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

Κατανομή Εφαρμογών σε Υπολογιστικά Συστήματα

Διπλωματική

Δημήτριος Κ. Δόλογλου

Επιβλέπων Καθηγητής: Γεώργιος Γκούμας
Καθηγητής Ε.Μ.Π.

Αθήνα, Απρίλιος 2022

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

Κατανομή Εφαρμογών σε Υπολογιστικά Συστήματα

Διπλωματική

Δημήτριος Κ. Δόλογλου

Επιβλέπων Καθηγητής: Γεώργιος Γκούμας
Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 13η Απριλίου 2022.

(Υπογραφή)	(Υπογραφή)	(Υπογραφή)
.....	.....	.....
Γκούμας Γεώργιος	Πνευματικάτος Διονύσιος	Κοζύρης Νεκτάριος
Καθηγητής Ε.Μ.Π.	Καθηγητής Ε.Μ.Π.	Καθηγητής Ε.Μ.Π.

Αθήνα, Απρίλιος 2022

…….

Δημήτριος Κ. Δόλογλου

Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π

Περίληψη

Ο σκοπός της διπλωματικής εργασίας ήταν η ανάπτυξη αλγορίθμων για την βελτιστοποίηση της διαδικασίας κατανομής εφαρμογών σε υπολογιστικά νέφη. Καθώς η χρήση του υπολογιστικού νέφους αυξάνεται, ταυτόχρονα αυξάνονται και οι απαιτήσεις υπολογιστικών πόρων για την διάθεση υπηρεσιών, όπως η εξυπηρέτηση εφαρμογών. Όμως, η παροχή υπολογιστικών πόρων αποτελεί σημαντικό κόστος και είναι φυσικά αδύνατο να έχουμε απεριόριστους πόρους. Συνεπώς είναι επιτακτική η υλοποίηση αλγοριθμικών τεχνικών που κατανέμουν εφαρμογές σε υπολογιστικά νέφη με τέτοιον τρόπο, ώστε αφενός να εξοικονομούμε τη διάθεση πόρων κατά το μέγιστο δυνατό και αφετέρου να διατηρούμε την εξυπηρέτηση των εφαρμογών στο απαιτούμενο επίπεδο.

Για την κατανομή εφαρμογών σε υπολογιστικά συστήματα, χρησιμοποιείται ευρέως ένα σύστημα ενορχήστρωσης γνωστό ως Kubernetes. Το Kubernetes αξιοποιεί την χρήση των containers για την διαχείριση και κατανομή εφαρμογών. Μάλιστα ένα από τα main components του Kubernetes είναι ο Scheduler του, ο οποίος καθορίζει το πως και το πότε θα κατανεμηθούν οι εφαρμογές που καταφτάνουν στον cluster. Σε αυτό ακριβώς το σημείο προσπαθούμε να εισάγουμε αλγοριθμικές τεχνικές, οι οποίες επηρεάζουν την συμπεριφορά του Scheduler και καθορίζουν τον τρόπο που θα κατανεμηθούν οι εφαρμογές. Ως αποτέλεσμα, δημιουργούνται ζεύγη εφαρμογών, με σκοπό την καλύτερη εξοικονόμηση των διαθέσιμων πόρων αλλά και την αδιάλειπτη διαθεσιμότητα των εφαρμογών

Για την πειραματική αξιολόγηση των παραπάνω, υλοποιήθηκε ένας server από φυσικά μηχανήματα, που επιτρέπει την εξυπηρέτηση εφαρμογών. Η βελτίωση της απόδοσης των εφαρμογών που παρατηρήθηκε όμως, δεν αποτελεί μονοσήμαντο συμβάν του συγκεκριμένου φυσικού server, καθώς οι αλγόριθμοι είναι υλοποιημένοι με τέτοιο τρόπο, ώστε να επηρεάζουν την συμπεριφορά του Kubernetes Scheduler ασχέτως του συστήματος που βρίσκονται. Συνεπώς, η παραπάνω τεχνική κατανομής εφαρμογών, μπορεί να βελτιώσει αισθητά την αντιστοίχιση εφαρμογών σε οποιοδήποτε server, εξοικονομώντας υπολογιστικούς πόρους στο μέγιστο δυνατό.

Λέξεις Κλειδιά

Κατανομή εφαρμογών, Resource Management, Kubernetes Extending, Scheduler, Plugin, Docker, Cluster, CPU sockets, CPU pinning, Containers, Νεφοϋπολογιστικό σύστημα, Interference Aware, Αλγόριθμοι, Virtualization, Servers, Διαθεσιμότητα

Abstract

The purpose of this thesis was the development of algorithmic concepts, in order to optimize the process of distributing applications to cloud-based environments. Considering that the usage of cloud computing has been increasing steadily over time, consequently has increased the demand of more computing resources, in order to provide services, such as hosting applications. Nonetheless, providing computing resources is a significant cost and it is physically impossible to have unlimited assets. Therefore, it is imperative to implement algorithmic techniques, which distribute applications to cloud computing services in such manner, so that on one hand, we save the disposal of resources as much as possible and on the other hand, we maintain hosting applications at the desired level.

In order to distribute applications to computing systems, an increasingly famous orchestration system is used, known as Kubernetes. Kubernetes makes use of containers to manage and distribute applications. Moreover, one of Kubernetes' main components is its Scheduler, which dictates when and how applications will be distributed, when arriving at the cluster. At that exact point, we aim at adding algorithmic techniques, which will affect the Scheduler's behavior and determine the assignment of applications to server sockets. As a result, pairs of applications are created, with the intension of conserving as many resources as possible, whilst also maintaining the availability of our applications.

Regarding evaluating the above mentioned, a physical server was set up, so that to allow application hosting. The improvement observed on application performance, is not dedicated to this specific physical server, since the algorithms are constructed in such way, so that they alter the behavior of the Kubernetes Scheduler, in spite of the system they are operating on. In conclusion, the aforementioned technique of distributing applications, can improve noticeably the workload share on any server, preserving computing resources as much as possible.

Keywords

Application distribution, Resource Management, Kubernetes Extending, Scheduler, Plugin, Docker, Cluster, CPU sockets, CPU pinning, Cloud Computing, Interference Aware, Algorithms, Virtualization, Servers, Availability

Table of Contents

Περιεχόμενα

1. Εισαγωγή 10

2. Κυβερνήτης (Kubernetes) και Εικονικοποίηση (Virtualization) 11

3. Επέκταση του Κυβερνήτη και Υλοποίηση Αλγορίθμων 13

3.1. Ο Δρομολογητής του Κυβερνήτη 13

3.2. Επέκταση του Δρομολογητή 14

3.3. Αλγοριθμικές Τεχνικές 15

4. Συγκριτική αξιολόγηση και Περιβάλλον Εκτέλεσης 17

4.1. Χρήση εφαρμογών ορόσημων 17

4.2. Δημιουργία Περιβάλλοντος Εκτέλεσης 18

5. Αποτελέσματα και Αξιολόγηση 18

5.1. Αποτελέσματα μετρήσεων 19

6. Σύνοψη και Μελλοντική Δουλειά 21

6.1. Σύνοψη 21

6.2. Μελλοντική δουλειά 22

1. Introduction 23

1.1. Thesis Purpose 23

2. Theoretical Background 24

2.1. The Issue 24

2.2. Setup 25

2.2.1. Virtualization 26

2.2.2. Resource Management 27

2.2.3. CPU Socket Architecture 29

2.2.4. Application Characteristics 29

2.2.5. Scheduling Applications 31

2.2.6. Interference Awareness 31

2.3. Kubernetes 32

2.3.1. Understanding Kubernetes 32

2.3.2. Docker 34
2.3.3. Kubernetes components 35
2.3.4. Pods 38
2.3.4.1. Deployments 39
2.3.4.2. ReplicaSets 39
2.3.4.3. DaemonSets 40
2.3.5. Nodes 40
2.3.6. Cluster Architecture 41
2.3.7. Scheduler 41
2.4. Benchmarking 42
2.4.1. SPEC Benchmarks 42
2.4.1.1. Runtimes 43
2.4.1.2. Containerization 44
2.4.1.3. Categorization 45
3. Solution Presentation 45
3.1. Environment Used 45
3.1.1. System Setup 45
3.1.2. Cluster Setup 46
3.2. Kubernetes framework 46
3.2.1. Scheduling framework 47
3.2.1.1. Scheduling & Binding Cycle 47
3.2.1.2. Extension Points 48
3.2.1.3. Plugin API 50
3.2.1.4. Plugin Configuration 51
3.2.2. Pods template 53
3.3. Application Classification 53
3.3.1. Predictors 54
3.3.2. Classification Process 54
3.4. Algorithms 55
3.4.1. Interference impact 56
3.4.2. Algorithmic Approach 56
3.4.3. CPU Pinning 57
3.4.4. Greedy 58
3.4.4.1. Concept 58

3.4.4.2. Plugin 59
3.4.4.3. Scheduling Process 59
3.4.5. Sparing 60
3.4.5.1. Concept 60
3.4.5.2. Plugin 61
3.4.5.3. Scheduling Process 62
3.4.6. Socket Based 62
3.4.6.1. Concept 62
3.4.6.2. Plugin 64
3.4.6.3. Scheduling Process 64
4. Solution Assessment 65
4.1. Evaluation Process 65
4.1.1. Service Level Objectives 66
4.1.2. Runtime Violations 74
4.1.3. Benchmark Runtimes 66
4.1.4. Workloads 67
4.2. Algorithmic Assessment 67
4.2.1. First Service Level Objective 68
4.2.2. Second Service Level Objective 69
4.2.3. Third Service Level Objective 70
4.2.4. CPU pinning Process 70
4.2.5. Proposed Solution 75
5. Conclusion 75
6. Related Work 75
6.1. Intel CPU Manager 75
6.2. Workload Classification 76
6.3. Interference Aware Managers 77
7. Future Work 77
8. Bibliography 79
9. Citation 80

1. Εισαγωγή

Επιχειρήσεις και οργανισμοί σήμερα προσπαθούν όλο και περισσότερο να εικονικοποιήσουν τα συστήματα τους ώστε να βελτιώσουν την εμπειρία του χρήστη κατά την χρήση των εφαρμογών τους. Στο παρελθόν που χρησιμοποιούνταν κατανεμημένα συστήματα παρατηρήθηκαν αρκετά προβλήματα όπως αστάθεια των ίδιων των συστημάτων, απώλεια δικτύου και εύκολη διάσπαση της ασφάλειας τους. Ως αποτέλεσμα έχουν αναπτυχθεί νέες τεχνικές με τις οποίες δημιουργούνται καλά οργανωμένα εικονικά περιβάλλοντα για την εξυπηρέτηση εφαρμογών. Τα περιβάλλοντα αυτά εξυπηρετούνται σε διακομιστές και τα δεδομένα τους αποθηκεύονται σε data-centers, παρέχοντας έτσι ισχυρή επεκτασιμότητα στις εφαρμογές με αντάλλαγμα ένα λειτουργικό κόστος. Συνεπώς πλέον οι χρήστες χρησιμοποιούν τις εφαρμογές που θέλουν από κάθε τοποθεσία χωρίς να περιορίζονται από τις δυνατότητες του μηχανήματος τους.

Αυτή η νέα πραγματικότητα έχει υλοποιηθεί χάρις την ανάπτυξη των εικονικών περιβαλλόντων. Ένας βασικός τύπος εικονικοποίησης είναι η μετατροπή των εφαρμογών σε containers (containerization), ο οποίος είναι επίσης γνωστός ως εικονικοποίηση λειτουργικού συστήματος και αφορά την δυνατότητα ενός λειτουργικού συστήματος να επιτρέπει την δημιουργία πολλαπλών απομονωμένων περιβαλλόντων χρήστη. Μέσω αυτής της εικονικοποίησης το λογισμικό μπορεί να παρέχεται σε μορφή πακέτων (containers). Αυτά τα πακέτα επιτρέπουν στις εφαρμογές να τρέξουν παντού προσφέροντας έτσι μεγάλη ευκινησία στην ανάπτυξη και εγκατάσταση τους. Αυτές οι νέες τεχνικές εξυπηρέτησης εφαρμογών έχουν βελτιώσει σημαντικά την εμπειρία του χρήστη καθώς η εικονικοποίηση των υπολογιστικών συστημάτων έχει αυξήσει τη διαθεσιμότητα των πόρων στις εφαρμογές σημαντικά. Όμως παρά αυτή τη βελτίωση στην διαθεσιμότητα των εφαρμογών, η ανάγκη εξυπηρέτησης των ίδιων και των δεδομένων τους, έχει προκαλέσει αύξηση στα κόστη συντήρησης των διακομιστών και των data-centers τους. Συγκεκριμένα, εφαρμογές που μοιράζονται πόρους κατά την εκτέλεση τους, παρατηρούνται πως αντιμετωπίζουν προβλήματα απόδοσης παρά την ποσοτική ικανοποίηση των υπολογιστικών αναγκών τους.

Ως αποτέλεσμα, για την ελαχιστοποίηση των κοστών αυτών έχουν αναπτυχθεί διάφορες τεχνικές διαχείρισης πόρων. Όμως σε ένα υπάρχον γνωστό πρόγραμμα ενορχήστρωσης εφαρμογών με το όνομα Kubernetes, η διαχείριση των υπολογιστικών πόρων βρίσκεται σε πρώιμο και ανεπαρκές στάδιο. Το Kubernetes αποτελεί έναν εξαιρετικό υποψήφιο για την αυτοματοποίηση την εγκατάστασης εφαρμογών, την διαχείριση τους και για την διαθεσιμότητα τους προς τον χρήστη. Ως αποτέλεσμα ήταν αναμενόμενη η χρήση του στην διαχείριση πακεταρισμένων εφαρμογών στα πλαίσια της εικονικοποίησης. Σκοπός μας είναι να επεκτείνουμε τον Kubernetes, ώστε να μπορεί να αντιλαμβάνεται ποιες εφαρμογές προκαλούν προβλήματα απόδοσης μεταξύ τους, και μέσω αλγοριθμικών τεχνικών, να τις ομαδοποιήσουμε κατάλληλα, μειώνοντας έτσι τα κόστη συντήρησης του και βελτιώνοντας της απόδοση τους.

2. Κυβερνήτης και Εικονικοποίηση

Προτού ξεκινήσουμε την περιγραφή της σημασίας επίγνωσης του υπολογιστικού συστήματος για παρεμβολές μεταξύ εφαρμογών σε ένα υπολογιστικό νέφος, είναι απαραίτητο να εξηγηθεί αναλυτικά η εικονικοποίηση. Τα υπολογιστικά νέφη έχουν επιτρέψει την αποθήκευση και προσπέλαση δεδομένων μέσω του Ίντερνετ, χωρίς πλέον να χρειάζεται η αποθήκευση τους στον σκληρό δίσκο του χρήστη. Ως αποτέλεσμα, οι χρήστες μπορούν να χρησιμοποιήσουν εφαρμογές μέσω διαδικτυακών συνδέσεων, χωρίς να χρειάζεται να εγκαταστήσουν κάτι στον προσωπικό τους υπολογιστή. Έτσι, οι εφαρμογές πλέον, στεγάζονται σε εξυπηρετητή με συγκεκριμένες συγκεντρωτικές αρχιτεκτονικές και οι χρήστες μπορούν να τις χρησιμοποιήσουν μέσω της κατάλληλης διεπαφής και της ανάλογης διαδικτυακής σύνδεσης. Συνεπώς, εικονικοποίηση έχουμε όταν η ψηφιακή μορφή μιας εφαρμογής δημιουργείται προς χρήση αντί για την ίδια την εφαρμογή καθαυτή. Οπότε μας είναι κατανοητό πως το υπολογιστικό νέφος βασίζεται στην εικονικοποίηση για να κάνει την χρήση των εφαρμογών πιο εύκολη και αρεστή στον χρήστη.

Η εικονικοποίηση στο υπολογιστικό νέφος αναφέρεται στη δημιουργία εικονικών υπολογιστικών μηχανημάτων, λογισμικών και λειτουργικών συστημάτων, αποτρέποντας έτσι την ανάγκη για ξεχωριστή εγκατάσταση λογισμικού σε κάθε φυσική μηχανή που πρόκειται να το χρησιμοποιήσει. Όμως η εικονικοποίηση συναντάται σε πολλές μορφές, για να μπορέσει κανείς έτσι να ικανοποιήσει τις διαφορετικές ανάγκες στην διαχείριση εφαρμογών και χρήση υπολογιστικών πόρων. Οι τρεις βασικότεροι τύποι εικονικοποίησης είναι γνωστοί ως, Εικονικοποίηση Διακομιστή, Εικονικοποίηση Διεπαφής και Εικονικοποίηση Αποθηκευτικού χώρου. Ο πιο γνωστός τύπος εικονικοποίησης που επηρεάζει το υπολογιστικό νέφος το περισσότερο είναι η εικονικοποίηση διακομιστή. Συγκεκριμένα, η εικονικοποίηση διακομιστή επιτρέπει στους παρόχους να βελτιστοποιούν την χρήση των μηχανημάτων τους και να επιτρέπουν στις εφαρμογές τους να τρέχουν με καλύτερη απόδοση. Αυτό καθίσταται εφικτό, καθώς η εικονικοποίηση αυτή επιτρέπει τον διαχωρισμό και την απομόνωση των μηχανημάτων του υπολογιστή, από το κάθε πρόγραμμα που μπορεί να τρέξει σε αυτόν. Οι φυσικοί πόροι του υπολογιστή εκφράζονται έτσι με λογικές αντιπαραστάσεις, όπως για παράδειγμα ο επεξεργαστής ενός υπολογιστή (CPU δηλαδή) , ο οποίος πλέον εκφράζεται ως vCPU, δηλαδή ψηφιακός επεξεργαστής. Αυτή η λογική απομόνωση των πόρων ενός υπολογιστή, επιτρέπει την δημιουργία εικονικών μηχανημάτων (VMs) στον ίδιο τον υπολογιστή, επιτρέποντας την καλύτερη κατανομή των εφαρμογών ανάλογα με τις ανάγκες τους.

Πέρα από τις εικονικές μηχανές, με την εικονικοποίηση έχουμε και την ανάπτυξη των containers, αλλιώς στα ελληνικά περιέκτες, μέσω των οποίων μπορούν να στηριχθούν και να αναπτυχθούν εφαρμογές όπως εφαρμογές διαδικτύου. Τα containers αυτά είναι φορητά μεταξύ μηχανημάτων, αξιόπιστα και κλιμακούμενα επιτρέποντας την εύκολη ανάπτυξη και διαχείριση των εφαρμογών τους. Το Kubernetes ή αλλιώς στα ελληνικά Κυβερνήτης, επιτρέπει την διαχείριση τέτοιων κιβωτίων - containers και χρησιμοποιείται ευρέως για την διαχείριση των εφαρμογών τους. Συγκεκριμένα, το Kubernetes είναι μια φορητή,

επεκτάσιμή πλατφόρμα ανοιχτού λογισμικού που διαχειρίζεται πακεταρισμένες εφαρμογές και υπηρεσίες ελέγχοντας την παραμετροποίηση και την αυτοματοποίηση τους. Ως αποτέλεσμα, έχει ένα μεγάλο και αναπτυσσόμενο οικοσύστημα το οποίο μας επιτρέπει να τρέξουμε κατανεμημένα συστήματα με άνεση και ελαστικότητα. Έτσι, το Kubernetes επιτρέπει στους πάροχους υπηρεσιών να χρησιμοποιούν αποδοτικά τους φυσικούς τους πόρους ενώ παράλληλα διατηρούν το επίπεδο απόδοσης των εφαρμογών που αναζητά ο πελάτης. Το Kubernetes διαχειρίζεται τις εφαρμογές, αυτοματοποιεί την ανάπτυξη λογισμικού και προσφέρει μεγαλύτερη ασφάλεια.

Έχοντας πλέον κατανοήσει την χρησιμότητα του Kubernetes, είναι σημαντικό να εξηγήσουμε και την λειτουργία του και την αρχιτεκτονική του. Αρχικά, όταν εγκαθιστούμε το Kubernetes σε ένα σύστημα έχουμε μία συστάδα γνωστή και ως cluster. Η συστάδα του Kubernetes αποτελείται από ένα πλήθος μηχανών-εργατών, οι οποίες ονομάζονται Κόμβοι (Nodes) και τρέχουν τις πακεταρισμένες εφαρμογές. Το σύνολο ενός ή περισσότερων τέτοιων περικετών με κοινόχρηστο δίκτυο και χώρο αποθήκευσης ονομάζεται Pod. Το Pod ή στα ελληνικά Κάψουλα, είναι το μικρότερο αναπτύξιμο κομμάτι λογισμικού στο Kubernetes και διαχειρίζεται ένα η περισσότερα κοντέινερ εφαρμογών. Οι εφαρμογές αυτές τρέχουν πάντα πάνω στους Κόμβους, δηλαδή στις μηχανές-εργάτες. Όμως μία (ή και περισσότερες αν θέλουμε να έχουμε υψηλή διαθεσιμότητα στο σύστημα μας) μηχανή έχει τον ρόλο του κόμβου-αφέντη, η οποία διαχειρίζεται τη συστάδα του Kubernetes και τις λειτουργίες του. Στον κόμβο αυτόν βρίσκεται το επίπεδο ελέγχου του Kubernetes, το οποίο θα πάρει τις αποφάσεις απαραίτητες για την λειτουργία της συστάδα, καθώς και για τυχόν μεταβολές και συμβάντα στο σύστημα. Οι αποφάσεις και οι ενέργειες αυτές λαμβάνονται από διάφορα υπο-μέρη του επιπέδου ελέγχου, όπως το **Kube-apiserver**, το οποίο είναι υπεύθυνο να επιβεβαιώνει και να επαληθεύει τα δεδομένα για άλλα αντικείμενα όπως για παράδειγμα τα Pods που αναφέρθηκαν προηγουμένως. Επιπλέον, υπάρχει και το **Etcd**, το οποίο είναι ένας συνεπής χώρος αποθήκευσης υψηλής διαθεσιμότητας για να αποθηκεύει όλα τα δεδομένα της συστάδας μας. Ακόμη, υπάρχει και ο **Controller-Manager**, ο οποίος τρέχει όλες τις μεθόδους ελέγχου για την συστάδα. Συγκεκριμένα , έχει πολλαπλούς ελεγκτές οι οποίοι είναι υπεύθυνοι για τα περισσότερα κομμάτια της συστάδας όπως για παράδειγμα, τους Κόμβους, τα Pods, τις υπηρεσίες που εκτελούνται κλπ. Τέλος, το εξάρτημα του Kubernetes που θα μας απασχολήσει περισσότερο είναι ο **Scheduler**, ή αλλιώς ο Δρομολογητής. Ο Δρομολογητής παρακολουθεί συνεχώς για νέα Pods, τα οποία δεν έχουν ακόμη δρομολογηθεί προς κανέναν Κόμβο, ώστε να εξυπηρετηθούν. Έτσι, ο Δρομολογητής θα βαθμολογήσει κάθε διαθέσιμο κόμβο ως προς το νέο Pod και με αυτόν τον τρόπο θα επιλέξει τον καλύτερο ως προς την εφαρμογή.

