pp. 349–360, 2009.

[52] *The CMU Audio Databases*, http://www.speech.cs.cmu.edu/databases/an4/.

[53] 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.

[54] Y. Li, X. Tang, and W. Cai, "Dynamic bin packing for on-demand cloud resource allocation," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 1, pp. 157–170, 2015.

[55] C. Intel, "Improving real-time performance by utilizing cache allocation technology," *Intel Corporation, April*, 2015.

[56] *Kubespray GitHub repository*, https://github.com/kubernetes-sigs/kubespray.

Chapter 8

Appendix

1 Kubernetes Cluster Setup

In this thesis, Kubespray[56] is used to install and make the initial setup of the Kubernetes cluster.

In the next few lines, the process which was followed is presented.

Step 1: Clone the git repository

Kubespray offers a variety of choices according to the Cluster configuration, such as naming the kube-nodes, classifying the nodes (master-workers), choose the network plugin and also select the Kubernetes release to be installed.

Step 2:

- ☐ Configure the inventory/sample/hosts.ini file using your own IPs and name for each node
- ☐ Configure the inventory/sample/group_vars/k8s_cluster/k8s_cluster.yml

```
1    kube_network_plugin: flannel
```

Step 3:

- ☐ Exchange RSA keys both private and public between VMs so as to communicate each other
- ☐ Disable firewall

```
1 $sudo ufw disable
```

- ☐ Enable ip forwarding

```
1 $sudo sysctl -w net.ipv4.ip_forward=1
```

- ☐ Disable swap

```
1 $swappoff -a
```

Step 4:

□ Install dependencies from requirements.txt

□ Run as root the ansible playbook :

```
$ansible-playbook  --private-key=/path/to/private/key --user=ubuntu \
    -i inventory/mycluster/hosts.ini --become --become-user=root cluster.yml
```

Step 5:

Run in Master Node

```
    $ mkdir -p $HOME/.kube
    $ sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
    $ sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

2 Custom Kubernetes Scheduler Setup

As it was referred before, the default scheduler of Kubernetes uses some pre-selected functions for its decision making.

□ Change the native code accordingly, compile, create a docker container and push it in a Docker registry

```
    #!/bin/sh
    KUBERNETES_PATH="/path/to/kubernetes/native_code"
    DOCKERFILE_PATH="${KUBERNETES_PATH}/_output/bin/"

    echo $KUBERNETES_PATH
    echo $DOCKERFILE_PATH

    cd $KUBERNETES_PATH
    sudo make all WHAT=cmd/kube-scheduler/
    cd $DOCKERFILE_PATH
    sudo docker build -t iwita/scheduler .
    sudo docker push iwita/scheduler:latest
    cd
    echo "END"

```

□ Create the deployment file (yaml)

```
    apiVersion: v1
    kind: ServiceAccount
    metadata:
      name: my-scheduler
      namespace: kube-system
    ---
    kind: ClusterRoleBinding
    apiVersion: rbac.authorization.k8s.io/v1
    metadata:
      name: my-scheduler-as-kube-scheduler
    subjects:
    - kind: ServiceAccount
      name: my-scheduler
      namespace: kube-system
```

```
  roleRef:
    kind: ClusterRole
    name: system:kube-scheduler
    apiGroup: rbac.authorization.k8s.io
  ---
  apiVersion: apps/v1
  kind: Deployment
  metadata:
    name: my-scheduler
    labels:
      component: scheduler
      tier: control-plane
    name: my-scheduler
    namespace: kube-system
  spec:
    selector:
      matchLabels:
        component: scheduler
        tier: control-plane
    replicas: 1
    template:
      metadata:
        labels:
          component: scheduler
          tier: control-plane
          version: second
      spec:
        serviceAccountName: my-scheduler
        containers:
        - command:
          - kube-scheduler
          - --leader-elect=false
          - --scheduler-name=my-scheduler
          image: docker.io/iwita/scheduler:latest
          livenessProbe:
            httpGet:
              path: /healthz
              port: 10251
            initialDelaySeconds: 15
          name: my-scheduler
          readinessProbe:
            httpGet:
              path: /healthz
              port: 10251
          resources:
            requests:
              cpu: '0.1'
          securityContext:
            privileged: false
          volumeMounts: []
          env:
            - name: NODE_NAME
              valueFrom:
                fieldRef:
                  fieldPath: spec.nodeName
        nodeSelector:
          kubernetes.io/hostname: kube-00

        hostNetwork: false
        hostPID: false
        volumes: []

```

□ Enable the scheduling in master node

```
$ kubectl taint nodes kube-00 node-role.kubernetes.io/master:NoSchedule-
```

- Create the deployment

```
$ kubectl create -f my-scheduler.yaml
```

- Disable again the scheduling in master node

```
$ kubectl taint nodes kube-00 node-role.kubernetes.io/master=:NoSchedule
```

- Finally append in the end of the file that the following command opens

```
$ kubectl edit clusterrole system:kube-scheduler
```

the following lines

```
- apiGroups:
  - storage.k8s.io
  resources:
  - storageclasses
  verbs:
  - watch
  - list
  - get
```

3 NFS Setup

In each server there is a folder containing all the needed scripts for metrics extraction and sharing. The process is the following:

- Start the PCM and redirect output to serverX.csv
- Create a daemon that cuts the output file serverX.csv keeping only the columns names which are needed for specific metrics extaction and the last 50 lines.
- Create another daemon that extracts the socket metrics
- Create another daemeon that extracts the core metrics for every socket
- Finally calculate the weighted averages and send them in the shared folder with the Network File System.

```
#!/bin/bash

cd
sudo modprobe msr
sudo ./pcm.x 0.4 -r -csv=/path/pcm-server1.csv 1>&- 2>&- &


cd /path

while true; do
  head -n 3 pcm-server1.csv > pcm-server1_cut.csv_temp &&
  tail -n 50 pcm-server1.csv >> pcm-server1_cut.csv_temp &&
  mv pcm-server1_cut.csv_temp pcm-server1_cut.csv
  sleep 2s
done &
```

```
16
17 #Extract the socket metrics
18 ./socket_extraction_manager.sh &
19
20 #Extract the core metrics
21 ./core_extraction_manager.sh &
22
23 #Calculate Averages and move them in the shared folder
24 ./orchestrator.sh &
25
26 wait
```

```
1  #!/bin/bash
2
3  #This scirpt will run on server1 server and will share server1 metrics
4  #with Kubernetes Master VM (kube-00) using a NFS.
5
6
7  SHARED_FOLDER="/mnt/metrics_client/"
8
9  while true
10 do
11
12   ./nfs-server1-calculateAverage.py > "${SHARED_FOLDER}server1_sockets.out_temp" &&
13   mv "${SHARED_FOLDER}server1_sockets.out_temp" "${SHARED_FOLDER}server1_sockets.out"
14
15   ./nfs_init_cores_server1_socket0.py > "${SHARED_FOLDER}server1_socket0.out_temp" &&
16   mv "${SHARED_FOLDER}server1_socket0.out_temp" "${SHARED_FOLDER}server1_socket0.out"
17
18   ./nfs_init_cores_server1_socket1.py > "${SHARED_FOLDER}server1_socket1.out_temp" &&
19   mv "${SHARED_FOLDER}server1_socket1.out_temp" "${SHARED_FOLDER}server1_socket1.out"
20
21   sleep 1s
22 done
```