3. Επέκταση του Κυβερνήτη και Υλοποίηση Αλγορίθμων

Όπως έγινε κατανοητό, η διαδικασία λήψης αποφάσεων προς την δρομολόγηση και κατανομή των εφαρμογών σε ένα σύστημα του Kubernetes γίνεται από τον Δρομολογητή. Συγκεκριμένα, το επίπεδο λήψης αποφάσεων για την δρομολόγηση μιας εφαρμογής εντός ενός Pod, προς έναν συγκεκριμένο Κόμβο και άρα επακολούθως σε ένα συγκεκριμένο φυσικό μηχάνημα, γίνεται στον Δρομολογητή. Συνεπώς, το ερώτημα ποιοι υπολογιστικοί πόροι θα καλύψουν τις ανάγκες της εκάστοτε εφαρμογής, απαντάται από τον Δρομολογητή. Η συνειδητοποίηση αυτή μας επιτρέπει να εξετάσουμε τον τρόπο με τον οποίο μπορούμε να επηρεάσουμε την συμπεριφορά του Δρομολογητή και να καθορίσουμε τον τρόπο που θα κατανεμηθούν οι εφαρμογές. Η σωστή επιλογή των κόμβων για τις εφαρμογές που τρέχουν στο σύστημα μας, θα αποτελέσει σημαντικό εργαλείο προς την εξοικονόμηση πόρων και την ικανότητα του Kubernetes να διατηρεί τις εφαρμογές του σε υψηλή απόδοση.

3.1 Ο Δρομολογητής του Κυβερνήτη

Ο Δρομολογητής, πριν αποφασίσει ποιος κόμβος ταιριάζει περισσότερο για κάθε Pod που καταφθάνει στο σύστημα μας, λαμβάνει υπόψιν του αρκετούς παράγοντες όπως οι απαιτήσεις υπολογιστικών πόρων, προδιαγραφές συγγένειας, περιορισμοί πολιτικής ή λογισμικού και άλλα. Για να επηρεάσουμε την διαδικασία αυτή, προσθέσαμε δικούς μας παράγοντες προς την επιλογή του κατάλληλου κόμβου, επεκτείνοντας έτσι τον Δρομολογητή με την χρήση πρόσθετων λειτουργιών (plugins). Πλέον ο βασικότερος παράγοντας για την επιλογή του κατάλληλου κόμβου θα είναι ο δικός μας, ο οποίος προσδίδει στο Kubernetes επίγνωση παρεμβολών μεταξύ εφαρμογών.

Η βελτιστοποίηση χρήσης των υπολογιστικών πόρων από τον Δρομολογητή αρκούταν απλά στον έλεγχο να ικανοποιούνται οι τεχνικές απαιτήσεις των εφαρμογών και εφόσον αυτές πράγματι καλύπτονταν, η δρομολόγηση λάμβανε μέρος. Όμως, όπως έχει παρατηρηθεί, εφαρμογές που φαινομενικά κάλυπταν με ευκολία τις απαιτήσεις τους αυτές, εμφάνιζαν σοβαρά προβλήματα απόδοσης. Αυτό οφείλεται στις παρεμβολές που μπορεί να έχουν συγκεκριμένες εφαρμογές με άλλες όταν μοιράζονται κοινούς υπολογιστικούς πόρους. Πλέον σκοπός μας είναι η επέκταση του δρομολογητή με τη χρήση των plugins, ώστε να αντιλαμβάνεται ποιες εφαρμογές προκαλούν παρεμβολές σε άλλες και να τις κατανέμει αναλόγως.

3.2 Επέκταση του Δρομολογητή

Η επέκταση του Δρομολογητή ώστε να του προσδώσουμε την συμπεριφορά που επιθυμούμε γίνεται με τη βοήθεια του Πλαισίου Δρομολόγησης ή αλλιώς Scheduling Framework. Η συμπεριφορά του Δρομολογητή μπορεί να αλλάξει εφόσον γράψει κανείς ένα ειδικό αρχείο παραμετροποίησης του Δρομολογητή με το οποίο θα επηρεάζει τις διάφορες φάσεις δρομολόγησης ενός Pod. Κάθε φάση εκτίθεται μέσω ενός σημείου επέκτασης γνωστό και ως extension point.

Όλα τα παραπάνω γίνονται εφικτά με τη χρήση του πλαισίου δρομολόγησης. Το πλαίσιο δρομολόγησης είναι μια βυσματούμενη αρχιτεκτονική για τον Δρομολογητή του Kubernetes. Προσθέτει μια νέα ομάδα από επεκτάσεις στον υπάρχοντα Δρομολογητή. Οι επεκτάσεις αυτές μεταγλωττίζονται στη συνέχεια πάνω στον δρομολογητή και επηρεάζουν έτσι αναλόγως την συμπεριφορά του, ενώ παράλληλα μας επιτρέπουν να διατηρούμε τον Δρομολογητή εύκολα διατηρήσιμο και ελαφρύ.

Κάθε διαδικασία δρομολόγησης ενός Pod, χωρίζεται σε δύο φάσεις. Η πρώτη φάση λέγεται φάση δρομολόγησης (scheduling cycle) και η δεύτερη φράση λέγεται φάση δέσμευσης (binding cycle). Κατά την φάση δρομολόγησης, υπάρχουν συγκεκριμένα σημεία επέκτασης του Kubernetes τα οποία ελέγχουν τον τρόπο με τον οποία ταξινομεί ο Δρομολογητής τα αφικνούμενα Pods, φιλτράρει τους κόμβους έτσι ώστε να μείνουν μόνο εφικτοί κόμβοι και στη συνέχεια τους βαθμολογεί ώστε να βρει το καλύτερο ζευγάρι για την εφαρμογή μας. Μετά την επιλογή του καταλληλότερου κόμβου, ξεκινά η φάση δέσμευσης. Κατά αυτή την φάση, μια διαφορετική ομάδα από σημεία επέκτασης ξεκινάει, που επηρεάζουν τη συμπεριφορά δρομολόγησης πριν της δέσμευση της εφαρμογής στον Κόμβο, κατά την διάρκεια της δέσμευσης και μετά το πέρας της. Με αυτόν τον τρόπο ολοκληρώνεται και η ανάπτυξη της εφαρμογής στον στοχευμένο κόμβο.

Η παρούσα βιβλιοθήκη επέκτασης του δρομολογητή προσφέρει πολλά σημεία επέκτασης, εμείς όμως θα σταθούμε σε αυτά που ήταν απαραίτητα για να προσδώσουμε στον δρομολογητή τη συμπεριφορά που θέλουμε. Αρχικά, πρώτο σημείο επέκτασης που εκτίθεται από το πλαίσιο δρομολόγησης είναι το Φίλτρο (**Filter**).

Το Filter Plugin χρησιμοποιείται για να αποκλειστούν κόμβοι που δεν μπορούν να τρέξουν την εφαρμογή. Αν κάποιο Filter plugin σημειώσει έναν κόμβο ως ακατάλληλο, τότε και όλα τα υπόλοιπα plugins του ίδιου είδους, θα τον παρακάμψουν.

Στη συνέχεια αφού πλέον έχουν ελεγχθεί οι κόμβοι και είναι διαθέσιμοι όσοι μόνο ταιριάζουν στην εφαρμογή μας, καλούνται plugins τα οποία αφορούν την βαθμολόγηση των εναπομεινάντων κόμβων. Αυτό το σημείο επέκτασης ονομάζεται σημείο επέκτασης Βαθμολόγησης ή αλλιώς **Scoring** extension point. Plugins που χρησιμοποιούνε το σημείο αυτό, έχουν δύο φάσεις. Κατά την πρώτη φάση, που ονομάζεται και φάση βαθμολόγησης οι διαθέσιμοι κόμβοι που περάσαν την φάση φιλτραρίσματος προηγουμένως, βαθμολογούνται ανάλογα με τα χαρακτηριστικά τους. Στην δεύτερη φάση και αφού έχουν βαθμολογηθεί όλοι οι κόμβοι έχουμε την ομαλοποίηση βαθμολογίας ή αλλιώς "scoring normalization". Κατά τη διάρκεια της φάσης αυτής, τροποποιούνται οι

βαθμολογίας του κάθε κόμβου, προτού ο Δρομολογητής παρουσιάσει την τελική βαθμολογία των κόμβων. Η φάση αυτή μας επιτρέπει να διορθώσουμε την βαθμολογία των κόμβων και να επαληθεύσουμε πως είναι πράγματι σε ένα λογικά ορισμένο πεδίο τιμών.

3.3 Αλγοριθμικές Τεχνικές

Όταν δρομολογούμε εφαρμογές στους κόμβους , ο δρομολογητής του Κυβερνήτη θα τρέξει κάθε plugin στη δηλωμένη σειρά τους έτσι ώστε να αποφασίσει και να επιλέξει τον καλύτερο κόμβο για την εφαρμογή που κατέφθασε στον σέρβερ. Κάθε αλγοριθμική τεχνική που παρουσιάζεται σε αυτή τη διπλωματική εργασία έχει γνώμονα αφενός το φιλτράρισμα των κόμβων ώστε να έχουμε ισόποση κατανομή των εφαρμογών στους διαθέσιμους κόμβους και αφετέρου την βαθμολόγηση τους σύμφωνα με το υπάρχον φορτίο που έχουν. Συγκεκριμένα κατά την φάση της βαθμολόγησης των κόμβων, ο δρομολογητής λαμβάνει το φορτίο εφαρμογών του κάθε κόμβου. Στη συνέχεια, σύμφωνα με την εκάστοτε αλγοριθμική τεχνική, βαθμολογεί τον κάθε κόμβο σύμφωνα πάντα με την πιθανή προσθήκη της νέας αφικνούμενης εφαρμογής. Η βαθμολογία του κόμβου δηλαδή, υπολογίζεται στην περίπτωση που θα προσθέταμε την νέα εφαρμογή.

Η βαθμολόγηση των κόμβων βασίζεται στον υπολογισμό της αναμενόμενης καθυστέρησης που θα προκληθεί από την τοποθέτηση δύο ή περισσότερων εφαρμογών μαζί στον ίδιο κόμβο. Έτσι όσο χειρότερη είναι η καθυστέρηση τόσο υψηλότερη θα είναι και η βαθμολογία του κόμβου. Αυτό διορθώνεται με αντιστροφή της βαθμολογίας κατά την φάση Ομαλοποίησης Βαθμολογίας. Συνεπώς, όσο μικρότερη καθυστέρηση έχουμε κατά την τοποθέτηση εφαρμογών μαζί, τόσο καλύτερη επιλογή αποτελεί ο κόμβος αυτός.

Οι αλγοριθμικές τεχνικές εφαρμόζονται κατά την διάρκεια της Βαθμολόγησης των κόμβων, καθώς η φάση Φιλτραρίσματος είναι κοινή ανεξαρτήτως τεχνικής ώστε να έχουμε ισόποση κατανομή των εφαρμογών στα μέρη του συστήματος. Συγκεκριμένα, οι τρεις διαφορετικές τεχνικές που εφαρμόσαμε είναι οι εξής.

Πρώτη αλγοριθμική τεχνική που εξετάστηκε είναι ο άπληστος αλγόριθμος ή αλλιώς Greedy Algorithm, ο οποίος προσπαθεί να τοποθετήσει μαζί όσο το δυνατόν περισσότερες εφαρμογές, που προκαλούν λίγες παρεμβολές στην απόδοση των συν-τοποθετημένων εφαρμογών. Η δεύτερη αλγοριθμική τεχνική προσπαθεί να δράσει με τον ακριβώς αντίστροφο τρόπο. Ονομάζεται χαριστικός αλγόριθμος ή αλλιώς Sparing algorithm, καθώς προσπαθεί να χαρίσει την επίδοση των περισσότερων εφαρμογών, τοποθετώντας μαζί εφαρμογές με υψηλή διάθεση να προκαλέσουν παρεμβολές σε συν-τοποθετημένες εφαρμογές. Κατ' αυτόν τον τρόπο «θυσιάζοντας» εφαρμογές που θα δημιουργούσαν προβλήματα σε άλλες, επιτρέποντας στις υπόλοιπες να διατηρήσουν την απόδοση τους σε αποδεκτά επίπεδα. Τέλος, δοκιμάστηκε και ένας πιο σύνθετος αλγόριθμος, βασισμένος στην τοπολογία των κόμβων και ονομάστηκε Socket-based Algorithm. Συγκεκριμένα, αυτός ο αλγόριθμος, προσπαθεί πιο έξυπνα να τοποθετήσει εφαρμογές μαζί, συνδυάζοντας τις ανθεκτικές με τις επιθετικές και τις ευαίσθητες με πιο ήρεμες. Κατ' αυτόν τον τρόπο τα

ζεύγη εφαρμογών που μοιράζονται υπολογιστικούς πόρους με τέτοιο τρόπο ώστε να εξοικονομούνται στο έπακρο.

3.4 Χαρακτηρισμός εφαρμογών

Όπως έγινε εύκολα αντιληπτό, η ικανότητα των αλγοριθμικών τεχνικών να δρουν και να προσφέρουν στον Κυβερνήτη την επίγνωση παρεμβολών στις αποδόσεις των εφαρμογών, βασίζεται αρχικά στην επίγνωση των χαρακτηριστικών των εφαρμογών αυτών. Κάθε εφαρμογή που δραστηριοποιείται σε ένα φυσικό μηχάνημα δεν σημαίνει πως έχει και την ίδια συμπεριφορά με τις υπόλοιπες. Αντιθέτως, έχει παρατηρηθεί πως η κάθε εφαρμογή δρα διαφορετικά όταν συν-τοποθετείται με άλλες και υπάρχουν συνεπώς, διαφορετικοί τύποι εφαρμογών με διαφορετικούς βαθμούς επίδρασης και επιθετικότητας. Τα δύο βασικά χαρακτηριστικά κάθε εφαρμογής που καθορίζουν τη συμπεριφορά της, ορίζονται ως η επιθετικότητα της εφαρμογής και η ευαισθησία της.

Η επιθετικότητα μιας εφαρμογής καθορίζει την ικανότητα της να επηρεάζει την απόδοση άλλων εφαρμογών όταν μοιράζονται κοινούς πόρους. Συγκεκριμένα, τέτοιες εφαρμογές προκαλούν την μείωση της απόδοσης των περισσότερων εφαρμογών που θα τοποθετηθούν μαζί τους. Όμως μια επιθετική εφαρμογή δε θα το προκαλέσει σε όλες τις εφαρμογές που θα τοποθετηθούν μαζί της. Αυτό συμβαίνει επειδή υπάρχει και ένα δεύτερο σημαντικό χαρακτηριστικό των εφαρμογών ως προς τις παρεμβολές μεταξύ τους. Αυτό το χαρακτηριστικό γνωστό ως ευαισθησία, καθορίζει αν και κατά πόσο μια εφαρμογή δύναται να παρουσιάσει σημαντική μείωση απόδοσης μετά την τοποθέτηση της με άλλες εφαρμογές. Αυτά τα δύο χαρακτηριστικά συνεπώς, δημιουργούν τέσσερις κλάσεις εφαρμογών. Η πρώτη κλάση, που κατά την διάρκεια των πειραματικών μετρήσεων ονομάζεται κλάση Α, είναι η τάξη εφαρμογών που δεν είναι επιθετικές και ούτε ευαίσθητες. Τέτοιες, εφαρμογές θεωρούνται ιδανικές και η καλύτερη επιλογή για τοποθέτηση με οποιονδήποτε άλλο τύπο εφαρμογής. Στη συνέχεια, οι εφαρμογές κλάσης Β, είναι εφαρμογές που επίσης δεν είναι επιθετικές, όμως τώρα εμφανίζουν αυξημένη ευαισθησία. Εφαρμογές τέτοιου τύπου έχουν το προτέρημα να μην προκαλούν παρεμβολές στην απόδοση άλλων εφαρμογών, όμως όντας ευαίσθητες αντιμετωπίζουν προβλήματα κατά την τοποθέτηση τους με άλλες επιθετικές εφαρμογές. Από την άλλη μεριά, εφαρμογές που είναι μεν επιθετικές αλλά δεν είναι ευαίσθητες ανήκουν στην κατηγορία Γ. Τέλος, υπάρχουν και εφαρμογές οι οποίες είναι και επιθετικές προκαλώντας μείωση στις αποδόσεις άλλων συν-τοποθετημένων εφαρμογών, αλλά και ευαίσθητες όντας έτσι ευάλωτες προς άλλες παρεμβολές. Αυτές οι εφαρμογές αποτελούν την κατηγορία Δ και χρίζουν σημαντικής προσοχής όταν τοποθετούνται σε ένα σύστημα.

4. Συγκριτική Αξιολόγηση και Περιβάλλον Εκτέλεσης

Σύμφωνα με τα παραπάνω, η επέκταση του Κυβερνήτη βασίζεται αφενός στην επέκταση του ίδιου του Δρομολογητή του, σύμφωνα με την εκάστοτε αλγοριθμική τεχνική και αφετέρου στην αναγνώριση και τον χαρακτηρισμό των εφαρμογών που δραστηριοποιούνται στο σύστημα του. Η πειραματική αξιολόγηση των παραπάνω έγινε με την δημιουργία του κατάλληλου περιβάλλοντος εκτέλεσης καθώς και με την χρήση συγκεκριμένων γνωστών εφαρμογών οροσήμων, ώστε να γίνει εφικτή η μέτρηση της απόδοσης σύμφωνα με τα πειραματικά πρότυπα.

4.1 Χρήση εφαρμογών ορόσημων

Για την απόκτηση μετρήσεων και την επαλήθευση των αλγοριθμικών τεχνικών χρησιμοποιήθηκε η γνωστή βιβλιοθήκη μετροπρογραμμάτων SPEC Benchmark suite. Οι εφαρμογές- μετροπρογράμματα που χρησιμοποιήθηκαν έχουν σκοπό να προσομοιώσουν ρεαλιστικές καταστάσεις από προγράμματα εκτελέσεων Java μέχρι σύνθετες υπολογιστικές διεργασίες. Η βιβλιοθήκη SPEC CPU επιτρέπει την αξιολόγηση της υπολογιστικής ικανότητας της εφαρμογής ως προς την χρήση του επεξεργαστή. Με αυτόν τον τρόπο, δημιουργούμε σενάρια προσομοίωσης αφικνούμενων εφαρμογών και αξιολογούμε αν πράγματι οι αλγοριθμικές τεχνικές βελτιώνουν την απόδοση τους σε σύγκριση με τον απλό τυχαίο αλγόριθμο δρομολόγησης.

Για την λήψη αυτών των μετρήσεων, είναι απαραίτητο να καταμετρήσουμε τον χρόνο εκτέλεσης της εκάστοτε εφαρμογής. Αρχικά, κάθε ορόσημο εφαρμογής που θα χρησιμοποιηθεί στα διάφορα σενάρια, τρέχει απομονωμένο με πλήρη έλεγχο πάνω στους διαθέσιμους υπολογιστικούς πόρους. Έτσι έχουμε την απόδοση κάθε εφαρμογής σε ιδανικό σενάριο χωρίς παρεμβολές, και μπορούμε πλέον να υπολογίσουμε τον βαθμό μείωσης της απόδοσης στα διάφορα σενάρια προσομοίωσης. Συγκεκριμένα, κάθε σενάριο εκτέλεσης θα τοποθετήσει μαζί δύο διαφορετικές εφαρμογές στους κόμβους ώστε στη συνέχεια να έχουμε έναν πίνακα NxN (όπου N διαφορετικοί τύποι μετροπρογραμμάτων) με τις μετρήσεις εκτέλεσης της κάθε εφαρμογής.

Έχοντας πλέον τις κατάλληλες μετρήσεις μπορούμε να καταλήξουμε στην κατηγορία εφαρμογής με τη χρήση μιας σειράς από δείκτες πρόβλεψης. Ο δείκτης πρόβλεψης αυτός υπολογίζει και την ευαισθησία της εκάστοτε εφαρμογής, σύμφωνα με τις τιμές εκτέλεσης της ίδιας, αλλά και την επιθετικότητα της σύμφωνα με την μείωση απόδοσης των άλλων εφαρμογών όταν τοποθετήθηκαν μαζί της. Με αυτόν τον τρόπο, καθίσταται δυνατή η κατηγοριοποίηση της κάθε εφαρμογής, ώστε να γίνει δυνατή η διαχείριση τους από την εκάστοτε αλγοριθμική τεχνική.

4.2 Δημιουργία Περιβάλλοντος Εκτέλεσης

Για την πειραματική αξιολόγηση της θέσης της διπλωματικής, δημιουργήθηκε ένα περιβάλλον εκτέλεσης για την προσομοίωση ρεαλιστικών συνθηκών κατά την εξυπηρέτηση εφαρμογών από έναν πάροχο. Συγκεκριμένα, στήθηκε ένα υπολογιστικό σύστημα βασισμένο στην αρχιτεκτονική εικονικοποίησης, το οποίο αποτελείται από τέσσερις υπολογιστές με την ίδια υπολογιστική ικανότητα. Αυτά τα τέσσερα υπολογιστικά συστήματα αποτέλεσαν τους 4 κόμβους της συστάδας του Kubernetes.

Στη συνέχεια οι εφαρμογές ορόσημα που χρησιμοποιήθηκαν, κατηγοριοποιήθηκαν πάνω σε αυτά τα μηχανήματα ώστε να καταγραφεί η συμπεριφορά τους κάτω από την συγκεκριμένη αρχιτεκτονική και υπολογιστική ικανότητα των συστημάτων. Για την λήψη ικανοποιητικής ποσότητας μετρήσεων, υλοποιήθηκαν πολλαπλά σενάρια φόρτου εργασίας, διαφορετική έντασης κάθε φορά, ώστε να εξεταστεί ο κάθε αλγόριθμος υπό κάθε πιθανή συνθήκη πίεσης.

5. Αποτελέσματα και Αξιολόγηση

Οι αλγοριθμικές τεχνικές ως τώρα έχουν περιοριστεί στο να λαμβάνουν τις αποφάσεις κατανομής των εφαρμογών σε ένα υψηλό επίπεδο. Αυτό είναι αναμενόμενο καθώς οι αλγόριθμοι εφαρμόζονται κατά την διάρκεια της δρομολόγησης, μέσω του Πλαισίου Δρομολόγησης. Με αυτόν τον τρόπο, οι εφαρμογές κατανέμονται στους κόμβους, όμως μετά η επιλογή της ομάδας επεξεργαστών που θα χρησιμοποιήσουν για την εκτέλεση τους παραμένει τυχαία.

Για αυτόν τον λόγο, υλοποιήθηκε και η τεχνική του CPU-pinning, δηλαδή του «καρφιτσώματος» μιας εφαρμογής σε μια συγκεκριμένη ομάδα επεξεργαστών. Συνεπώς δημιουργήθηκε ένα DaemonSet του Kubernetes, δηλαδή ένας δαίμονας που τρέχει περιοδικά και καλεί την λίστα των εφαρμογών που τρέχουν στον εκάστοτε κόμβο. Με την χρήση των control groups , μπορούμε να καρφιτσώσουμε μια εφαρμογή σε έναν ή περισσότερους επεξεργαστές, δηλαδή κατ’ επέκταση σε ένα socket επεξεργαστή. Με αυτόν τον τρόπο μπορούμε να δούμε την βελτίωση που προσφέρουν οι αλγοριθμικές τεχνικές στην απόδοση των εφαρμογών είτε με το καρφίτσωμα επεξεργαστή ενεργοποιημένο είτε χωρίς αυτό.

Για την λήψη αποτελεσμάτων τα οποία θα επαληθεύουν την ικανότητα του δρομολογητή να παρουσιάζει ικανότητα επίγνωσης των παρεμβολών μεταξύ εφαρμογών, είναι απαραίτητο να εξεταστούν πολλαπλές διαφορετικές συνθήκες πίεσης του συστήματος. Για αυτόν τον λόγο, χρησιμοποιήθηκαν τρεις διαφορετικοί τύποι φόρτου εργασίας κλιμακούμενης έντασης. Ο πρώτος τύπος, γνωστός ως φόρτος εργασίας χαμηλής έντασης (low-intensity workload) αποτελείται κυρίως από εφαρμογές οι οποίες είναι χαμηλής επιθετικότητας. Συγκεκριμένα, η πλειοψηφία των εφαρμογών δεν θα προκαλούν

σημαντικά προβλήματα στην επίδοση των άλλων δημιουργώντας έτσι συνθήκες ήπιας πίεσης στο σύστημα. Στη συνέχεια εξετάστηκαν και φόρτοι εργασίας μεσαίας έντασης (mid-intensity), που αποτελούνται από ισορροπημένες κατανομές των διάφορων τύπων εφαρμογών. Συγκεκριμένα, προσπαθούν να προσομοιώσουν καταστάσεις μέσης πίεσης στο σύστημα για να εξετάσουν τις πιθανές παρεμβολές μεταξύ εφαρμογών στην πιο συνηθισμένη κατάσταση ενός υπολογιστικού συστήματος. Τέλος, ο τρίτος τύπος φόρτου εργασίας είναι γνωστός ως υψηλής έντασης (high-intensity workloads). Με αυτούς τους τύπους φόρτου εργασίας εξετάζονται οι πιο ακραίες συνθήκες πίεσης στο σύστημα, όπου πλέον συσσωρεύονται πολλές εφαρμογές υψηλής επιθετικότητας. Έτσι μας δίνεται η δυνατότητα να εξετάσουμε την ικανότητα των αλγορίθμων και σε τέτοιες δυσμενείς συνθήκες.

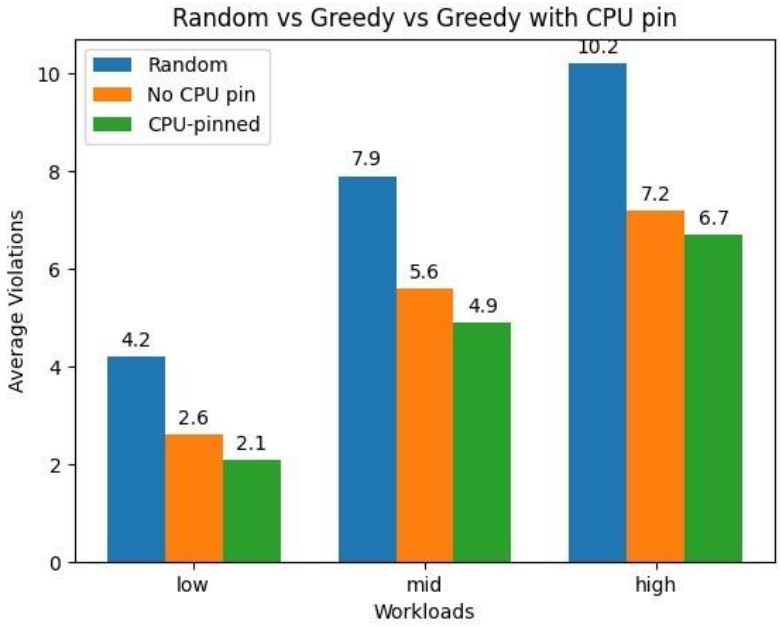
Η αποτελεσματικότητα των εφαρμογών όμως προϋποθέτει τον αναλυτικό και ξεκάθαρο ορισμό της παρεμβολής μιας εφαρμογής. Αρχικά πρέπει να οριστεί ξεκάθαρα σε ποιο σημείο μείωσης της απόδοσης μιας εφαρμογής έχουμε παρεμβολή. Αυτή η μείωση της απόδοσης της εφαρμογής, πέραν του αποδεκτού ορίου αποτελεί παράβαση της συμφωνίας μεταξύ πελάτη και παρόχου. Συγκεκριμένα, όταν ο πάροχος εξυπηρετεί τις εφαρμογές κάποιου πελάτη, συμφωνείται ένα όριο μείωσης απόδοσης το οποίο είναι αποδεκτό. Αυτό το όριο αν ξεπεραστεί, τότε έχουμε παράβαση και η απόδοση των αλγορίθμων μας βασίζεται ακριβώς πάνω σε αυτό. Πόσες παραβιάσεις έχουμε για τον εκάστοτε αλγόριθμο, σύμφωνα πάντα με το όριο που έχει συμφωνηθεί μεταξύ παρόχου και πελάτη.

5.1 Αποτελέσματα μετρήσεων

Στη συνέχεια ακολουθούν οι μετρήσεις του εκάστοτε αλγορίθμου. Για τον υπολογισμό των αποτελεσμάτων χρησιμοποιήθηκαν τρία διαφορετικά όρια παρεμβολής που θα μπορούσαν να συμφωνηθούν μεταξύ πελάτη και παρόχου. Αυτά τα όρια επισημαίνουν αντίστοιχα, πως οι εφαρμογές δεν μπορούν να παρουσιάσουν μείωση απόδοσης μεγαλύτερη της τάξης του 10%, 20% και 30%. Στην περίπτωση που παρατηρηθεί μείωση της απόδοσης μεγαλύτερη του ορίου, τότε έχουμε και μια παράβαση.

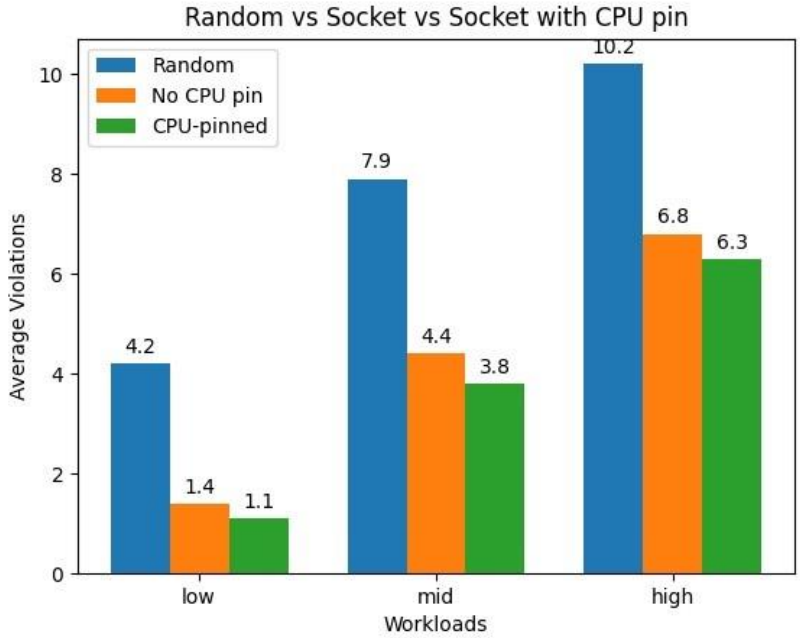
Παρακάτω ακολουθούν συγκεντρωτικά τα αποτελέσματα του κάθε αλγορίθμου για το κάθε όριο, αρχικά με τις εφαρμογές “καρφιτσωμένες” στους επεξεργαστές και στη συνέχεια χωρίς, πάντα σε σύγκριση με τον κλασσικό τυχαίο αλγόριθμο δρομολόγησης του Κυβερνήτη.

ΑΠΛΗΣΤΟΣ ΑΛΓΟΡΙΘΜΟΣ



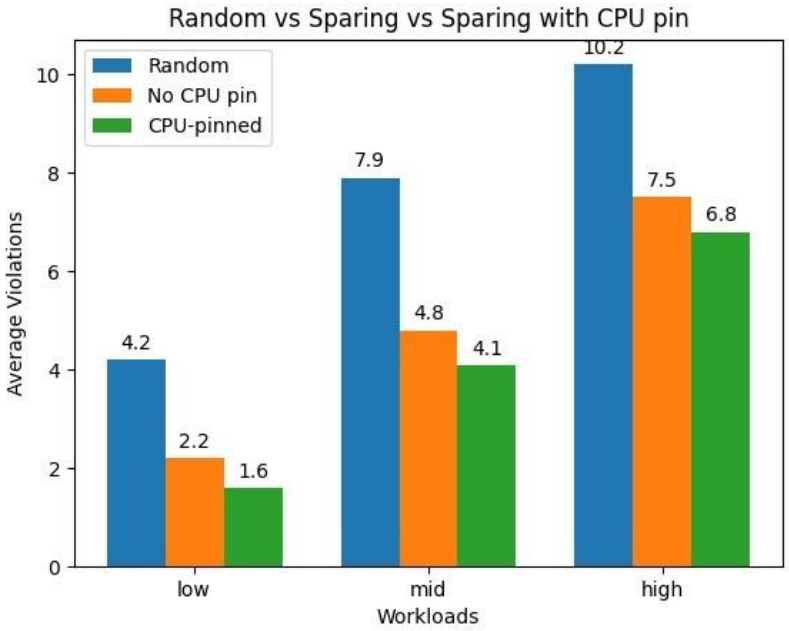
Μετρήσεις Άπληστου Αλγόριθμου

ΤΟΠΟΛΟΓΙΚΟΣ ΑΛΓΟΡΙΘΜΟΣ



Μετρήσεις Τοπολογικού Αλγόριθμου

ΧΑΡΙΣΤΙΚΟΣ ΑΛΓΟΡΙΘΜΟΣ



Μετρήσεις Χαριστικού Αλγορίθμου

Όπως εύκολα παρατηρεί κανείς, η πιο αποτελεσματική αλγοριθμική τεχνική δρομολόγησης είναι του Socket-based αλγορίθμου με ενεργοποιημένο το "κλείδωμα" των εφαρμογών στους επεξεργαστές τους. Ο αλγόριθμος συγκεκριμένα, αποφεύγει να δημιουργήσει σπάταλες τετράδες εφαρμογών που θα μπορούσαν να αξιοποιηθούν καλύτερα, όπως ακριβώς κάναν οι άλλοι δύο πιο άπληστοι αλγόριθμοι. Συγκεκριμένα, ο αλγόριθμος δημιουργεί τις τετράδες με σκοπό μετά οι δυάδες εφαρμογών στο κάθε socket επεξεργαστή να είναι η βέλτιστη. Για αυτό ακριβώς, η τεχνική αυτή παρουσίασε σημαντική βελτίωση με ενεργοποιημένο το σύστημα κλειδώματος των εφαρμογών στους επεξεργαστές.

6. Σύνοψη και Μελλοντική δουλειά

6.1 Σύνοψη

Έχοντας πλέον δημιουργήσει πληθώρα εφαρμογών προς εξέταση, αλγοριθμικές τεχνικές, συστάδες Κυβερνήτη και μεθόδους κατηγοριοποίησης εφαρμογών, καταφέραμε κάτι ξεχωριστό στην απόδοση επίγνωσης παρεμβολών στον Κυβερνήτη. Αυτή η ικανότητα του Κυβερνήτη να μπορεί πλέον να κατανέμει πιο έξυπνα τις εφαρμογές σε ένα υπολογιστικό σύστημα, αποφεύγοντας έτσι την δημιουργία ζευγαριών που θα προκαλέσουν παρεμβολές μεταξύ τους είναι κάτι που δεν έχει δημιουργηθεί επαρκώς ως τώρα. Οι αλγοριθμικές τεχνικές, με την βοήθεια του καρφιτσώματος τους στους επεξεργαστές που τους έχουν ανατεθεί από τον αλγόριθμο, επιτρέπουν την μείωση των

παραβιάσεων μέχρι και 74% στην καλύτερη περίπτωση. Αυτή η μείωση των παραβιάσεων και άρα αύξηση της απόδοσης των εφαρμογών οφείλεται στην εξοικονόμηση υπολογιστικών πόρων που στοχεύει ο κάθε αλγόριθμος. Αν και ο αριθμός των εφαρμογών οροσήμων προς εξέταση των τεχνικών ήταν μικρός, καθώς και το περιβάλλον εκτέλεσης αποτελούταν από τέσσερις υπολογιστές, η διπλωματική αυτή αποτέλεσε την πρώτη σοβαρή προσπάθεια να έχουμε έναν ικανό Κυβερνήτη με επίγνωση πιθανών παρεμβολών μεταξύ εφαρμογών.

6.2 Μελλοντική δουλειά

Όπως αναφέρθηκε, η παρούσα διπλωματική περιορίστηκε σε μικρό αριθμό εφαρμογών, και αποτέλεσε το πρώτο σοβαρό βήμα στην βελτίωση του Κυβερνήτη. Όμως για την παρουσίαση ενός ρεαλιστικού και ικανού συστήματος Κυβερνήτη, πρέπει να εξεταστεί μια διαφορετική και πιο ρεαλιστική διαδικασία ελέγχου των αλγοριθμικών τεχνικών. Συγκεκριμένα, οι εφαρμογές πλέον δε θα είναι ένας σταθερός αριθμός συγκεκριμένου τύπου υπολογιστικής εργασίας διεργασιών, αλλά αντιθέτως Ν (δηλαδή τυχαίος και υψηλός αριθμός) μικρό-υπηρεσίες ή αλλιώς microservices που συνηθίζονται να εξυπηρετούνται σε υπολογιστικά συστήματα παρόχων. Ακόμη, οι εφαρμογές πλέον προφανώς, δεν θα μπορούν να έχουν χαρακτηριστεί νωρίτερα αφού εμφανίζονται για πρώτη φορά στον εξυπηρετητή. Συνεπώς, είναι απαραίτητη η δημιουργία μιας νέας τεχνικής κατηγοριοποίησης εφαρμογών που με την χρήση άλλων μέσων πρόβλεψης θα μπορεί να κατατάξει την νέα εφαρμογή σε μια από τις τέσσερις κατηγορίες πριν δράσει ο αλγόριθμος δρομολόγησης. Κάτι τέτοιο θα είναι δυνατό με τη χρήση ενός συγκεκριμένου κενού κόμβου όπου θα αξιοποιείται για την κατηγοριοποίηση της εφαρμογής, χωρίς την επίδραση και τη μείωση της απόδοσης της από κάποια άλλη εφαρμογή.

Αυτή η νέα κατάσταση απαιτεί και καλύτερη παρακολούθηση των εφαρμογών. Το κλείδωμα των εφαρμογών στους επεξεργαστές δε θα είναι πλέον μια απλή μεμονωμένη συνάρτηση που θα ακολουθεί την αλγοριθμική τεχνική κατά τον ίδιο τρόπο. Αντιθέτως θα χρειαστεί να περνάμε την πληροφορία της θέσης των εφαρμογών σε ένα Custom Resource, δηλαδή σε έναν ειδικής κατασκευής πόρο. Αυτός ο ειδικός πόρος θα περιγράφει αναλυτικά την κατάσταση του κάθε κόμβο με κοινούς υπολογιστικούς πόρους. Έτσι ανά πάσα στιγμή και ο αλγόριθμος δρομολόγησης αλλά και ο αλγόριθμος κλειδώματος των εφαρμογών στο DaemonSet, θα γνωρίζουν την κατάσταση του κόμβου και θα λαμβάνουν την σωστή απόφαση αναλόγως.

Η ρεαλιστική και εξονυχιστική παρουσίαση των παραπάνω θα δημιουργήσουν επανάσταση στα νεφοϋπολογιστικά συστήματα επιτρέποντας επιτέλους στον Κυβερνήτη, να κατανέμει εφαρμογές με πλήρη επίγνωση πιθανών παρεμβολών μεταξύ τους, επιτρέποντας έτσι στους παρόχους να εξοικονομούν πόρους για τις εφαρμογές τους, διατηρώντας τες όμως σε υψηλά επίπεδα απόδοσης.

1. Introduction

1.1 Thesis Purpose

Organizations are continuously virtualizing their client architectures, data environments, and back-end systems in order to improve the user experience provided to customers. In the past, distributed client computing, has had and still has many limitations, such as network losses, compromised security and unpredictability creating as a result, a suboptimal quality of service. Virtualization has allowed a more converged infrastructure to take place, which created well-managed virtual clients. That way, applications and client operating environments are hosted on servers and their data are stored in data centers, providing scalability in exchange for a monthly operating cost. Therefore, users can access services from any location, without being tied to a single client device. However, hosting such applications on servers is expensive and requires an important amount of computing resources (such as CPU, RAM, etc.) to be provided. Thus, the continuous increase in number of workloads uploaded and executed on the cloud, has forced cloud providers and data center operators to embrace workload co-location and resource sharing as the primary concern regarding resource efficiency. Correspondingly, one of the most common and gradually increasing problems in informatics is about resource management and cost reduction in providing the above-mentioned cloud services. Resource management frameworks have been developed and are still being valued as necessary tools in resource efficiency, by every major cloud operator that provides hosting to applications.

Another type of virtualization, that is also gaining popularity rapidly, is containerization. Containerization also known as operating-system-level virtualization refers to the operating system feature in which the kernel allows the existence of multiple isolated user-space instances. This practice has led to the increasing popularity of Docker, a platform as a service (PaaS) framework[2] that uses the aforementioned operating-system-level virtualization, delivering software in packages called containers. Containers allow applications to run anywhere (provided the container is targeted at the operating system) providing great mobility on development and deployment. Furthermore, containerization allows one application to be isolated from other applications, thus achieving easier maintenance, testing, and debugging.

The purpose of this thesis is to provide solutions in resource management for cloud operators and data centers. Organizations are not only using container-based virtualization to develop their applications, but also to allow cloud computing to increase availability and help them improve the user experience. As a result, it is imperative that computing resources are well managed and distributed to support such cloud technologies. Nonetheless, orchestrators focus mostly on availability rather than performance

optimization. One exceedingly popular container orchestration system is Kubernetes. Kubernetes has been an excellent candidate for automating software deployment, scaling, and managing applications. However, Kubernetes still lacks interference awareness and the ability to co-locate applications based on application usage and efficiency.

In this work, we propose a set of algorithmic solutions that customize the behavior of the Kubernetes scheduler, implementing interference awareness. These algorithmic concepts are deployed on the scheduler via a plug-in framework creating an out-of-tree version of the Kubernetes scheduler, allowing it to efficiently place applications on a cluster of physical machines. The purpose of this project is for our set of scheduler plug-ins to meet the Quality of Service (QOS) constrains, whilst also ensuring resource utilization at the maximum possible value. Four different algorithmic solutions are presented with this paper, all of them outperforming the default Kubernetes scheduler in application efficiency. These algorithmic concepts are implemented via the above-mentioned scheduler plug-ins and are compared with the default scheduler and with each other. Several types of applications in different sets of workloads arrived to our cluster, testing our scheduler plugins and stressing the available resources in discrete tasks. In order to simulate the arriving applications, we used specific benchmarks from the SPEC library, which were classified into the required types of applications beforehand, with the use of our custom predictors.

2. Theoretical Background

2.1 The Issue

Orchestration software usability is growing rapidly, offering solutions in automation deployment and high-availability for container-based virtualized systems. Nonetheless, resource management is still based on coarse metrics and neglects interference effects, overlooking the stress applied upon shared resources. More specifically, Kubernetes, a widely used container orchestration tool lacks interference awareness when scheduling applications on a server. The default scheduling mechanism of Kubernetes filters the available nodes and deploys the applications given that there are enough resources. However, the potential interference between collocated applications is not taken into consideration by the default scheduler and as a result, performance issues appear. The resource criteria required by every application might be met, but application contentiousness is not considered, which can lead to performance degradation.

Applications arriving at a server have a set of characteristics that define application intertemporality. Specifically, in the span of a server runtime, the collocation of several applications on a CPU socket will cause significant changes on application performance, based on a set of two characteristics, namely contentiousness and sensitivity[14]. Contentiousness defines the potential of one application, to affect other applications

hindering their performance and increasing their runtime. Sensitivity on the other hand, defines how susceptible one application is to performance decline because of increased stress applied on needed resources. As a result, four sets of applications can be extracted from the above two attributes. For an easier reference throughout this document, applications that are insensitive (little to no change in runtime and performance when stressed by other applications) and non-contentious (do not alter the performance of other applications, when collocated together), will be classified under category A. On the same page, applications that are sensitive but also non-contentious are classified under category B, those that are insensitive but contentious are classified under category C and lastly, those that are both sensitive and contentious are classified under category D.

2.2 Solution Setup

The purpose of this thesis and project is to create algorithmic solutions that aim to create pairs of applications, which fit well together and eliminate performance degradation. Applications require certain amounts of computing resources, with the most common resource type being CPU. CPUs can have multiple cores and sockets allowing a greater number of physical Processors to do the computational work. In order to simulate application distribution and resource management for our project, we set up a server consisting of four physical machines. Each CPU has two sockets and each socket has four cores so as a result, we have a total of eight virtual CPUs (or vCPUs). In Kubernetes terms, each physical machine (and as a result, each CPU) is set as a Kubernetes Node, where applications will be sent to. To accurately test application degradation and sufficiently evaluate the performance of our algorithmic solutions, we require quadruples of applications to be formed and sent to each Node. Then accordingly, two pairs of applications will be created, one for each socket. Therefore, our workloads consisted of sixteen applications, distributed by the Kubernetes Scheduler as quadruples to the four Nodes.

We created several different workloads of applications to stress our server's resources and display different scenarios of application arrival. These workloads can be then classified into three categories, High, Mid and Low. This characterization describes the intensity of each workload and the contentiousness of most applications. Workloads aimed to accurately evaluate our algorithms in different scenarios of intensity. The applications used in the workloads, had to meet certain research criteria. Thus, we used the well-known benchmark library named SPEC, which offered us a set of benchmarks with multiple attribute cases. A set of predictors classified the benchmarks into the above-mentioned four categories.

To test diverse ways of application distribution and improve application performance, we implemented three different algorithms. The first algorithm is the greedy one, which tries to fit as many non-contentious applications together in nodes, before pairing other contentious applications with them. However, such an algorithm can and will

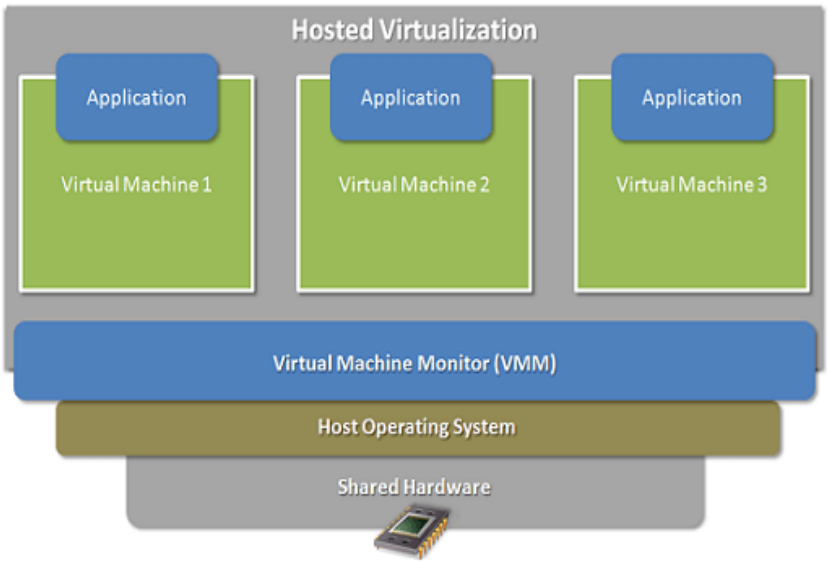
create wasteful pairings, by not collocating contentious applications with insensitive ones as much as possible. As a result, the second developed algorithm, named resourceful, tries to create good socket pairs, by setting up as many good pairs as possible, avoiding pairing insensitive applications with non-contentious ones and instead pairing them with contentious ones. Furthermore, a third algorithm was developed in order to test a different approach of pairing applications into quadruples. This algorithm, called neglectful, tries to sacrifice applications that are contentious by pairing them together, in order to allow sensitive applications to reach optimal performance. All these algorithms are evaluated by comparing their performance with the usual random scheduling algorithm that does not consider such application characteristics.

2.2.1 Virtualization

Before we dive into the importance of interference awareness and resource management in cloud services, it is vital we understand the use of virtualized systems for such operations. Virtualization and cloud computing are increasingly becoming more popular and are a core dynamic for many organizations. Cloud computing, simply means storing and accessing data and programs over the Internet, instead of a physical machine's hard drive. Users can access applications remotely through internet connection devices and by this, computer resources are used more efficiently and effectively. As a result, applications are centralized in server-specific architectures and then are accessed over wireless connection based on a thin local client or a web browser. Therefore, virtualization occurs when a virtual version of something is created, instead of an actual version. As we can easily understand, virtualization is the backbone of Cloud Computing, since it allows the imitation of hardware within a software program. Via virtualized systems, cloud computing can function and provide service to end users over the Internet. To summarize, virtualization in cloud computing refers to the creation of virtual hardware, software and operating systems, thereby preventing the need for separately installing software on every physical machine [1].

Different virtualization types have been developed, in order to satisfy computing needs on software management and resource optimization. The three major types of virtualization are known as, Server virtualization, Client Virtualization and Storage Virtualization[1]. The most common type of virtualization, and the one that affects cloud computing the most, is Server virtualization. Specifically, Server Virtualization in cloud provides certain advantages like hardware optimization and improvement to application uptime. Server virtualization accomplishes that by abstracting or isolating the computer's hardware from all the software that might run on that hardware. This abstraction is achieved by the hypervisor, a specialized software product, recognizing the computer's physical resources and creating logical aliases for those resources. For example, a physical processor can be abstracted into a logical representation called a virtual CPU or vCPU. The

hypervisor is responsible for managing all the virtual resources that it abstracts and handles all the data exchanges between virtual resources and their physical counterparts. This logical isolation [1], combined with careful resource management, enables the creation and control of multiple Virtual Machines on the same physical computer at the same time, with each VM capable of acting as a complete, fully functional computer. Nonetheless, Virtual Machines are not the only solution when creating virtual environments. Instead, containers may provide even better support for web applications and microservices, since containers are portable, scalable, and reliable. With the increasing popularity of Docker and Kubernetes, container-based systems are becoming more popular, as a solution, in cloud-based Server virtualization.



Virtualization Hierarchy

However, the actual number of VMs or container-based systems that can be created is limited by the physical resources present on the host server and the computing demands imposed by the enterprise applications running on the virtualized environment. The logical representation of the computer resources is attached to the actual set of physical resources and therefore, the provided resources to virtualized systems must be competently managed, to save up costs. This intent has fabricated an entire new area of interest known as, resource management.

2.2.2 Resource Management

The increasing number of CPU cores and storage capacity in data centers has increased competition of software components for finite resources on the platform. Applications, hosted on such data centers, compete for resource usage, and interfere with

each other. Therefore, a negative impact is produced on service assurance, the performance of the applications and on the end user's overall quality of experience. Current virtualized environments have orchestrators, which help improve workload management and application resource allocation. Specifically, Kubernetes is a well-known orchestration tool for managing and scheduling applications on containerized systems and has subsequently gained a massive following in cloud-based computing. Kubernetes helps automate software deployment, scaling, and management whilst also allocating resources to deployed applications in a cloud-based environment. However, Kubernetes allocates resources to hosted applications aiming to improve availability[4] rather than performance optimization. Resource management proposals for frameworks such as Kubernetes depend on coarse metrics like CPU or memory utilization, thus neglecting interference effects neglecting the imposed stress from applications onto shared resources.

In this paper, we aim to examine resource management under the prism of interference awareness, implementing a new way of monitoring applications on a server with the use of the well-known framework Kubernetes. Kubernetes, with its default behavior, will schedule applications onto available machines given that there are available resources for the applications to run on. That way, not considering the contentiousness of applications collocated together; performance issues may arrive reducing the targeted Quality of Service (QoS) of applications. As a result, we aim to understand how applications affect each other when hosted together on the same physical machine. The resource criteria for the scheduled applications might be met, thus creating the assumption that applications are running without performance degradation, however the shared pool of resources between the applications can still be stressed to the point where performance issues come forth.

Subsequently, it is crucial we understand how applications operate and how much stress they impose on shared computing resources. In this paper, we study the impact that collocated applications can have, on the resource known as CPU. Since our environment of study is virtualized, the CPU resource type will be instead be referenced as vCPU. Collocated applications hosted on a physical machine, will not necessarily cause performance downgrade to each other. Instead, this possibility is examined under two sets of characteristics, known as contentiousness and sensitivity. One application can be contentious and as a result, stress the shared pool of resources enough to hinder the ability of other applications to operate. On the other hand, one application can be insensitive and being collocated with a contentious application, will cause no performance drop, as it can withstand such a stress and operate under the targeted QoS. This revelation, generates two tasks to carry out with the first being a way to classify the applications, and the latter being about finding an approach to pair different types of applications together.

2.2.3 CPU Socket Architecture

Before we can classify applications based on their characteristics, it is essential we firstly understand how applications interact with the shared pool of CPU they have, at a cloud-based server. In a multi-node cluster environment like most cloud systems, resources are grouped and are assembled as separate node, with their own memory and CPU resource count. In multiprocessing systems, which dictate and consist of most of today's computing environments, we have what could be called NUMA nodes. Non-uniform memory access (NUMA) is a computer memory design used in such systems, where the memory access time depends on the memory location relative to the processor. Under NUMA, a processor can access its own local memory faster than non-local memory. That allows modern CPUs to operate considerably faster than those that used a shared memory link. However, with today's dramatically increased size of operating systems and of the applications run on them, a system can starve several processors at the same time, since only one processor can access the computer's memory. This is surpassed by grouping CPUs in a multi-channel memory architecture. NUMA nodes consist of CPUs and one shared memory link. Not every CPU is coupled under the same memory thus causing significant CPU throttling but instead, different Nodes balance the memory load and serve applications with their respective CPU resources.

This implementation allows the assemblage of several CPU sockets under a shared memory link and a NUMA node. A CPU socket is the physical socket where the physical CPU capsules are places. In many of today's cloud computing designs, physical CPUs have multiple sockets and each socket hosts several CPU cores. Then each core hosts two threads that execute the load requested by the application. In a node environment like in Kubernetes, physical restrictions still exist and need to be examined. Creating a NUMA node under each physical machine, equals the creation of a group of CPU sockets with a shared memory link. Each socket can have several amounts of CPU cores that enable it to run applications. Those CPU cores are only accessible by applications hosted directly on those sockets and such applications will battle for CPU resources only when collocated there. Every socket as a result is a shared pool of CPU resources to run computational tasks and every applications hosted on them, will affect the rest collocated there.

2.2.4 Application Characteristics

Now that we understand how applications make use of computing resources, such as CPU, it is easy to classify them based on their behavior. Specifically, applications placed on a physical machine, will stress the available resources, in order to complete their tasks. In a real-time server environment, infinite space is not available and as a result, applications will be collocated together and share resources among the pool of available ones. Several applications might run on the same physical machine, given that there are enough resources and their resource criteria are met. This can falsely create the belief that since their

resource criteria are met, applications will run appropriately and show no signs of performance degradation. This common misbelief is caused by the ignorance of application contentiousness. Application contentiousness defines the ability of one application to severely alter the performance of other collocated applications. More specifically, two or more applications can be collocated, with all of their resource criteria met, and performance degradation will still be observed, because of at least one application is contentious. Therefore, applications observed during our time will be labeled with the contentiousness tag, depending on whether they can alter the runtimes of other applications.

However, not all applications will display performance degradation when collocated with contentious applications. The behavioral factor that decides whether one application will be susceptible to performance drop is their sensitivity. Many applications can be collocated with others, with increased stress on the shared pool of resources and still show no sign of performance issues. Such applications will be classed as insensitive, whereas those that do show performance decline will be classed as sensitive.

The two tags, namely contentiousness and sensitivity, describe the deciding behavioral factors of application collocation. Resource criteria fail to explain the interference caused between collocated applications, whereas sensitivity and contentiousness do so. Correspondingly, it is easily understood, that four different application categories can be created, depending on the two given tags. Under category A (which will be called catA for the rest of this paper), we will classify applications that are insensitive and non-contentious. Such applications cause no interference to how other applications run, whilst also having no performance issues regardless of the amount of stress allocated to their resources. Under category B (catB), we place applications that are non-contentious too, but also sensitive. These applications do not cause performance issues to other collocated applications; however, they are susceptible to performance issues on their own. On the other hand, applications that are contentious and insensitive will be classed under category C (catC). Such applications are insensitive and as a result can withstand stress upon their shared resources by other applications, but also cause severe interference to other collocated applications. Lastly, applications that are both sensitive and contentious will be classed under category D (catD). Applications that fall under category D, are not only susceptible to performance degradation, but also cause performance issues to other collocated applications.

In practical terms, it is imperative that applications are classified under the aforementioned categories, in order to have a better understanding of their behavior. Therefore, a set of predictors need to be implemented, which will cross-validate application runtimes when collocated together. A change in runtime performance that surpasses the established QoS, will be the factor in deciding under which category applications will fall onto. Correspondingly, applications will be scheduled on a virtualized system depending on the application category they fall under.

2.2.5 Scheduling Applications

In a virtualized system, physical resources are abstracted into logical representations, such as the CPU cores. When scheduling applications in such systems, with the use of well-known frameworks like Kubernetes, the criteria are limited in satisfying resource quota and providing the necessary resources for the applications to run on. A physical server may consist of several CPU cores, and as a result with several CPU sockets. Given that we now are in a virtualized environment, each CPU is represented as a vCPU and is shared between hosted applications. The default scheduling mechanism of any container-management framework will search in satisfying the resource needs of as many applications as possible, regardless of the interference caused by their collocation.

Thus, a new scheduling mechanism needs to be created that can understand the category, under which an application may fall, and enforce algorithmic solutions on distributing them to the available nodes. Kubernetes can make use of VMs or physical machines and represent them as available nodes that will host and manage arriving applications. When an application arrives to a server, the Kubernetes scheduler ranks the available nodes and selects the best fit, on which to host the arriving application. In this paper, we examine the behavioral change of the Kubernetes scheduler, when applying plugins that implement different algorithms to application scheduling and receive measurements in evaluating them.

With our set of scheduler plugins, we aim to change the focus of the Kubernetes scheduler from resource quota availability to interference awareness. Regardless of which algorithm is implemented to our scheduler, the aim is to alter the behavioral focus of Kubernetes scheduling, in order to better pair applications together on available nodes and reduce performance issues created by application interference.

2.2.6 Interference awareness

Many enterprises and organizations have developed interference aware[5] scheduling mechanisms to increase resource utilization in cloud environments. Due to concurrent running of co-located tasks on top of physical machines, the performance of some tasks suffers significantly because of interference. Older scheduling mechanisms that aimed into distributing applications depending on resource quota, failed to constrain performance issues attributed to interference. To improve the QoS and reduce the amount of Service Level Agreement (SLA) violations between the user and the cloud service provider, interference-scheduling approaches have been developed.

Predictor models have been introduced to estimate potential interference latency during application co-location, followed by interference aware schedulers that distribute applications, depending on the classification imposed by the predictors[13]. Such

mechanisms have already been developed and implemented in virtualized systems, yet no concrete breakthrough has happened around Kubernetes. Intel has proposed a Resource Aware CPU Scheduling mechanism, known as Intel CPU Manager for Kubernetes[3] (also called CMK). This mechanism enables core pinning and isolation, depending on resource quota dictated by applications. However, interference metrics are not considered and CPU pinning is based on CPU affinity and CPU limits, set upon the placed Pod.

Our proposal aims not only in creating a set of predictors that classify applications based on their interference, but also in allowing actual CPU pinning on arriving applications. The predictors cross-validate application runtimes and classify applications, in order to tag them accordingly when arriving at a server farm. The base metrics in performance evaluation are based on CPU stress and execution runtime. Runtimes that exceed the set QoS are classed as SLA violations and are registered for the predictors' evaluation. Then, the scheduling plugins implemented on the Kubernetes scheduler filter and score available Nodes, in order to choose the best fit for the arriving application. That way Kubernetes can acquire interference awareness and utilize CPU resources accordingly.

2.3 Kubernetes

Though already mentioned many times in the introductory part of this thesis, Kubernetes is a huge and complex orchestration tool that needs increased analysis into understanding its key features. In order to efficiently comprehend and then develop an interference aware logic, it is imperative that we firstly understand its primary features and components.

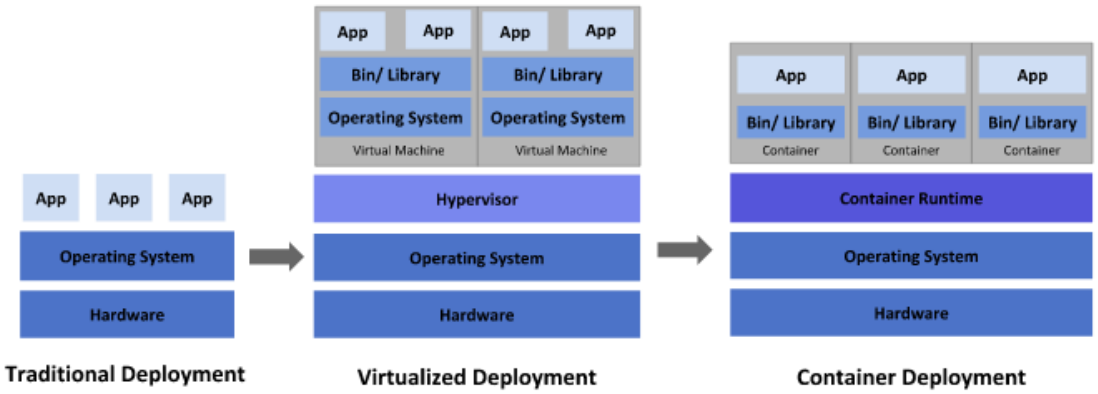
2.3.1 Understanding Kubernetes

Before we dive into the importance of interference awareness in containerized systems, that Kubernetes manages and orchestrates, it is important to understand the reason behind the increased popularity of Kubernetes. An interference aware Kubernetes, is not just a thesis topic to be examined under the prism of academic research, but an actual practical problem in cloud-based server hosting. Kubernetes is widely used by many hosting platforms and is a standard approach in managing and developing containerized software in cloud-based environments, such as data centers. Early on, organizations and server providers ran applications on physical servers. There was no way to define resource boundaries and collocated applications on a physical machine caused resource allocations issues. The solution to that was to run each application on a different server, which caused resources to be unutilized and increased expenses dramatically.

In today's virtualized and containerized deployment era, virtualization has allowed providers to multiple VMs or containers on a single physical server's CPU. Virtualization has enabled applications to be isolated between VMs, have better resource utilization and

better scalability. Containers like VMs, have relaxed isolation properties to share the Operating System (OS) among applications. As a result, containers have become extremely popular since they provide many benefits such as:

- Agile application creation and deployment.
- Continuous development, integration, and deployment.
- Environmental consistency across development, testing and production.
- Cloud portability. Can run on all major public clouds.
- Loosely coupled, distributed and elastic applications. Applications are broken into smaller, independent pieces and can thus be deployed and managed dynamically.
- Resource isolation and predictable application performance (NOT interference aware though!)
- Resource utilization providing high efficiency and density with application collocation.



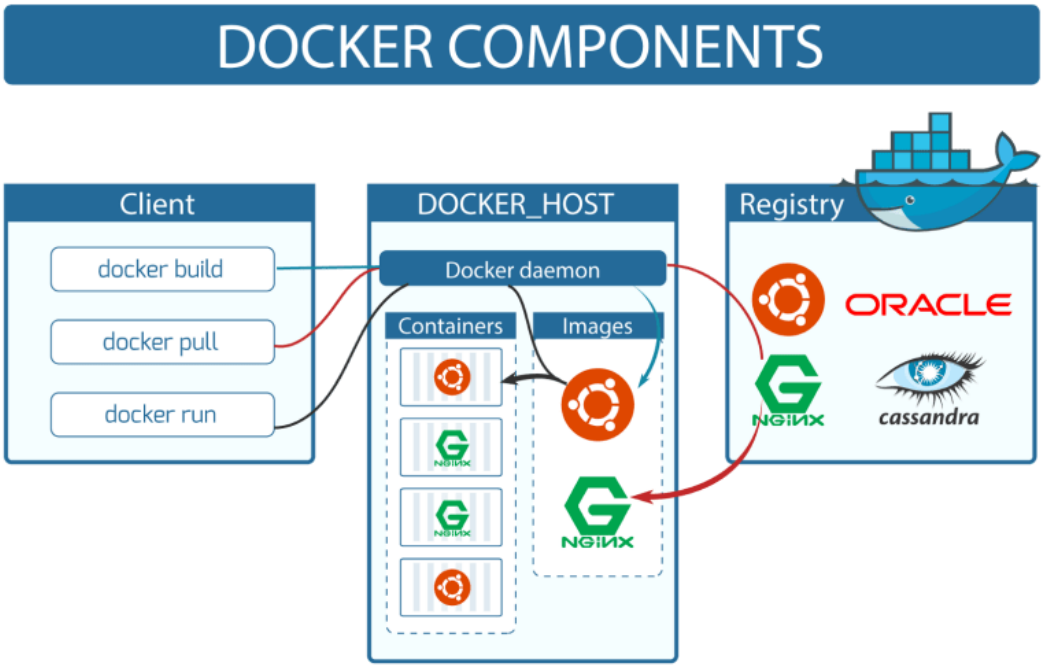
From Traditional to Containerized Applications

Kubernetes is a portable, extensible, open-source platform for managing containerized workloads and services that facilitates both declarative configuration and automation. It has, as a result, a large rapidly growing ecosystem that allows us to run distributed systems resiliently. Kubernetes has allowed service providers, to efficiently make use of their physical machines, utilize the available resources and ensure the desired QoS that the client demands [4]. Kubernetes manages the available resources, automates software development and deployment, offers self-healing to the hosted containers by restarting failed containers and allows for increased security and secret configuration management. Nonetheless, the management of such containerized systems is made available by the use of Container Runtime systems. Originally, Kubernetes interfaced exclusively with Docker runtime, through “Dockershim”[2], a platform as a service (PaaS) product that uses OS-level virtualization to deliver software in containers.

2.3.2 Docker

In order to understand how Kubernetes manages containers, it is imperative we understand how Docker firstly enables the packaging of applications into containers. Docker is an open platform for developing, shipping and running applications. Docker allows us to separate applications from the infrastructure so that software can be delivered quickly. Software is packaged in containers and thus can be run in a loosely isolated environment. This isolation allows many containers to run simultaneously on a physical host, that way improving resource utilization.

Specifically, Docker streamlines the development lifecycle by allowing developers to work in standardized environments using local containers which provide applications and services[2]. Docker is lightweight and fast. It provides a viable, cost-effective alternative to hypervisor-based virtual machines, so that more of the compute capacity can be used to achieve hosting goals. Docker manages that by using a client-server architecture. The Docker client talks to the Docker daemon, which then builds, runs, and distributes the Docker containers. The Docker client and daemon communicate using a REST API, over UNIX sockets or a network interface.



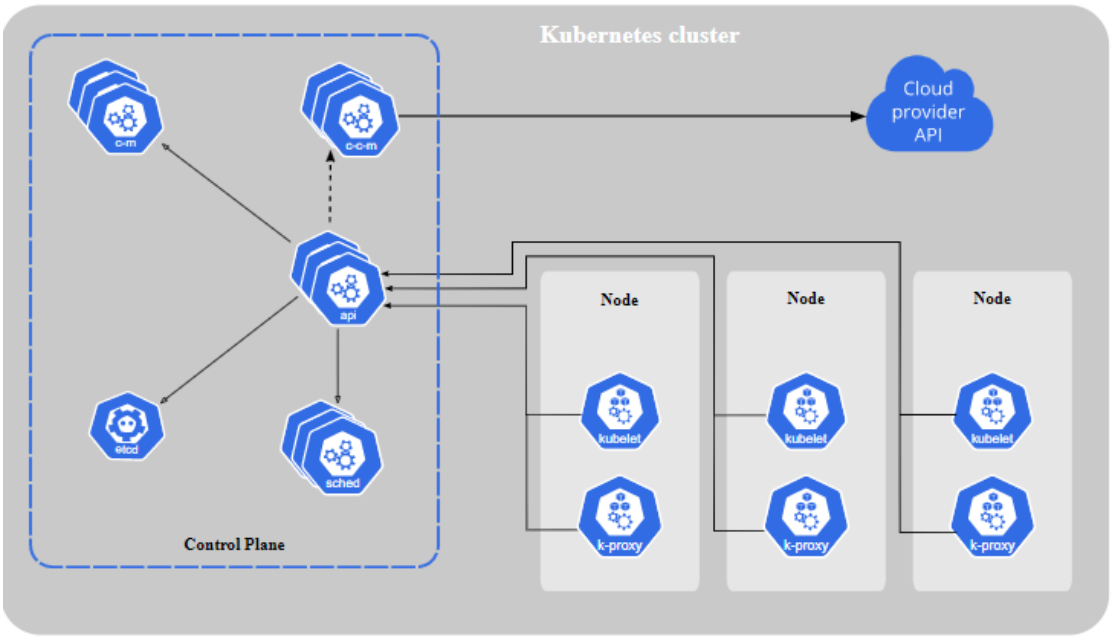
Docker Components

Docker is a viable and preferred option for containerizing applications in order to manage them with the use of Kubernetes. In this thesis, Docker images are created and then deployed as containers to run standardized tests and evaluate Kubernetes performance. A Docker image is a Docker object and a read-only template with instructions for creating a

Docker container. Docker images are built with the use of Dockerfiles[2] and therefore, a change to a Dockerfile will rebuild the image, only changing the layers needed. This is part of what makes images so lightweight, small and fast, when compared to other virtualization technologies. Therefore, we can summarize into having lightweight, easily deployed and isolated applications running as containers, on a virtualized system that are managed by Kubernetes.

2.3.3 Kubernetes components

Now that we have understood the use of Kubernetes and Docker, it is time to dive deep into the Kubernetes architecture and understand the way it implements all of its features. Firstly, when Kubernetes is deployed on a system, we get a cluster[6]. A Kubernetes cluster consists of a set of worker machines, called nodes that run containerized applications. Every cluster needs to have at least one worker node. The application workload hosted on the worker nodes, comes in containerized nature, as already described. A group of one or more containers, with shared storage and network resources, is called a Pod[6]. Pods are the smallest deployable units of computing that can be created and managed in Kubernetes. Aside from the worker nodes hosting the Pods, we also have a physical machine [or set of physical machines when aiming for High Availability (HA)] that is the control plane, also known as the master node, which manages the cluster and its functions.



Kubernetes Cluster Architecture

The control plane has components that make global decisions about the cluster, as well as detecting and responding to cluster events. The first component we will describe is the kube-apiserver or simply the API server.

Kube-apiserver: The API server is a component of the Kubernetes control plane that exposes the Kubernetes API. It is designed to scale horizontally, by deploying more instances and that way balancing the traffic. The Kubernetes API server is responsible for validating and configuring data for the API objects such as Pods, services, and others[6]. The API Server services REST operations and provides the frontend to the cluster's shared state through which all other components interact. It exposes an HTTP API that lets end users, different parts of the cluster, and external components to communicate with one another. Furthermore, the Kubernetes API allows us to query and manipulate the state of API objects in Kubernetes. We exploit that capability later, when we manipulate the Kubernetes scheduler and via the Kubernetes API, we can query the cluster state and provide interference awareness to the desired scheduler. Kubernetes stores the serialized state of objects by writing them into the etcd[6].

Etcd: Etcd is a consistent and highly available key value store used as Kubernetes' backing store for all cluster data. It is the primary data store of Kubernetes and allows Kubernetes to store data and manage the critical information that distributes systems need to keep running. The status of the system is saved at the etcd providing a single and standard truth about the state of the system Etcd is highly available, fast and secure since it supports TLS and SSL client certificate authentication[7].

Kube-scheduler: This control plane component watches for newly created Pods with no assigned node and selects a node for them to run on. A node, also called a worker node, is a physical or virtual machine that hosts deployed Pods. It ranks each valid Node and binds the Pod to the suitable Node[6]. The scheduler considers several factors in order to decide and find the best fit for the Pods, from the available nodes. These factors include resource requirements, affinity specifications, software, and policy constraints, and more. For us to implement interference awareness to Kubernetes, we created our own set of factors, via the use of scheduler plugins, to alter the behavior of the scheduler. This will be described thoroughly at a later stage of the thesis.

Kube-controller-manager: The controller manager is the control plane component that runs controller processes[6]. These processes are control loops that watch the state of the cluster, then make or request changes where needs. Each controller tries to move the cluster from its current state, closer to the desired state. A controller tracks at least one Kubernetes resource type. Almost every Kubernetes object includes two nested object fields that govern the object's configuration, the object *spec* and the object *status*. The spec field provides the characteristics that the object wants to have, its desired state. The status field describes the current state of the object, supplied by the Kubernetes systems and its components. The controller will act in order to get the object to each desired state. Sometimes the controller may carry the action out itself, though more commonly, in Kubernetes, a controller will send messages to the API Server to proceed with the task.

Kubernetes that way can handle constant change and could be changing at any point as work happens and control loops automatically make changes to help the cluster reach its desired state. Some types of controllers are:

- **Node controller**: Responsible for noticing and responding when nodes go down
- **Job controller**: Watches for Job objects that represent one-off tasks, then created Pods to run those tasks to completion.
- **Endpoints controller**: Populates the Endpoints object
- **Service Account & Token controllers**: Create default accounts and API access tokens for new namespaces.

Aside from the control plane components, there are Node components that run on every node, maintaining running pods and providing the Kubernetes runtime environment.

Kubelet: Kubelet is an agent that runs on each node in the cluster. It can register the node with the API Server using either the hostname, a flag to override the hostname, or a specific log for a cloud provider[6]. The kubelet works in terms of a PodSpec. A PodSpec is a YAML or JSON object that describes a pod. The kubelet takes a set of PodSpecs that provided through various mechanisms (primarily thought the API Server) and ensures that the containers described in those PodSpecs are running and healthy. It will not manage containers that were not created by Kubernetes. Other than receiving the PodSpec from the API Server, there are three more ways that a container manifest can be provided to the Kubelet.

- **File:** A container manifest can be passed as a file to the Kubelet in order to be deployed and monitored. A path will be provided and files under it will be monitored periodically for updates.
- **HTTP endpoint:** HTTP endpoints can be passed as parameters on the command line.
- **HTTP server:** The kubelet can also listen for HTTP and respond to a simple API call.

Kube-proxy: Kube-proxy is a network proxy that runs on each node in the cluster. This reflects services as defined in the Kubernetes API on each node and can do simple TCP, UDP, and SCTP stream forwarding or round-robin TCP, UDP, and SCTP forwarding across a set of back-ends. More specifically, kube-proxy maintains network rules on nodes. These network rules allow network communication to the Pods from network sessions inside or outside the cluster.

Container runtime: The container runtime is the software that is responsible for running the containers. As already referenced, Docker is a container runtime environment that is used in Kubernetes, though since being deprecated in 2016; containerd and CRI-O are used more often.

Addons: Addons use Kubernetes resources to implement cluster features. Many add-ons extend the functionality of Kubernetes, however in this thesis we will focus more on two specific ones, that were later used in our test environment cluster.

Calico: The first one to describe is Calico. Calico is an open source networking and network security solution for containers, virtual machines, and native host-based workloads[8]. It also supports Kubernetes and was as a result used in this thesis' testing environment, as a networking plugin for the cluster. Calico offers rich network policy models in order to lock down communication so that the only traffic flows are the ones the system wants. Furthermore, Calico's core design principles leverage best practice cloud-native design patterns. The result is a solution with scalability in multi-node clusters. This helps with implementing Kubernetes network policy during the development of the API. It implements the full set of features defined by the Kubernetes-API giving users the required flexibility.

CoreDNS: CoreDNS is a flexible, extensible DNS server that can server as the Kubernetes cluster DNS[9]. It can replace kube-dns in the cluster in providing the DNS server functionality. CoreDNS is a single container per instance, while kube-dns uses three. Thus, the larger number of containers per instance in kube-dns increases base memory requirements and adds some performance overhead, legitimizing the CoreDNS choice.

2.3.4 Pods

Aside from the vital Kubernetes components, worker nodes host applications, in small deployable unites called Pods. A Pod is a group of one or more containers, with shared storage and network resource, and a specification for how to run the containers[6]. A Pod's contents are always co-located and c-scheduler, and run in a shared context. A Pod, models an application-specific "logical host", which contains one or more application containers that are relatively tightly coupled. These co-located containers form a single cohesive unit of service and the Pod wraps these containers, storage resources, and an ephemeral network identity together as a single unit. They are automatically co-located and co-scheduled on the same physical or virtual machine in the cluster. They communicate with each other and coordinate when and how they are terminated.

Pods resource sharing is enabled via data sharing and communication among their constituent containers. A Pod can specify a set of shared storage volumes. At its core, a volume is a directory, possibly with data in it, which is accessible to the containers in a pod. How that directory comes to be, the medium that backs it, and the contents of it, are determined by the particular volume type used. All containers in the Pod can access the shared volumes, allowing those containers to share data that way. Volumes also allow persistent data in a Pod to survive in case one of the containers within needs to be restarted or terminated. Aside from storage sharing, Pods communicate via networking too.

Each Pod is assigned a unique IP address for each address family. Every container in a Pod shares the network namespace, including the IP address and network ports. Inside a Pod, the containers that belong to the Pod can communicate with one another using localhost. When containers in a Pod communicate with entities outside the Pod, the must coordinate how they use the shared network resources, such as ports. Within a Pod,

containers share an IP address and port space and can find each other using standard inter-process communications like SystemV semaphores or POSIX shared memory. Containers in different Pods have distinct IP address and cannot communicate by IPC without special configuration. Containers that want to interact with a container running in a different Pod can use IP networking to communicate.

Regardless of single or multiple containers, shared storage and network communication, all Pods are created from pods templates. Pod Templates are specifications for creating Pods, and are included in workload resources such as Deployments, Jobs and DaemonSets. Each controller for a workload resource uses the PodTemplate inside the workload object to make the actual Pod[6]. The PodTemplate is part of the desired state of the Pod. Modifying the pod template or switching to a new pod template has no direct effect on the Pods that already exist. If a pod template for a workload resource is changes, that resource needs to create replacement Pods that use the update template. On Nodes, the kubelet does not directly observe or manage any of the details around pod templates and update, since those details are abstracted anyway. That abstraction and separation of concerns simplifies system semantics, and makes it feasible to extend the cluster's behavior.

As already mentioned, when the Pod template for a workload resource is changed, the controller creates new Pods based on the update template instead of updating or patching the existing Pods. In order to update or maintain a set of Pods, workload resources can be used.

2.3.4.1 Deployments

One workload resource that provides declarative update for Pods is the Deployment. In the Deployment, one can describe the desired state and the Deployment Controller changes the actual state to the desired state at a controlled rate[6]. The Deployment can declare a new state of the Pods by updating the PodTemplateSpec of the Deployment. The Deployment manages moving the Pods to their new state at a controlled rate. The Deployment can also rollback the pods to an earlier version if the current state is not stable. Furthermore, assuming that horizontal Pod auto-scaling is enabled in the cluster, an auto-scaler can be set up for the Deployment, which chooses the minimum and maximum number of Pods necessary to run based on the CPU utilization of the existing Pods. In order to accomplish the aforementioned techniques, the Deployments will roll out a ReplicaSet.

2.3.4.2 ReplicaSets

A ReplicaSet's purpose is to maintain a stable set of replica Pods running at any given time. As such, it is often used to guarantee the availability of a specified number of identical Pods. A ReplicaSet is defined with fields, including a selector that specifies how to identify Pods it can acquire, a number of replicas indicating how many Pods it should be maintaining, and a pod template specifying the data of new Pods it should create to meet

the number of replicas criteria[6]. A ReplicaSet then fulfills its purpose by creating and deleting Pods as needed to reach the desired number. When a ReplicaSet needs to create new Pods, it uses its Pod template.

Conclusively, a Deployment is a high-level concept that manages ReplicaSets and provides declarative updates to Pods along with a lot of other useful features.

2.3.4.3 DaemonSets

DaemonSets are similar to Deployments in that they both create Pods, and those Pods have processes, which are not expected to terminate, such as web servers and storage servers. Deployments are used for stateless services, like frontends, where scaling up and down the number of replicas and rolling our updates are more important than controlling exactly which host the Pod runs on. On the other hand, a DaemonSet is used when it is important that a copy of a Pod always runs on certain hosts, if the DaemonSet provides node-level functionality that allows other Pods to run correctly on that particular node. For example, in a scheduling plugin, a DaemonSet component that enables the functionality of binding a Pod to a CPU socket, will make sure that the affected Pod will run at the selected Node.

2.3.5 Nodes

Kubernetes runs all of its components, Pods, and workload resources on Nodes. A node may be a virtual or physical machine, depending on the cluster[6]. Each node is managed by the control plane and contains the services necessary to run Pods.

There are two ways to have Nodes added to the API server:

1. The kubelet on a node self-registers to the control plane
2. A user manually adds a Node object

After a Node object is created the control plane checks whether new Node object is valid. Kubernetes creates a Node object internally and checks that a kubelet has registered to the API server matching the metadata name field of the Node. If the Node is healthy, it is eligible to run Pods. Otherwise, the Node is ignored for any cluster activity until it becomes healthy.

Each Node that is self-registered report their resource capacity during registration. If it is manually added, then the user needs to set the node's capacity information when adding it. Then the pods can consume the available resources in order to run on them. The Kubernetes scheduler ensures that there are enough resources for all the Pods on a Node. The scheduler checks that the sum of the requests of containers on the node is no greater than the node's capacity. That sum of requests includes all containers managed by the

kubelet, but excludes any containers started directly by the container runtime, and also excludes any processes running outside of the kubelet's control.

2.3.6 Cluster Architecture

Kubernetes has a "hub-and-spoke" API pattern[6]. All API usage from nodes (or the pods they run) terminates at the API-server. None of the control plane components is designed to expose remote services. The API-server is configured to listen for remote connections on a secure HTTPS port with one or more forms of client authentication enabled.

The control plane components also communicate with the cluster API-server over the secure port. As a result, the default operating mode for connections from the nodes and pods running on the node to the control plane is secured by default and can run over untrusted and/or public networks.

The control plane has two primary communication paths to the nodes. The first is from the API-server to the kubelet process, which runs on each node in the cluster. The second is from the apiserver to any node, pod, or service through the apiserver's proxy functionality. The connections from the apiserver to the kubelet terminate at the kubelet's HTTPS endpoint. On the other hand, the connections from the apiserver to a node, pod, or service default to plain HTTP connections and are therefore neither authenticated nor encrypted. However, they can be run over a secure HTTPS connection by prefixing *https:* to the node, pod, or service name.

2.3.7 Scheduler

Now, with a basic understanding of Kubernetes' functionalities, a more thorough understanding of the Kubernetes' scheduler can be achieved. As already described, a scheduler watches for newly created Pods that have no Node assigned. For every Pod that the scheduler discovers, the scheduler becomes responsible for finding the best Node for that Pod to run on.

The Kubernetes scheduler also known as kube-scheduler is the default scheduler and runs as part of the control plane. Kube-scheduler is designed so that, if someone wants and needs to, can write their own scheduling component and use that instead. For every newly created pod or other unscheduled pods, kube-scheduler selects an optimal node for them to run on. However, every container in pods has different requirements for resources. Therefore, existing nodes need to be filtered according to the specific scheduling requirements. After the scheduler finds available nodes, it runs a set of functions to score the nodes and picks the Node with the highest score among them, to run the Pod. The scheduler then notifies the API server about this decision in a process called binding. The factors that need to be taken into account for scheduling decisions include individual and

collective resource requirements, hardware, software, and policy constraints, affinity specifications, data locality, workload interference and so on.

Consequently, the 2-step operation, in which the kube-scheduler selects a node for the arriving pod, comes as followed:

1. Filtering
2. Scoring

The filtering step find the set of Nodes where it is feasible to schedule the Pod. The filters check the candidate Nodes for certain parameters, such as resource availability. If the Node does meet the Pod's resource requests, it adds the Node to the suitable Node list. In the scoring step, the scheduler ranks the remaining nodes to choose the most suitable Pod placement. The scheduler assigns a score to each Node that survived filtering, basing this score on the active scoring rules. Finally, kube-scheduler assigns the Pod to the Node with the highest ranking. If there is more than one node with equal scores, kube-scheduler selects one of these at random.

The Kubernetes scheduler is configurable and one can implement different scheduling profiles. A scheduling profile allows the configuration of different stages of scheduling in the kube-scheduler, via extension points. Plugins provide scheduling behaviors by implementing one or more of these extension points, such as filtering, scoring, sorting and more. This is the basis of our interference awareness implementation. By using the plugin extension points of the Kubernetes scheduler, we can enable our custom filtering and scoring steps in accordance with interference avoidance and enhanced application collocation.

2.4 Benchmarking

In order to create test cases and validate the assumptions of our thesis, it is imperative that actual applications are used, which will in fact stress the resources, cause interference to other workloads and allow a suitable evaluation of the scheduling plugins we have implemented. Using benchmarks, allows us to have quantitative data, choose from a selection of different types of test applications and helps imitate real workload scenarios.

Benchmarks are the acts of running computer programs, or other operations, in order to assess the relative performance of an object, in our case, the runtime performance of workloads during cloud hosting.

2.4.1 SPEC Benchmarks

For our test environment, the set of benchmarks that were used originate from the SPEC Benchmark suite. The benchmarks aim to test real-life situations from Java scenarios to simple computation jobs. The SPEC CPU suites test CPU performance by measuring the run time of several programs. For our workload cases, the CPU suite will be used in order to evaluate CPU performance and provide information on interference between applications. The 2006 SPEC CPU benchmarks contains two packages, the one being CINT2006 with two

suites, which tests integer arithmetic programs such as compilers, word processors etc, and the other one being CFP2006 with four suites, testing floating point performance, such as physical simulations, image processing etc. From these benchmark suites, we selected certain benchmarks to create different types of application templates and use them as test workloads arriving at a server farm.

- **400.perlbench**: This workload includes certain jobs, such as an email indexer and a SPEC tool that checks benchmark outputs.
- **445**.**gobmk**: Plays the game of Go, a simply described but deeply complex game.
- **462**.**libquantum**: Simulates a quantum computer, running Shor's polynomial-time factorization algorithm.
- **471**.**omnetpp**: Uses the OMNet++ discrete event simulator to model a large Ethernet campus network.
- **434.zeusmp**: ZEUS-MP is a computational fluid dynamics code developed at NCSA for the simulation of astrophysical phenomena.
- **436.cactusADM**: Solves the Einstein evolution equations using a staggered leapfrog numerical method.
- **444.namd**: Simulates large biomolecular systems.
- **450.soplex**: Solves a linear program using a simplex algorithm and sparse linear algebra. Test cases include railroad planning and military aircraft models.
- **459.GemsFDTD**: Solves the Maxwell equations in 3D using the FDTD method

These benchmarks will imitate an application being hosted on a Kubernetes server. A benchmark will be hosted on a container, and since a Pod can have multiple containers of the same benchmark running, we are able to create widely different and complex Pod specifications.

2.4.1.1 Runtimes

In order to evaluate benchmark and CPU performance, the benchmark runtime will be registered. The base runtime performance that will be used as the basis for any service level objectives, is the runtime recorded from the benchmarks when they are hosted alone. When applications are collocated together on nodes, the new runtime performances will be registered and compared with the base runtime of the benchmark, to determine if the Service Level Objectives have been met.

Since benchmarks have different singular runtimes, it is also imperative that workloads executing applications, have established common total runtimes, by repeated benchmark executions. For example, 436.cactusADM may have a singular runtime execution of 10 minutes and 444.namd may have a singular execution of 5 minutes. As a result, and to have a better depiction of the performance degradation that might occur when collocating the two different applications on a node, it is crucial that 444.namd executes twice as many

times over the course of action, so that the total runtime is the same. Then the average runtime is calculated and compared with the base runtime of each benchmark, to assess if the SLOs have been met.

An SLO or also Service Level Object is a key element of a service-level agreement (SLA) between a service provider and a customer. SLOs are agreed upon as a means of measuring the performance of the Service Provider and are outlines as a way of avoiding disputed between the two parties based on misunderstanding. In this thesis, we will cover multiple SLOs in order to test the efficiency of the scheduling algorithms. These SLOs will be determined on the runtime performances of the applications, by dividing the runtime performance of an application when collocated with others with the base runtime that it obtained when it operated alone. For example, the service-level agreement between a cloud provider and the customer might have established an SLO objective of 1.2, meaning that any application running on the server should avoid showing an increase in runtime performance by 20% or more. Such SLA violations, will dictate how efficient one scheduling algorithm has been, under the stress of a certain workload.

2.4.1.2 Containerization

For benchmarks to run on a computer, it was never anything more than a suite installation, allowing users to run test workloads at ease. However, testing benchmarks via Pod placement requires firstly the containerization of the benchmark suite.

In order to create a container that includes the SPEC Benchmark suite, we created a Dockerfile. Docker can build images automatically by reading the instructions from a Dockerfile. A Dockerfile is a text document that contains all the commands a user could call on the command line to assemble an image/container. The Dockerfile is as follows:

```
6 lines (5 sloc)  153 Bytes
1  FROM cmbant/docker-gcc-build
2  MAINTAINER jdologl@cslab.ece.ntua.gr
3
4  COPY SPEC_bench /benchmarks/SPEC_bench
5  WORKDIR /benchmarks/SPEC_bench
6  CMD ['runspec']
```

Dockerizing SPEC Library

This Dockerfile builds the image on a gcc compiler to allow the SPEC benchmark to run on. Then it copies the SPEC benchmark suites on the Docker image and by choosing the

correct work directory, we run the runspec tool to access the suite. The image is also uploaded to DockerHub and can be pulled via jmmisd/testspec .

2.4.1.3 Categorization

With the benchmark suite now containerized, we are able to construct Kubernetes Pods that can be deployed on a cluster. In order to categorize the benchmarks into the four available categories, we first need to acquire their runtimes. Once the benchmarks have been executed alone in the cluster, and their runtimes are registered, we can begin the process of classifying them. The classification process of the benchmarks resides on the cross-running of the benchmarks on the same socket. Each benchmark will run with every other benchmark allowing us to register their runtimes when collocated together. After the runtime data is available, we are able to start the classification progress with the use of the whiskers. The whiskers implement a method that handles a dictionary of results for each benchmark and calculates a limit based on the provided metrics. That limit is compared with a maximum value calculated once again by the whisker algorithm and that comparison is pivotal in the categorization of the application.

3. Solution Presentation

3.1 Environment Used

In order to assess the current proposal, a test environment was set up that allowed the simulation of a hosting provider infrastructure. We set up four computing systems that allowed as a result, the creation of a 4-node cluster to take place.

3.1.1 System Setup

The infrastructure available in our lab consists of four computing machines that are locally connected to each other. Each computing machine has 24 CPUs in total, of x86_64 architecture. Each core has two threads and every socket consists of six cores and as a result twelve threads. The CPU model name is an Intel Xeon CPU of 2.40 GHz frequency. In order to evaluate different pairing algorithms and construct specific topologies where applications will be collocated, two NUMA nodes are apparent on each machine. Each NUMA node has its own memory link and balances its own load. NUMA node0 has the first two sockets meaning twelve CPUs. All of the computing machines are interconnected in a local network, in order to communicate with each other. The first CPU, called termi7 is the machine selected to act as the master node. However, all four of the physical machines will work also

as worker nodes, meaning that the first master node, also works as a worker node and will host applications, since it has the capabilities for it. The rest of the physical machines are called termi8, termi9 and termi10 for identification purposes. All of the machines share the same architecture, number of cores and NUMA nodes. In each NUMA nodes, there are shared resources that up to four applications can utilize. They will be placed as pairs into the two sockets available in one NUMA node interfering with each other.

3.1.2 Cluster Setup

With the hardware topology well known and understood, the deployment and configuration of the cluster is now able to begin. The first terminal also called termi7 will serve as the master node, hosting every Kubernetes core component and allowing the cluster to take place. Then the control plane components that are created will set up the cluster. However, the deployment of a multi-node cluster that has each node on a different computing machine is challenging and not the same with deploying a local cluster on a single machine.

For that reason the Kubernetes cluster was deployed with the use of Kubespray. Kubespray is a composition of Ansible playbooks, inventory, provisioning tools, and domain knowledge for Kubernetes configuration managements tasks[11]. Kubespray allows us to run on bare metal, which is exactly the case for our cluster setup. It uses Ansible as its substrate for provisioning and orchestration.

Ansible on the other hand is an IT automation engine, which automates cloud provisioning, configuration management, intra-service orchestration and other IT needs[12]. By using an Ansible inventory, we can configure the Kubespray cluster inventory and provision how the cluster should be installed. That way and with the use of the Ansible playbook, we are able to deploy the Kubernetes cluster in its desired setup. Furthermore, the Calico plugin is installed and used via the Ansible playbook to enable network communication between the different bare metal nodes.

3.2 Kubernetes Framework

With the four-node cluster set-up and Kubernetes available and at a healthy state, the deployment of applications can begin. However, with the cluster scheduler being at a default state, the application deployment would completely ignore interference issues from application collocation. Thus, it is necessary that the scheduling algorithms be implemented via the scheduler plugins. The scheduling framework allows the creation of out-of-tree scheduling plugins that alter the behavior of the Kubernetes scheduler.

3.2.1 Scheduling Framework

The Kubernetes scheduler is a primary component of the Kubernetes control plane, and as already described, it is the component that chooses which Nodes will host the available pods that have not already been scheduled. Through filtering and scoring the feasible nodes, the scheduler will choose the best node candidate, and the Pod will be hosted on the selected Node. However, the default behavior of the Kubernetes scheduler is not interference aware and mismanages resource utilization when collocating applications. This can be altered, via the Scheduler configuration. The behavior of the scheduler can be customized by writing a custom configuration file and passing its path as a command line argument. A scheduling Profile allows the configuration of the different stages of scheduling in the Kubernetes scheduler. Each stage is exposed in an extension point. Plugins provide scheduling behaviors by implementing one or more of these extension points.

All of this is available via the implementation of the Scheduling Framework. The scheduling framework is a pluggable architecture for the Kubernetes scheduler. It adds a new set of plugin APIs to the existing scheduler. Plugins are compiled into the scheduler and the APIs allow most scheduling features to be implemented as plugins, while keeping the scheduling "core" lightweight and maintainable. The scheduling framework allows us to write custom, performant scheduler features without forking the scheduler's code. Custom scheduler can write their plugins "out-of-tree" and compile a scheduler binary with their own plugins included. A custom interference aware scheduler will make use of different extension points to create "out-of-tree" plugins and schedule the Pods in an interference aware manner.

3.2.1.1 Scheduling & Binding Cycle

Each attempt to schedule one pods is split into two phases, the scheduling cycle and the binding cycle[10]. The scheduling cycle selects a node for the pod, and the binding cycle applies that decision to the cluster. Together, a scheduling cycle and a binding cycle are referred to as a "scheduling context". Scheduling cycles are run serially, while binding cycles may run concurrently. During the scheduling cycle, there are certain extension points that the scheduling framework exposes, which control how the scheduler sorts the arriving pods, filters the feasible nodes and scores them, to choose the correct fit. After the selection of the best node candidate has been made, the binding cycle begins. During the binding cycle, a different set of extension points, allows further implementation of scheduling behaviors pre-bind, during binding time, or even post-bind. That way the available Pod is bound to the targeted Node.

3.2.1.2 Extension Points

Plugins can use one or more extension points to alter the scheduler behavior and enforce different scheduling techniques during the scheduling or the bind cycle. The extension points that the scheduling framework exposes are as follows[10]:

- **QueueSort**: These plugins are used to sort pods into the scheduling queue. A queue sort plugin essentially provides the ability to sort the queue of a concurrent influx of arriving pods, to allow Kubernetes to schedule certain pods before others. Only one QueueSort plugin may be enabled at a time.
- **PreFilter**: These plugins are used to pre-process info about the pod, or to check certain conditions that the cluster or the pod must meet. A pre-filter plugin should implement a PreFilter function, and if PreFilter returns an error, the scheduling cycle is aborted. The PreFilter plugin is called once in each scheduling cycle.
- **Filter**: These plugins are used to filter out nodes that cannot run the Pod. For each node, the scheduler will call filter plugins in their configured order. If any filter plugin marks the node as infeasible, the remaining plugins will not be called for that node. Nodes may be evaluated concurrently, and the Filter plugin may be called more than once in the same scheduling cycle.
- **PostFilter**: These plugins are called after the Filter phase, but only when no feasible nodes were found for the pod. Plugins are again called in their configured order. If any PostFilter plugin marks the node as Schedulable, the remaining plugins will not be called. A typical PostFilter implementation is preemption, which tries to make the pod schedulable by preempting other Pods.
- **PreScore**: This is an informational extension point for performing pre-scoring work. Plugins will be called with a list of nodes that passed the filtering phase. A plugin may use this data to update internal state or to generate logs/metrics.
- **Scoring**: These plugins have two phases:
    1. The first phase is called “score” which is used to rank nodes that have passed the filtering phase. The scheduler will call Score of each scoring plugin for each node.
    2. The second phase is the “score normalization” phase, which is used to modify scores before the scheduler computes a final ranking of Nodes, and each score plugin receives scores given by the same plugin to all nodes in “normalize scoring” phase. It is called once per plugin per scheduling cycle right after the “score” phase.

  The output of a score plugin must be an integer in range of [**MinNodeScore, MaxNodeScore**]. If not, the scheduling cycle is aborted. This is the output after running the optional NormalizeScore function of the plugin. If NormalizeScore is

not provided, the output of Score must be in this range. After the optional NormalizeScore, the scheduler will combine node score from all plugins according to the configured plugin weights. If either Score or NormalizeScore returns an error, the scheduling cycle is aborted.

- **Reserve:** A plugin that implements the Reserve extension has two methods, namely Reserve and Unreserve, which back two informational scheduling phases, with the same names respectively. Plugins, which maintain runtime state (also known as "stateful plugins", should use these phases to be notified by the scheduler when resources on a node are being reserved and unreserved for a given Pod.

  The Reserve phase happens before the scheduler actually binds a Pod to its designated node. It exists to prevent race conditions while the scheduler waits for the bind to succeed. The Reserve method of each Reserve plugin may succeed or fail. If one Reserve method call fails, subsequent plugins are not executed and the Reserve phase is considered to have failed. If the Reserve method of all plugins succeeds, the Reserve phase is considered to be successful and the rest of the scheduling cycle and the binding cycle are executed.

  The Unreserve phase is triggered if the Reserve phase or a later phase fails. When this happens, the Unreserve method of all Reserve plugins will be executed in the reverse order of the Reserve method calls. This phase exists to clean up the state associated with the reserved Pod.
- **Permit:** These plugins are used to prevent or delay the binding of a Pod. A permit plugin can do one of three things.
  1. **Approve**: Once all permit plugins approve a pod, it is sent for binding.
  2. **Deny**: If any permit plugin denies a pod it is returned to the scheduling queue. This will trigger the Unreserve method in the Reverse plugin.
  3. **Wait**: If a permit plugin returns "wait", then the pod is kept in the permit phase until a plugin approves it. If a timeout occurs, wait becomes deny and the pod is returned to the scheduling queue, triggering the unreserved method in the Reserve phase.

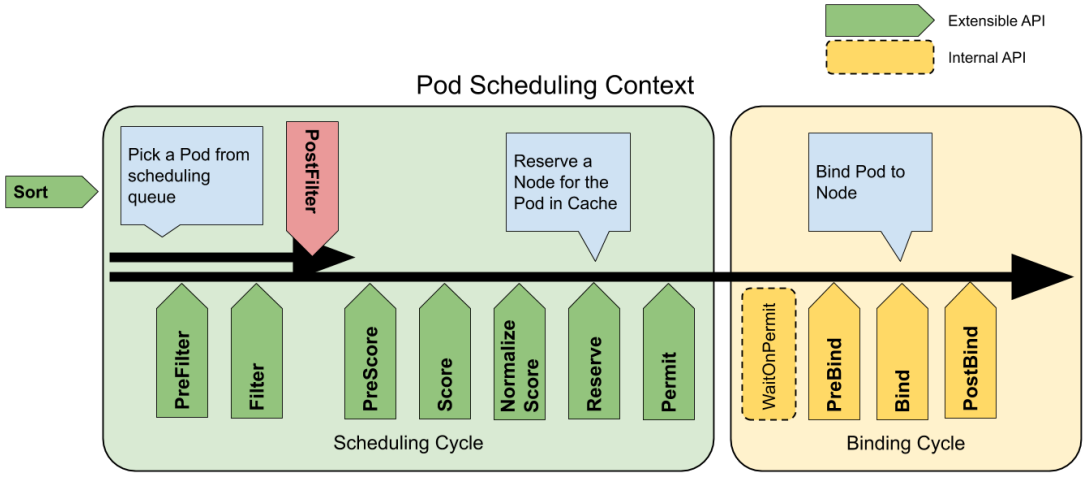
  Permit plugins are executed as the last step of a scheduling cycle, however waiting in the permit phase happens at the beginning of a binding cycle, before PreBind plugins are executed.

  While any plugin can receive the list of reserved pods from the cache and approve them, via the FrameworkHandle, we expect only the permit plugins to approve the binding of reserved Pods that are in a "waiting" state. Once a pod is approved, it is sent to the pre-bind phase.
- **PreBind**: These plugins are used to perform any work required before a pod is bound. For example, a pre-bind plugin may provision a network volume and mount it on the target node before allowing the pod to run there. If any PreBind

plugin returns an error, the pod is rejected and returned to the scheduling queue.

- **Bind:** These plugins are used to bind a pod to a Node. Bind plugins will not be called until all PreBind plugins have completed. Each bind plugin is called in the configured order. A bind plugin may choose whether to handle the given Pod. If a bind plugin chooses to handle a Pod, the remaining bind plugins are skipped.
- **PostBind:** This is an informational extension point. PostBind plugins are called after a pod is successfully bound. This is the end of a binding cycle, and can be used to clean up associated resources.

Based on the aforementioned extension points, the scheduling Cycle can be summarized to the following graphic:



Scheduling Plugin Context Cycle

3.2.1.3 Plugin API

There are two steps to the plugin API. First, plugins must register and get configured, then they use the extension point interfaces mentioned.

Most plugin functions will be called with a **CycleState** argument[10]. A CycleState represents the current scheduling context. It will provide APIs for accessing data whose scope is the current scheduling context. Because binding cycles may execute concurrently, plugins can use the CycleState to make sure they are handling the right request. The CycleState also provides an API that can be used to pass data between plugins at different extension points. Multiple plugins can share the state or communicate via this mechanism. This state is preserved only during a single scheduling context and the data available through a CycleState is not valid after a scheduling context ends, so plugins should not hold references to that data longer than necessary. It is worth noting that plugins are assumed to

be trusted. The scheduler does not prevent one plugin from accessing or modifying another plugin's state.

While the CycleState provides APIs relevant to a single scheduling context, the **FrameworkHandle** provides APIs relevant to the lifetime of a plugin[10]. This is how plugins can get a client and, or read data from the scheduler's cache of cluster state. The handle will also provide APIs to list and approve or reject waiting pods. Frameworkhandle provides access to both the Kubernetes API server and the scheduler's internal cache. The two are not guaranteed to be coordinated and extreme care should be taken when writing a plugin that uses data from both of them. Providing plugins access to the API server is necessary to implement useful features, especially when those features consume object types that the scheduler does not normally consider. Providing a SharedInformerFactory allows plugins to share caches safely.

Each plugin must define a constructor and add it to the hard-coded registry. After the plugin is registered, the scheduler uses its configuration to decide which plugins to instantiate. If a plugin registers for multiple extension points, it is instantiated only once. After the plugin is registered, there are two types of concurrency that should be considered. A plugin might be invoked several times concurrently when evaluating multiple nodes and a plugin may be called concurrently from different scheduling contexts. In the main thread of the scheduler, only one scheduling cycle is processed at a time. Any extension point up to and including permit will be finished before the next scheduling cycle begins. After the permit extension point, the binding cycle is executed asynchronously. This means that a plugin could be called concurrently from two different scheduling contexts, if at least one of the calls is to an extension point after permit. Stateful plugins should take care to handle such situations. Finally, the Unreserve method in Reserve plugins may be called from either the main thread or the bind thread, depending on how the pod was rejected.

3.2.1.4 Plugin Configuration

The scheduler's component configuration will allow plugins to be enabled, disabled, or otherwise configured. Plugin configuration is separated into two parts.

1. A list of enabled plugins for each extension point (and in the order they should run in). If one of these lists is omitted, the default list will be used.
2. An optional set of custom plugin arguments for each plugin. Omitting config arguments for a plugin is equivalent to using the default config for that plugin.

Extension points organize the plugin configuration. A plugin that registers with multiple points must be included in each list accordingly. When specified, the list of plugins for a particular extension point are the only ones enabled. If an extension point is omitted

from the config, then the default set of plugins is used for that extension point. Furthermore, plugins may receive arguments from their config with arbitrary structure. Because one plugin may appear in multiple extension points, the config is in a separate list of PluginConfig[10].

The scheduling framework allows a set of use cases to be implemented.

1. Plugins can allow a functionality of co-scheduling applications. For arriving Pods in a batch, such plugin would accumulate pods in the permit phase by using the wait option. Because the permit stage happens after reserve, subsequent pods will be scheduled as if the waiting pod is using those resources. Once enough pods from the batch are waiting, they can all be approved at the same time allowing co-scheduling to happen.
2. Plugins can also implement Dynamic Resource Binding, meaning that Topology-Aware Volume Provisioning can be implemented as a plugin, that registers for filter and pre-bind extension points. At the filtering phase, the plugin can ensure that the pod will be scheduled in a zone, which is capable of provisioning the desired volume. Then at the PreBind phase, the plugin can provision the volume before letting the scheduler bind the pod.
3. Last but not least, the scheduling framework allows the creation of custom, performant scheduler features without forking the scheduler's code, that way keeping the scheduling "core" lightweight and maintainable. That way the custom plugins are enabled as normal plugins in the scheduler config, allowing the implementation of different scheduler features. One certain example is the subject of this thesis, aiming to create custom plugins that alter the scheduler behavior to become interference-aware when scheduling applications on physical servers.

The scheduling framework is expected to be backward compatible with the existing Kubernetes scheduler. As a result, it is logical that all existing tests of the scheduler pass during and after the framework is developed.

1. Unit Tests. Each Plugin developed for the framework is expected to have its own unit tests with reasonable coverage.
2. Integration Tests. As we build extension points, we must add appropriate integration tests that ensure plugins registered at these extension points are invoked and the framework processes their return values correctly. If a plugin adds a new functionality that didn't exist in the pas, it must be accompanied by integration tests with reasonable coverage.
3. End-to-end tests. These tests should be added for new scheduling features and plugins that interact with external components of

Kubernetes such as the API server. They are however not needed, when integration tests provide adequate coverage.

3.2.2 Pods Template

Before an interference-aware plugin is analyzed, a better understanding of the applications used is necessary. As already described, the benchmarks are containerized into Docker components, and via Kubernetes, they can be scheduled as Pods and handled by the scheduler inside a server farm. However, for such benchmark applications to be run correctly, a Pod template needs to be established, that will dictate how they are to be de described in a default manner. The Pod template is as follows:

```
13 lines (13 sloc)  |  243 Bytes
1   apiVersion: v1
2   kind: Pod
3   metadata:
4     name: namd
5     labels:
6       greedy: catA
7   spec:
8     containers:
9     - image: jmmisd/testspec:1.0.0
10      command: ["/bin/bash"]
11      args: ["-c" , "source shrc" , "bin/relocate" , "runspec --config=cslab-spec-static.cfg --size=ref \
            --noreportable --iterations=3 namd" , " sleep 9999999999999999" ]
12      name: spec-ctn
13    restartPolicy: "Never"
```

Pod Template of a SPEC Benchmark

The Pod pulls the Docker image that has the SPEC Benchmark library and then calls the runspec commands, in order to run the desired benchmark under the targeted configuration. From the SPEC library, ten benchmarks are used and classified accordingly.

3.3 Application Classification

Now that the benchmarks are deployable to Kubernetes, the classification progress of each benchmark can begin. Firstly, each benchmark needs to run alone in a single socket, allowing us to document their single runtime performance. Then every single benchmark will run with each other benchmark in order to produce an NxN matrix (given that we have N total benchmarks) of results about runtime performances of collocated benchmarks.

More specifically, each script execution collocates two different SPEC benchmarks on the same CPU socket in the same node. Then the two applications are executed at the same time and for the same duration (with minimal time differences). Each runtime performance is documented to be later crosschecked with the individual runtime performance of the application.

Now with the runtime data available, we can use the whisker classification progress to classify and categorize the ten SPEC benchmarks respectively.

3.3.1 Predictors

In order to classify and have an accurate description of each Benchmark application used, it is imperative that a predictor is used. A predictor will decide if an application is contentious or sensitive, before deploying it to the cluster. By running a series of tests and recording different runtimes, a predictor algorithm is able to decide the category of an application before it is even deployed. In order to classify a benchmark, a dictionary of every single runtime registered is needed, when collocated with every other possible pairing on the same socket.

Then the whisker method selects the top and bottom quartiles of that dictionary. The top quartile is at the 75% percentile whilst the bottom quartile is the 25% percentile from the available measurements. Then the predictor calculates the whisker limit based on those two calculations and then compares it to the max available value. That comparison is the deciding factor on whether that benchmarks surpasses that limit and is indeed contentious or sensitive.

3.3.2 Classification Process

In order to classify the applications, the whisker produced from the execution of the method is compared to a QoS standard, which has been set by us. The QoS limit that was established on factoring one application's sensitivity and contentiousness is 20% or numerically 1.2. The whisker result of a SPEC benchmark application above 1.2 regarding the Sensitivity matrix, classifies the application as a sensitive one. On the other hand a whisker result above 1.2 regarding the Contentiousness matrix, classifies the application as a contentious one. That way we can derive the two characteristics of our benchmarks and conclude on which category one application is classified to. More elaborately, the script implements the following case scenarios to decide and tag the applications depending on their characteristics:

- If an application has a sensitivity whisker of 1.2 or higher, then it is classified as a sensitive one.

- If an application has a contentiousness whisker of 1.2 or higher, then it is also classified as a contentious one.

This set of data classifies the application to **category D** (or **catD**) and the appropriate tag is placed upon every single newly created Pod that executes that specific benchmark.

Accordingly, the sensitivity and contentiousness whiskers will classify each and every benchmark to their respective category depending on the whisker value surpassing the QoS value set by us. An average and standard QoS value that is largely used is the QoS of 1.2, meaning that an increase in more than 20% of runtime performance is enough to classify the benchmarks as contentious or sensitive, according to the current classification process.

This resulted in the following classification matrix:

Benchmark	Category
Zeusmp	catC
Namd	catA
Gobmk	catA
Omnetpp	catB
Perlbench	catB
cactusADM	catA
libquantum2	catD
GemsFDTD	catD
Libquantum	catD
soplex2	catD

3.4 Algorithms

With the cluster set-up to a stable and empty state, and with the benchmarks containerized and classified, it is time that the algorithmic solutions of application distributing are thoroughly described. These algorithms will then be compared to the standard and non-interference-aware way to analyze the improvement applied to application runtime performance. All of these algorithms are implemented with the help of

the scheduling framework providing an easy, deployable, and easily maintained way of adding interference awareness to the Kubernetes scheduler.

The three algorithmic solutions are as follows:

1. The Greedy Algorithm
2. The Socket-Based Algorithm
3. The Sparing Algorithm

3.4.1 Interference Impact

A random scheduler will allocate applications to nodes after checking that the resource requirements are met. That way it seemingly ensures that applications have enough resources to run in an efficient way and meet the service requirements. However, Kubernetes, lacking interference awareness, will still fail to meet the service requirements set by the client and as a result, applications will have severe performance issues. Collocated applications may have enough resources available from the common resource pool on the node, but will still have contentiousness issues, battling to achieve full resource utilization when they run concurrently. This is the reason behind researching new algorithmic solutions that utilize applications categorized depending on sensitivity and contentiousness.

When collocating applications on a socket, the shared CPU resource pool comes under stress and the runtime performance of applications is hindered. This can cause a performance drop that can exceed the SLOs set from the client. This creates two issues that require solution. The first issue resides on applications scheduling to the cluster nodes. Before applications are even hosted on CPU sockets, Kubernetes decides the node on which they will run on. In a small-scale test of quadruples arriving to each node, there can be quadruples that are inherently considered a bad set and will cause SLO violations regardless of how they are placed in sockets. That requires that algorithmic solutions are established and implemented onto the Kubernetes scheduler in an effort to avoid such cases from happening.

3.4.2 Algorithmic Approach

When scheduling applications on nodes, the Kubernetes scheduler will run each plugin in the selected order to evaluate and decide which node is the best fit. Each algorithmic solution presented in this paper, is based on a principle of filtering and scoring the available nodes. The filtering function is mainly used to ensure the even distribution between applications and to allow us to run standardized small workloads on our 4-node cluster. After the filtering process has ended, the scoring plugin will rank the available nodes and choose the best fit for our Pod, depending on its category. In a simplified explanation,

the scoring function will give a perfect score on every empty node and accordingly score every other node that is hosting applications already. The scoring function is dependent and hard-coded on the algorithmic technique implemented.

When a new pod arrives at the scheduler, the scoring plugin reads the tag provided by the Pod description template and calculates the total score that each node already has. If a node is empty, that node is given the best score and is as a result, chosen as the best fit. If two or more nodes are empty, then the selected node is chosen randomly. The same applies to nodes with the same score throughout the scoring process. The scoring plugin also utilizes a reversing function during Normalization. As already explained, Score Normalization modify scores before the scheduler computes a final ranking of the nodes. In this case, the plugin reverses the scoring provided from the Scoring function. This is mandatory because the scoring plugin provides the score depending on how many applications are hosted. Each application on a node, regardless of category has a positive weight and as a result the more applications on a server, equate to a higher score. However, this is the opposite of what is expected. An empty node would receive the lowest score of 0, when it should be instead the best fit for every arriving Pod. As a result, each application provides a positive weight when its score is calculated, and during the NormalizeScoring phase, the ranking is simply reversed.

Thus, the final score of each node is correct after the NormalizeScoring phase, and the best-fit node is selected for the Pod. The scoring plugin is different on every algorithmic solution and provides different results that are thoroughly observed in the following sections.

3.4.3 CPU Pinning

After the scheduler has distributed the Pods accordingly, depending on which algorithm was enabled, the process of CPU pinning applications begins. The set-up environment that is used, utilizes two CPU sockets on each node (physical machine). Thus, each quadruple of applications arriving at a node will be randomly split into the two sockets. This can still create not only wasteful pairs, where sockets remain un-utilized, but also create contentious pairs that battle for resources and cause severe performance issues. As a result, a daemon, which could also be described as an internal scheduler, is needed to monitor and manage how the arriving applications will be split. Each node needs a running daemon, implementing a simple utilization pairing algorithm to ensure that the best pairs possible are created on each socket. The DaemonSet that runs on each Node uses an entry-point via the GoLang client, to periodically request the list of Pods on a specific namespace. This is accomplished with the use of the Kubernetes API.

In order to effectively change the socket selection of a Pod, the DaemonSet then finds and utilizes cgroups. Cgroups also known as Control Groups provide a mechanism for aggregating and partitioning sets of tasks, with all their future children, into hierarchical

groups of specialized behavior. More specifically, cgroup's filesystem allows us to change the set of cores that are used to run a Pod. Each socket has certain specific cores that it utilizes, and with the use of cgroup's files, we can pin each Pod to the specific set of CPU cores we wish to do so.

3.4.4 The Greedy Algorithm

The first algorithm that was implemented to the scheduling framework is the Greedy algorithm. As its name suggests they course of action that this plugin implements, is to greedily assign as many non-contentious pods together in the early quadruple, so that to minimize the chance of bad pairs forming in sockets. That way, sensitive applications are more likely to be placed with non-contentious ones and as a result display minimal performance degradation. However, since the algorithmic approach is in fact greedy, there can still be certain quadruples formed which contain many contentious applications. Nodes hosting such formations will have increased interference and as a result cause heavy performance issues to applications hosted there.

3.4.4.1 Concept

The algorithm applied resides on the following principle:

Algorithm:

Input: N applications in set

Outputs: N/2 pairs per socket

Sets <- {1. non-contentious/insensitive,

2.non-contentious/sensitive,

3.Contentious/insensitive,

4.Contentious/sensitive}

Quiets <- Non-Contentious or insensitive (1,2,3)

Not quiets <- The rest (4)

For **set** ∈ **sets** do

If set arriving is **quiet** then

If server is empty then

- Place set

If server is not empty then

○ Pair set with other quiet till there is a quadruple formed (Preferring 1s, then 2s and lastly 3s)

If set is **Not quiet** then

○ Add the set to a new empty socket or pair it randomly in a socket if no server is empty.

Complexity: O(n)

3.4.4.2 Plugin

The Greedy plugin will only track applications that have a tag describing their contentiousness and sensitivity. Any Pod arriving to the scheduler that does not have a tag will be ignored and be handled with the default behavior.

Every plugin that has been created firstly filters the nodes in order to allow only quadruples to be formed and utilizes a hard-coded scoring function that simply calculates the score of each node based on the applications running on it. If it is empty, it has a score of zero and if it has applications, it calculates the total score depending on how many and which type of Pods are hosted there. Then during the normalization phase, that scoring is reversed and as a result, the node with the lowest score (or zero) is the one with the highest, and as a result chosen as the best fit.

3.4.4.3 Scheduling Process

The scheduling process that the Greedy algorithm instantiates when a new Pod arrives at the cluster is a hard-coded scoring-based function.

The Greedy algorithm has a simple scoring function that allocates a low score to every non-contentious and insensitive app, also known as Category A. Applications under category B (which are sensitive but non-contentious) have a slightly higher score and Pods with the tag of category C (contentious and insensitive) have an even higher one. Lastly applications under category D, which are both contentious and sensitive have the highest scoring as they will ruin pairs in nodes, contradicting the very principle under which the Greedy algorithm works on.

As a result, during every scheduling cycle the plugin will implement the following path of execution:

- Recognize the arriving Pod and check for the available tag, indicating that the pod is to be tracked by the plugin.
- Filter the available nodes in order to disqualify full ones.

- Score the feasible nodes given the greedy algorithmic principle.
- Category A receives the lowest score when paired regardless with anyone, category B the second lowest and so on.
- Normalize the scoring and choose the best fit for the new application

After the scheduling process is completed, the CPU-pinning DaemonSet will get a list of all deployed Pods in the node with the CPU socket they are pinned at, and will decide if the application is placed correctly. If not, Pods will be swapped around the available CPU sockets, in order to minimize contentiousness issues.

3.4.5 The Sparing Algorithm

The second algorithm that was implemented to the scheduling framework is the Sparing algorithm. This algorithm similarly to the Greedy one, tests a more radical and greedy approach by inherently understand the impossibility of having perfect runtime performances for all hosted applications, and tries as a result to pair inherently bad applications together that way sparing the rest of the applications from being very affected. More specifically, applications that fall under category D, meaning applications that have both contentiousness and sensitivity, will be paired together and as a result leave the rest of the applications free to pair each other at the rest of the nodes. The result of such an algorithmic approach is to create a similar result with the Greedy algorithm but instead trying to achieve good results with the exact opposite approach.

3.4.5.1 Concept

Algorithm:

Input: N applications in set

Outputs: N/2 pairs per socket

Sets <- {1. non-contentious/insensitive,

2.non-contentious/sensitive,

3.Contentious/insensitive,

4.Contentious/sensitive}

Quiets <- Non-Contentious (1,2)

Not quiets <- Contentious (3,4)

For **set** ∈ **sets** do

If set arriving is **not quiet** then

If server is empty then

- Place set

If server is not empty then

- Pair set with other **not quiet** till there is a quadruple formed (Preferring 4s and lastly 3s)

If set is **quiet** then

If server is empty then

- Place set

If server is not empty then

- Pair set with other **quiet** till there is a quadruple formed (Preferring 1s and lastly 2s)

Complexity: O(n)

3.4.5.2 Plugin

The Sparing plugin works with the exact same mechanics as the rest of the plugins developed. The key difference is apparent at the scoring function of the plugin, where the lowest scores are set when pairing applications of the category D together. Every other pair has a linear approach following the principles of the sparing algorithm concept described above.

As a result, during the normalization phase later, we will have two main scenarios folding out. Firstly, if the arriving application is of class category D or C, and thus considered a bad contentious pairing component, the scheduler will try to bind it in the most clustered node, given that there are no empty ones. The second scenario unfolds when applications of category A or B arrives. These applications will try to create pairs together if such an available spot is available. Thus, sensitive applications are rarely placed with contentious ones, reducing SLO violations as much as possible.

3.4.5.3 Scheduling Process

The sparing algorithm follows another simple hard-coded process of deciding the best node for every arriving Pod.

More specifically the scoring plugin implements a linear scoring function. Every arriving application under category D, has a very low score (and as a result during normalization phase a very high score, only second to an empty node) and so does every Pod under category C, with a slightly higher score. Then applications that are non-contentious receive a very high score unless when paired together. Then, they are similarly scored with a very low total in order to be paired together. Consequently, the scheduling process is as follows:

- Recognize the arriving Pod and check for the available tag, indicating that the pod is to be tracked by the plugin.
- Filter the available nodes in order to disqualify full ones.
- Score the feasible nodes given the sparing algorithmic principle.
- If the arriving application is a contentious one, (meaning either category C or D) then pair it with another contentious application.
- If the arriving application is a non-contentious one (meaning category A or B) then it is also paired with another non-contentious application.
- Schedule the Pod to the chosen node.

After the scheduling process is completed, the CPU-pinning DaemonSet will again abide by the same rules of fitting applications in the most preferred way applicable.

3.4.6 The Socket-Based Algorithm

Finally yet importantly, there is a different approach in pairing applications together. This algorithm is based on removing the wasteful behavior showcased by other algorithms and tries to get good pairings between applications. This is possible by taking advantage of the CPU-pinning DaemonSet that was created to split arriving applications on the node, into the different sockets.

3.4.6.1 Concept

Before explaining the concept behind this application, it is imperative that pairing between different categories is well understood.

Two characteristics describe each application. Specifically, contentiousness and sensitivity. Thus, the applications are assigned into four categories:

Categories	Application Tags	Description
Category 1	Non-contentious and insensitive	Best ones, can fit everywhere
Category 2	Non-contentious and sensitive	Optimally fit with Categories 1 and 2
Category 3	Contentious and insensitive	Can fit with its own kind, Category 3
Category 4	Contentious and sensitive	Worst ones, can fit with Category 1 only

As a result, we have these pairs in sockets:

Best: 1-2, 1-4, 2-2, 3-3

Worst: 2-3, 2-4, 3-4. 4-4

Wasteful: 1-1, 1-3

Wasteful pairings: This means that such pairings work perfectly but might cause problems on different sockets by not being resourceful enough.

Then the algorithm applied resides on the following principle:

Algorithm:

Input: N applications in set

Outputs: N/2 pairs per socket

Sets <- {1. passive/insensitive,

2.Passive/sensitive,

3.Aggressive/insensitive,

4.Aggressive/sensitive}

Quiets <- Passive or insensitive (1, 2, 3)

Not quiets <- The rest (4)

For **set** ∈ **sets** do

If set is **quiet** then

If server is empty then

- Place set

If server is not empty then

- Pair set with other quiet till there is a Best set
- If there is no best set, form a Wasteful one

If set is **Not quiet** then

- Add the set to a new empty socket or pair it randomly in a socket if no server is empty.

Complexity: O(n)

3.4.6.2 Plugin

The Socket-Based Plugin aims at minimizing wasteful pairings by allowing only specific sets of applications to fit together. More specifically, arriving Pods under category A will be fitted preferably with Pods under category D, thus allowing applications under category D to run in the best practice possible, since there is no other type of application they can be placed with avoiding performance issues. On the same note, applications under category B will prefer being placed with each other and applications under category C will do the same.

After the arriving workload has been scheduled, a set of quadruples will be hosted on each node. How pairs will be formed on the different CPU sockets is based on luck and can as a result cause a variety of different performance results. This makes the CPU-pinning DaemonSet imperative for the intended run of the plugin.

3.4.6.3 Scheduling Process

The socket-based algorithm is again based on a hard-coded scoring function that abides on a resourceful principle. Avoid creating pairs that could have been used better. A pairing of two applications, with each one being category A, wastes a chance of utilizing one application to be fitted with other sensitive ones, that need non-contentious applications with them. As a result, we have the following scheduling process.

- Recognize the arriving Pod and check for the available tag, indicating that the pod is to be tracked by the plugin.
- Filter the available nodes in order to disqualify full ones.
- Score the feasible nodes given the socket-based algorithmic principle.

- If the arriving application is of type A, its score will be incredibly low against contentious applications to create good fits.
- If the arriving application is of type B, its score will be incredibly low against its own category, but very high with every other application in a linear manner (from category A to D).
- If the arriving application is of type C, preferably score it with itself and if not possible, pair it with other applications from category A.
- Lastly, if the arriving Pod is of type D, it will only have a low score when paired with applications from category A.
- Schedule the Pod to the chosen lowest-scored node.

After the scheduling process is completed, the CPU-pinning DaemonSet will again abide by the same rules of fitting applications in the most preferred way applicable, ensuring that the socket-based algorithm works as intended.

4. Solution assessment

4.1 Evaluation Process

In order to evaluate and assess how each algorithm is performing, several factors and measurements need to be examined. Each algorithm developed, will be compared to the default Kubernetes scheduler, which will serve as the base workload performance metric. In order for an algorithm to be effective and provide interference awareness to the scheduler, an improvement has to be seen in comparison to the default, random way of scheduling applications. The default scheduler will assign applications randomly, from an interference perspective, allowing us to measure every performance runtime of the scheduled applications. The runtime of each benchmark is registered and compared to different SLOs

Each workload will record several if any, violations when exceeding the Service Level Agreement (SLA) set between the service provider and the customer. The service provider will set several Service Level Objectives (SLOs) to try to evaluate how its applications are performing. Each application will have its runtime divided by its default one, providing a numerical value that is later compared with the SLOs set by the service provider. If the numerical value is greater than the SLO, then we have a violation, with each violation hindering the efficiency score of the algorithm.

In order to efficiently measure the algorithmic performance of each plugin, every out-of-tree scheduler will be stressed under the same number of workloads, varying in contentiousness and load.

4.1.1 Service Level Objectives

Whilst the increase in runtime performance of applications, is a metric that showcases interference between applications, the performance of such applications may not be hindered enough to cause customer discontent. This results in an agreement, between the service provider and the customer, in order to better understand and evaluate how applications are performing. That SLA can be about the application uptime, the response time, or about the application performance. In our case, every SLO is focused around application performance. Each application is running and trying to fulfill a task in a set amount of time. An increase in runtime performance can violate the SLO set by the service provider, and dissatisfy the customer. For our measurements, three SLOs were set, at 1.1, 1.2 and 1.3. Respectively the three objectives will present a violation if an application has an increase in runtime performance of 10%, 20% or 30%.

4.1.2 Runtime Violations

The number of violations registered for each plugin and as a result each algorithm, is providing us with the efficiency of that algorithm under a workload. The violation is filed only if the target SLO is exceeded. Given that three different SLOs have been set to better test and evaluate the algorithms, that runtime performance will be compared with the entirety of the target SLOs.

4.1.3 Benchmark Runtimes

Respectively the three objectives will present a violation if an application has an increase in runtime performance of 10%, 20% or 30%. That increase is compared to the default benchmark runtime that was registered before the workloads were applied and each application was running isolated on a socket. In order to calculate the benchmark runtimes correctly during a workload run, all applications on the same node need to be run concurrently and for the same time. Since the benchmarks have different default runtimes, they were run recursively in different batches to have a total runtime of around the same volume. For example, the benchmark soplex has a default runtime of 323 seconds whereas the cactusADM benchmark has a default runtime of about 1333 seconds. Dividing these numbers will provide us with the approximate ratio of running the applications concurrently, which is four (4) for this example. This means that in order to concurrently run the two benchmarks and receive correct measurements, soplex will need to run four times more than cactusADM. The same logic applies to every concurrent run of benchmarks.

4.1.4 Workloads

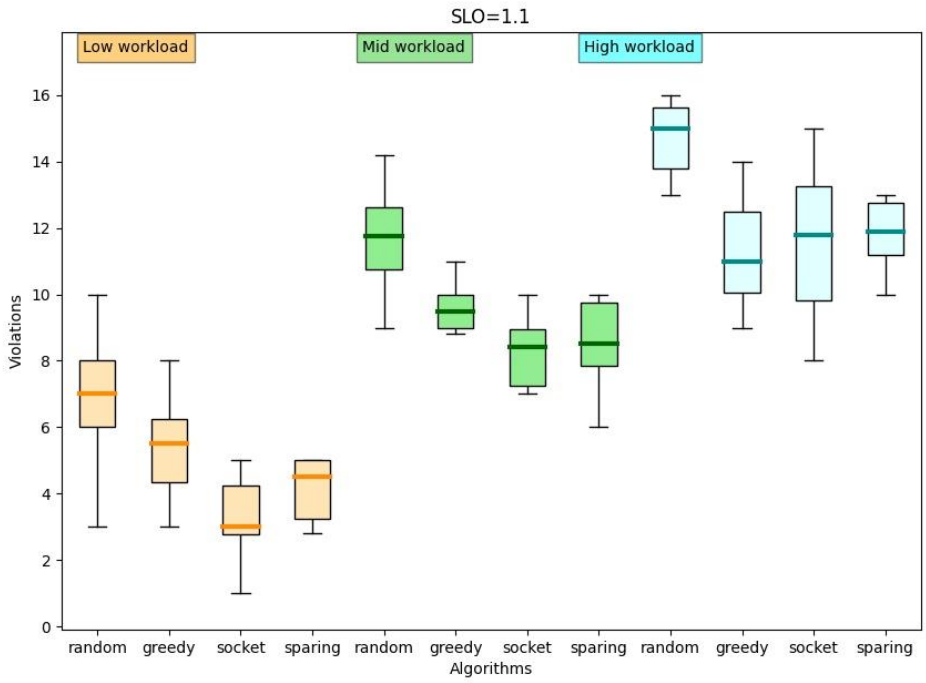
To have a concrete image of how plugins are effectively changing the interference awareness efficiency of the Kubernetes scheduler, different sets of workloads need to be run and evaluated. Thus, several workloads were created which could be roughly be classified into three different major categories.

The first category of workloads is the Heavy-Workload one. This category has many contentious applications that cause a lot of interference to collocated applications and stress the resources to heavy amounts. Another type of workload category is the Medium-Workload one that has a balanced amount of contentious and non-contentious applications, in order to evaluate the algorithms under an average stress test. Lastly but not least, all algorithms are also stressed with Low intensity workloads, namely under the category Low-Workload. Such workloads cause little interference even when randomly scheduled and allow us to compare different algorithms in low stress scenarios.

4.2 Algorithmic Assessment

In the following section, each algorithm is stressed under many different workloads, evenly chosen between low, medium and heavy intensity in order to register the number of violations happening every run. The amount of violations happening is also dependent on the SLO objective set at the time. Then these violations are analyzed with the use of boxplots, providing us with an average metric in order to compare algorithms sufficiently.

4.2.1 First Service Level Objective



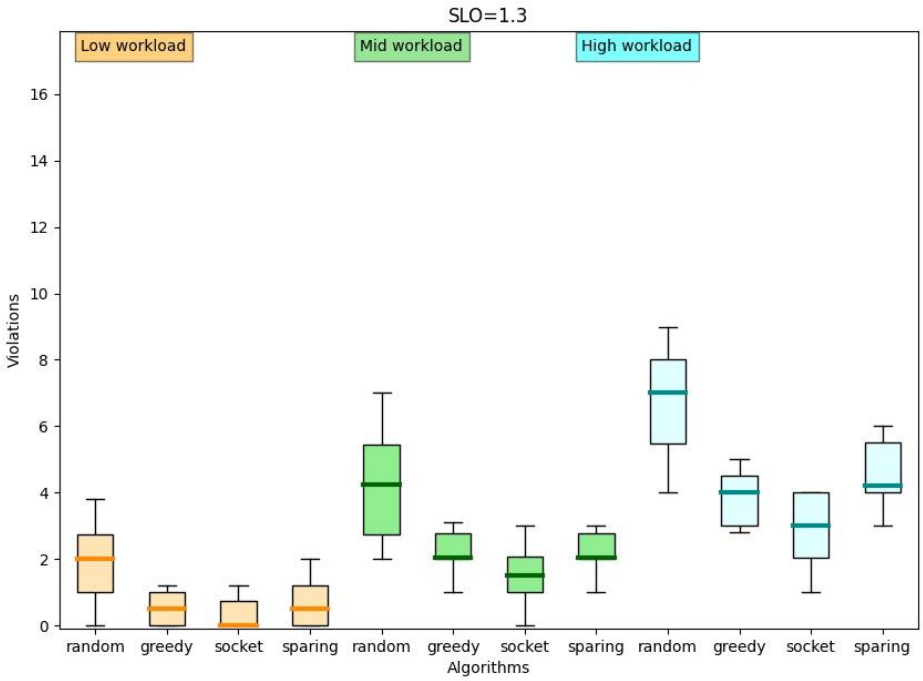
Algorithms under SLO of 1.1

Under an SLO of 1.1, there are the following violations:

Here we notice that when we have an SLO that is strict, which requires that applications do not exceed the 10% increase in performance, the overall best performer is the Socket-Based algorithm. However, when the arriving workload is a High intensity one, the Greedy algorithm seemed to have surpassed the others in efficiency. That is expected since having few non-contentious applications and greedily pairing them together, will reduce potential violations significantly more. In lower intensity workloads though, the Socket-Based algorithm provided the least amount of violations.

4.2.2 Second Service Level Objective

Under an SLO of 1.2, there are the following violations:



Algorithms under SLO of 1.2

On a more conservative approach of an SLO of 1.2 (or 20%), we notice similar efficiency from all algorithms. Every algorithm significantly outperformed the default scheduler and reduced violations regardless of the provided workload. In the scenario where the arriving workload is either pretty balanced or has little intensity (Low and Mid workloads), we noticed that the Greedy algorithm failed to reduce violations as much as the other two, though not for a significant amount. That is because the greedy nature of that algorithm, as its name suggests, will sometimes create wasteful pairings that could have been utilized in a more specialized manner avoiding the creation of wrongful pairs.

4.2.3 Third Service Level Objective

Algorithms under SLO of 1.3

Under an SLO of 1.3, there are the following violations:

When the Service Level Objective is rather loose and allows violations to reach up to 30% of an increase in runtime performance, we notice that the balanced Socket-Based approach performs in incredible fashion. By allocating applications in a selective manner and forming quadruples of similar contentiousness, the socket-based algorithm, even without any CPU pinning, allocated applications to Nodes in order to allow the formation of only acceptable pairs. Even in cases where that did not succeed, the very loose SLO, did not register enough violations, since all nodes had evenly distributed contentiousness.

4.2.4 CPU-pinning Process

Whilst algorithms have significantly improved the performance of hosted applications, the applications are randomly hosted on the CPU-sockets. For that exact reason, the CPU-pinning process must be enabled allowing us to choose how applications will be paired. More specifically the DaemonSet responsible for managing applications on each node will map the arriving applications into several hard-coded checks.

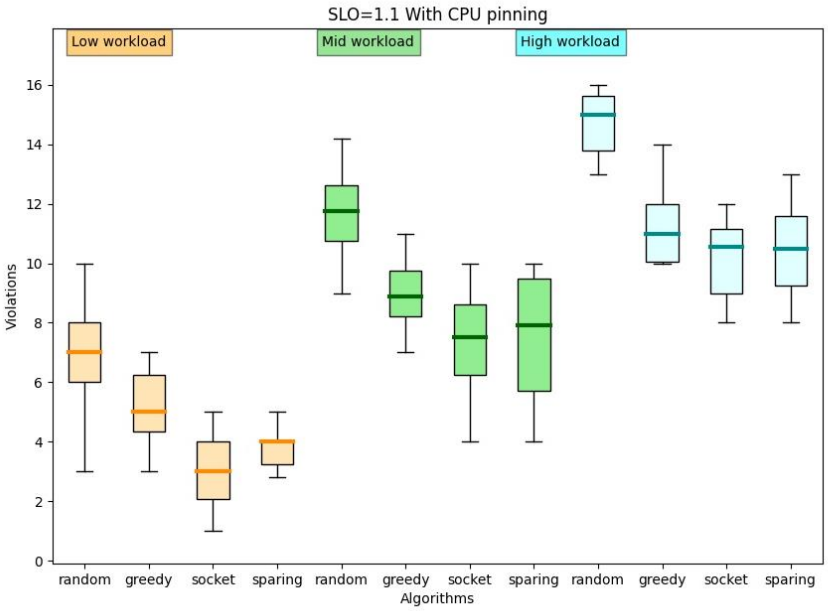
- Get the list of applications running collocated on the same CPU-socket.
- Check if the pairing that was already randomly created is sufficient and causes no interference issues. Allowed pairings fall under these categories:
    - ❖ catA: No restrictions.
    - ❖ catB: 1) Alone, 2) with catA, 3)with cat B

- ❖ catC: 1) Alone, 2) with catA, 3) with cat C
- ❖ catD: 1)Alone, 2) with catA

- If the current state of the node is not the aimed one, start implementing each category rule.
- If one of the above restrictions cannot be met, skip it.
- If none of the above restrictions can be met, then pair the applications randomly.

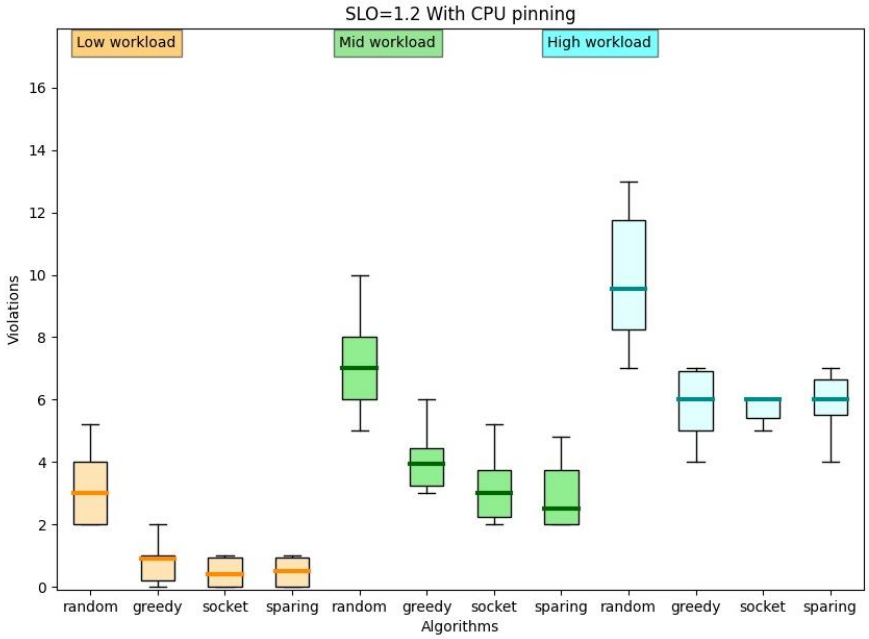
The CPU-pinning process allowed us to improve the performance of each algorithm by up to 27%. This is showcased in the following boxplots.

SLO 1.1:



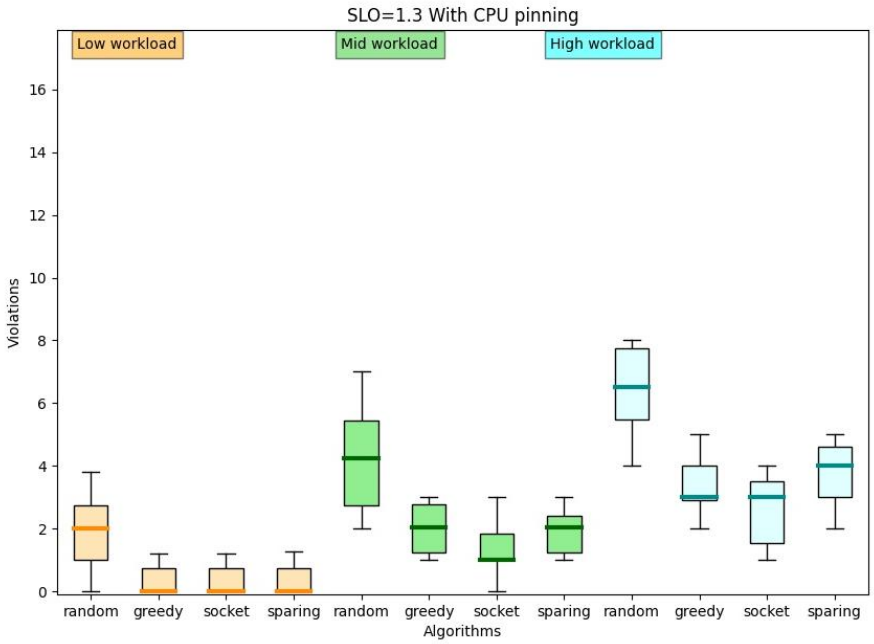
SLO 1.1 with CPU-pinning

SLO: 1.2



SLO 1.2 with CPU-pinning

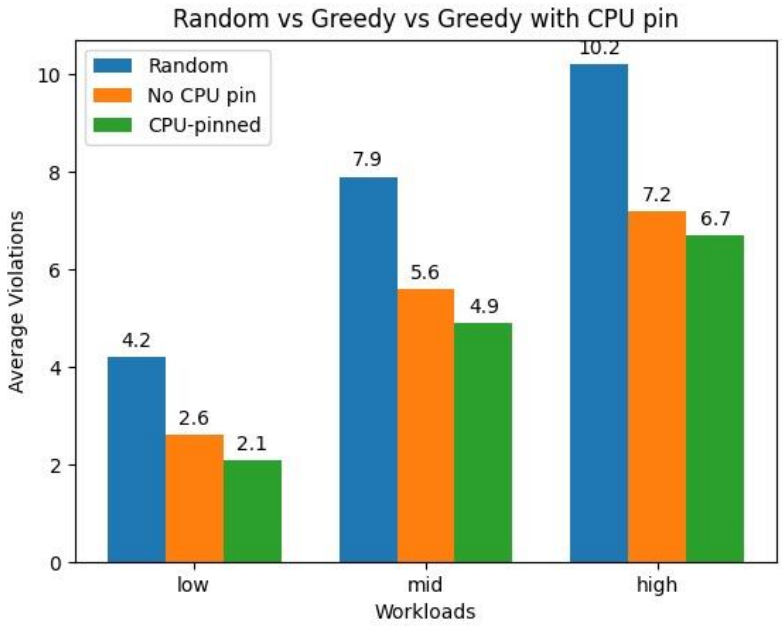
SLO: 1.3



SLO 1.3 with CPU-pinning

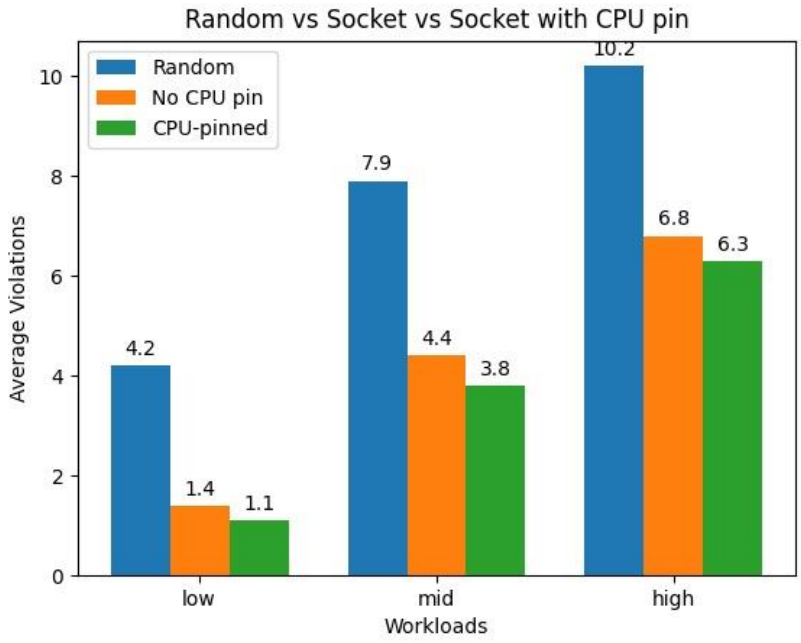
The difference and improvement that was delivered from the CPU pinning is better explained in the following diagrams that compare every algorithm with itself (with and without CPU pinning) and with the default scheduler.

GREEDY ALGORITHM



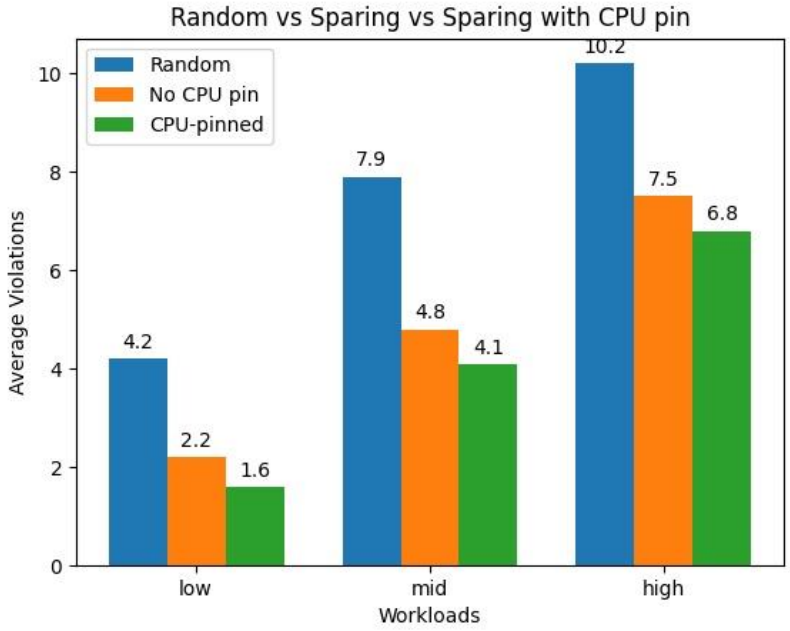
Greedy Algorithm Results

SOCKET-BASED ALGORITHM



Socket-Based Algorithm Results

SPARING ALGORITHM



Sparing Algorithm Results

4.2.5 Proposed Solution

The most effective way of scheduling applications is by implementing the socket-based algorithm with CPU-pinning. The algorithm avoids creating wasteful quadruples and manages to efficiently schedule applications to the available nodes. Then the CPU-pinning Daemon-Set that is present on each node, allocates the CPU resources accordingly to pairs of applications hosted on each node. As it is obvious by the diagrams before, the Socket-Based algorithm with CPU-pinning provides the least amount of violations for every workload type and every SLO.

5. Conclusion

After many tests, classification processes, algorithmic implementations and application CPU pinning, a major breakthrough is established in extending Kubernetes. Kubernetes has lacked any interference awareness in its existence, and now we are able to create a cloud-native way of adding that as an extension. The algorithms via the scheduling framework, alongside with a CPU-pinning DaemonSet have reduced application interference up to the point of a 74% reduction in SLO violations (Socket-based algorithm with CPU pinning, when stressed under low-stress workloads). Applications have better runtimes due to resourceful application collocation. Even though the workloads were of a moderate size, and the test environment was rather small and limited to certain sockets, the whole project is a first and vital step into creating a cloud-native, Kubernetes-extensible way of have interference aware systems that utilize resources and reduce costs for cloud providers.

In a larger cloud-based environment with different types of applications, a different and more sophisticated approach would be required. However, the core principle would still be the same in extending Kubernetes and making it interference-aware. The scheduling framework will schedule classified application into the available worker nodes and a CPU-pinning internal scheduler will schedule them to the available CPU sockets. That way, application co-location will no longer be an issue regardless of the infrastructure that hosts the arriving workloads.

6. Related Work

6.1 Intel CPU Manager

One interference aware system for Kubernetes that has been developed in a cloud-native way, is the Intel CPU Manager[3]. When using CPU-intensive workloads, the pods hosted on a cloud-based system can become throttled if there is a limited amount of CPUs available. In such a scenario, some of these pods will contend for resources available and

cause interfering issues on one another. Intel's CPU Manager uses a control called CPU affinity, which dictates which CPUs a Pod can use.

By default, all the pods and the containers running on a compute node of the Kubernetes cluster, can execute on any available cores in the system. When the CPU Manager is enabled with the static policy, it manages a shared pool of CPUs. When Kubelet creates a container, with a CPU request, CPUs for that container are removed from the shared pool and assigned exclusively for the lifetime of the container.

With the use of exclusive CPUs, each container running on the cluster does not share its resources and a better performance is expected due to isolation. In co-located workloads with aggressive applications the shared pool of resources can be stressed thus hindering the performance of hosted Pods. Intel's CPU Manager provides some interference awareness to Kubernetes by allocating CPU core to containers using the affinity metric. However, CPU pinning is not available and providing concrete core and socket selection for the Pod is impossible. Furthermore, applications cannot utilize the same cores from a socket thus potentially wasting resources. In our thesis, Pods that are not aggressive can use shared CPU cores thus not only avoiding performance degradation but also utilizing CPU quota to its fullest.

6.2 Workload Classification

Workload classification in terms of resource consumption is an active research field for many years and has as a result provided several models in classifying workloads on resource usage. Researchers like Haritatos proposed a classification process that exhibits which shared resource was pressured mostly.

In this thesis, workload classification is based on two sets of characteristics, namely contentiousness and sensitivity. These two characteristics were proposed by Tang et al. [14]as more elaborate metrics to capture the interference potential of applications. Zhao et al.[18] proposed a method to directly predict performance degradation due to interference.

In that note, we aim to classify applications with the use of the contentiousness and sensitivity tags. The metric used in classifying applications and resolving the selection of those tags is CPU usage. ActiManager firstly implemented the same process of measuring and classifying applications based on their interference. The whisker method used in this research is based on a larger classification process by ActiManager, aiming to preemptively classify applications arriving at a server[15]. Instead of quantifying these metrics, the coarser classes of quiet and not-quiet were used in the algorithmic implementations, to better analyze performing events.

6.3 Interference Aware Managers

Interference awareness has existed as a major field of research and has been implemented significantly in cloud based environments. PACMan[16] suggests the use of laboratory nodes to profile VMs and derives the potential degradation for the co-location of VMs, greatly limiting the applicability of their approach. DeepDive[17] does not rely on any prior information about the application and relies on online monitoring to identify interference. It instead, clones the VM to a sandboxed environment where it is stressed accordingly to discover potential interference. If that interference is validated, then the probe of the VM is generated and tested with potential destinations to pick the best selection.

In our approach, aside from it being an extension of Kubernetes and as a result something new to the current selection of Interference aware managers, we do not require any prior information and also we do not interfere with the production execution of the arriving workload. Instead every benchmark executed is classified using the established predictor and then handled by the scheduler accordingly, without interfering with its execution.

7. Future Work

The aim of this thesis is to be the pivotal start in creating cloud-native interference aware systems. With the use of Kubernetes and its Scheduling Framework, we were able to extend the Kubernetes scheduler into understanding how collocated applications affect and interfere with each other.

However, for this proposal to succeed a much more realistic test case needs to be examined. At first, a different test environment is required. Workloads will no longer consist of sixteen applications arriving and running concurrently, but will instead host an N amount of microservices. That amount is accordingly specified to imitate a live production system of a cloud-based hosting server farm. As a result, there is no limit in how many applications will be scheduled per Node. Instead, nodes will host applications depending on their CPU resource quota and the scoring mechanism of the scheduler. Furthermore, all Benchmarked applications were categorized via the use of the predictor before they arrived at the server farm. Such an assumption is not apparent in realistic hosting server environments. Therefore, a new kind of predictor alongside the usage of an empty Node where the application can run is needed. There, the application can run without any interference and can be classified during its execution. That way, not only do we succeed in allowing the application to run without any interruption, but we also provide a realistic classification process for newly arriving Pods.

This new reality creates the need of better monitoring. Sockets inside NUMA nodes, will need to be registered as a Custom Resource in a cloud-native way in order to utilize the Kubernetes API in monitoring and logging hosted Pods per socket. That Custom Resource Definition extends the Kubernetes API and allows us to register all running Pods per socket. Then the Kubernetes scheduler retrieves that information to decide which socket is the best available fit for each arriving Pod. Lastly, the internal DaemonSet will pin the Pods to the available socket, in a similar manner with the current research. This time however, the CPU pinning process is not a simple hardcoded function, but is instead executed based on the available information from the Custom Resource Definition describing Sockets' status.

A realistic and thorough examination of the above sentiment will revolutionize cloud-native systems allowing them to become truly interference aware and save resources reducing costs significantly. Cloud providers will be able to extend current Kubernetes-based systems in becoming interference aware, that way utilizing computing resources.

8. Bibliography

[1] intelintelVMWare Virtualization Overview. https://www.vmware.com/pdf/virtualization.pdf

[2] Docker Official Documentation. https://docs.docker.com/

[3] Intel CPU Manager. https://kubernetes.io/blog/2018/07/24/feature-highlight-cpu-manager/. 24-7-2018.

[4] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes. “Borg,Omega, and kubernetes.

[5] Chinmaya Kumar Swain, Aryabartta Sahu. Interference Aware Scheduling of Real Time Tasks in Cloud Environment. ieeexplore.ieee.org/document/8622899

[6] Kubernetes Official Documentation. https://kubernetes.io/docs/home/.

[7] Redhat Etcd. https://www.redhat.com/en/topics/containers/what-is-etcd.

[8] Calico Tigera Documentation. https://www.tigera.io/calico-documentation/.

[9] CoreDNS Official Documentation. https://coredns.io/manual/toc/

[10] Kubernetes Scheduling Framework. https://github.com/kubernetes/enhancements/blob/master/keps/sig-scheduling/624-scheduling-framework/README.md

[11] Kubespray Official Documentation. https://kubespray.io/.

[12] Ansible Official Documentation. https://docs.ansible.com/

[13] Alexandros-Herodotos Haritatos, Konstantinos Nikas, Georgios I.Goumas, and Nectarios Koziris. 2016. A resource-centric Application Classification Approach. In COSH@HiPEAC. TUM Library, 7–12.

[14] Lingjia Tang, Jason Mars, and Mary Lou Soffa. 2011. Contentiousness vs. sensitivity: improving contention aware runtime systems on multicore architectures. In Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era (EXADAPT ’11). ACM, New York, NY, USA, 12–21. https://doi.org/10.1145/2000417.2000419

[15] Stratos Psomadakis, Stefanos Gerangelos, Dimitrios Siakavaras, Ioannis Papadakis, Marina Vemmou, Aspa Skalidi, Vasileios Karakostas, Konstantinos Nikas, Nectarios Koziris, Georgios Goumas. ActiManager: A practical, interference-aware cloud resource manager. http://www.cslab.ece.ntua.gr/~vkarakos/papers/middleware19_actimanager-demo.pdf.

[16] Alan Roytman, Aman Kansal, Sriram Govindan, Jie Liu, and Suman Nath. 2013. PACMan: Performance aware virtual machine consolidation. In Proceedings of the 10th International Conference on Autonomic Computing ({ICAC} 13).

[17] Dejan Novaković, Nedeljko Vasić, Stanko Novaković, Dejan Kostić and Ricardo Bianchini. 2013. DeepDive: Transparently Identifying and Managing Performance Interference in Virtualized Environments. In Proceedings of the 2013 USENIX Conference on Annual Technical Conference (USENIX ATC’13).

USENIX Association, Berkeley, CA, USA, 219–230. http://dl.acm.org/citation.cfm?id=2535461.2535489

[18] Jiacheng Zhao, Huimin Cui, Jingling Xue, and Xiaobing Feng. 2015. Predicting cross-core performance interference on multicore processors with regression analysis. IEEE Transactions on Parallel and Distributed Systems 27, 5 (2015), 1443–1456.

9. Citation

All of the needed code is available and thoroughly explained in the following github repository.

Github.com/dimitrisdol/thesis_workspace