Εθνικο Μετσοβιο Πολυτεχνειο

Σχολη Ηλεκτρολογων Μηχανικων και Μηχανικων Υπολογιστων

Εργαστηριο Υπολογιστικων Συστηματων

Fine-Grained Container Orchestration and Scheduling On Kubernetes Clusters

Διπλωματικη Εργασια

του

Γεωργίου Στεφανάκη

Επιβλέπων: Γεώργιος Γκούμας
Καθηγητής Ε.Μ.Π.

Αθήνα, Φεβρουάριος 2025

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Τομέας Πληροφορικής
Εργαστήριο Υπολογιστικών Συστημάτων

Fine-Grained Container Orchestration and Scheduling On Kubernetes Clusters

Διπλωματικη Εργασια

του

Γεωργίου Στεφανάκη

Επιβλέπων: Γεώργιος Γκούμας
Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 1η Φεβρουαρίου, 2025.

.....	.....	.....
Γεώργιος Γκούμας	Νεκτάριος Κοζύρης	Διονύσιος Πνευματικάτος
Καθηγητής Ε.Μ.Π.	Καθηγητής Ε.Μ.Π.	Καθηγητής Ε.Μ.Π.

Αθήνα, Φεβρουάριος 2025

.....

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

Περίληψη

Στις μέρες μας, η ποικιλομορφία των εφαρμογών που φιλοξενούνται στο Cloud είναι εκτενής, περιλαμβάνοντας από εφαρμογές υψηλής επίδοσης (high-performance computing) έως αρχιτεκτονικές λογισμικού βασισμένες σε μικροϋπηρεσίες, αναλύσεις δεδομένων (data analytics) και ροές (pipelines) μηχανικής μάθησης. Καθώς το υπολογιστικό μοντέλο του Cloud επεκτείνεται συνεχώς σε κάθε σύγχρονο τομέα πληροφορικής, η ανάγκη βελτίωσης της επίδοσης και της αξιοποίησης των πόρων των υποδομών των κέντρων δεδομένων (datacenters) καθίσταται ιδιαίτερα κρίσιμη, όχι μόνο για τον τελικό χρήστη αλλά και από την άποψη της ενεργειακής και οικονομικής αποδοτικότητας. Η κιβωτιοποίηση (containerization) των εφαρμογών αποτελεί μία στρατηγική βελτιστοποίησης που προσφέρει πολλά πλεονεκτήματα σε σχέση με την προγενέστερη εικονικοποίηση (virtualization) βασισμένη σε επιβλέποντα συστήματα (hypervisors), όπως φορητότητα, αναπαραγωγιμότητα, χαμηλότερη επιβάρυνση επίδοσης και απαιτήσεις μνήμης, καθώς και ταχύτερη εκτέλεση και κλιμάκωση. Ο Κυβερνήτης (Kubernetes) είναι ένας ανοιχτού κώδικα ενορχηστρωτής για την εκτέλεση, διαχείριση και κλιμάκωση containerized εφαρμογών σε παραγωγικά περιβάλλοντα. Παρόλο που η αναβάθμιση της επίδοσης σε σχέση με τα παραδοσιακά συστήματα εικονικοποίησης είναι εμφανής, οι ενορχηστρωτές γενικά δεν βασίζονται σε λεπτομερή δεδομένα πόρων για την δρομολόγηση και την εκτέλεση εφαρμογών. Ο Kubernetes λαμβάνει υπόψη μόνο απλοϊκούς δείκτες, όπως το φορτίο στον επεξεργαστή και στη μνήμη, γεγονός που συχνά οδηγεί σε μη βέλτιστες αποφάσεις δρομολόγησης και φαινόμενα σύγκρουσης μεταξύ συνυπάρχοντων εφαρμογών. Οι πάροχοι υπηρεσιών Cloud αναγνωρίζουν αυτό το ζήτημα και συχνά θυσιάζουν αποδοτική αξιοποίηση των πόρων προκειμένου να διατηρήσουν την απαιτούμενη κλάση Ποιότητας Υπηρεσίας (QoS) που έχει ζητηθεί από τον πελάτη.

Στην παρούσα διπλωματική εργασία, εντοπίζουμε πειραματικά τις προαναφερθείσες προκλήσεις με στόχο τη σχεδίαση ενός πιο αποδοτικού μηχανισμού διαχείρισης πόρων που ενσωματώνεται στο Kubernetes. Αξιοποιούμε τα σημεία επέκτασης του Kubernetes για προγραμματιστές, καθώς και διάφορα εργαλεία παρακολούθησης συστήματος και benchmarking, ώστε να δημιουργήσουμε διαφορετικές πολιτικές δρομολόγησης που βασίζονται σε προφίλ εφαρμογών και σε μετρικές συστήματος, σε αντίθεση με την βασική λειτουργία του δρομολογητή που λαμβάνει υπόψη μόνο τη CPU και τη μνήμη. Καταγράφουμε τα χαρακτηριστικά των εισερχόμενων εφαρμογών βάσει χαμηλού επιπέδου μετρικών συστήματος, όπως ταχύτητα μεταφοράς δεδομένων από και προς την μνήμη (Memory Bandwidth), εντολές ανά κύκλο (Instructions per Cycle - IPC), αστοχίες cache επιπέδου L2 και L3 (L2 & L3 Cache Misses), και εφαρμόζουμε αποφάσεις δρομολόγησης μέσω του εξατομικευμένου μας δρομολογητή (scheduler). Στη συνέχεια, αξιολογούμε την αποτελεσματικότητα της λύσης μας συγκρίνοντας την επιβράδυνση των εφαρμογών πριν και μετά την εφαρμογή του scheduler που υλοποιήσαμε. Διεξάγουμε πειράματα χρησιμοποιώντας διάφορα benchmarks που προσομοιώνουν ρεαλιστικά σενάρια καταπόνησης του συστήματος και αποδεικνύουμε τον αντίκτυπο της λύσης μας στην πρόβλεψη παρεμβολών και στη βελτίωση της συνολικής επίδοσης του συστήματος.

Λέξεις-Κλειδιά — Υπολογιστικό Νέφος, Κιβωτιοποίηση, Kubernetes, Διαχείριση Πόρων, Βελτιστοποίηση Επίδοσης, Μετριασμός Παρεμβολών, Δρομολόγηση Εφαρμογών, Δοκιμές Επίδοσης.

Abstract

Nowadays, the diversity of workloads that are hosted on the Cloud is extensive, ranging from high-performance applications to microservice software architecture, data analytics and machine learning pipelines. As the Cloud paradigm is constantly expanding in every modern field of computation, the need to improve the performance and resource utilization of datacenter infrastructure becomes crucial, not only for the end-user, but also from a power and cost efficiency perspective. Containerization of applications is one such optimization strategy that offers many advantages over the preceding hypervisor-based virtualization, such as portability, reproducibility, lower performance overhead and memory requirements, faster deployment and scaling. Kubernetes is an open-source container orchestrator for deploying, managing and scaling containerized applications in production environments. While the performance upgrade over traditional clusters is apparent, orchestrators generally do not rely on fine-grained resource information for scheduling and executing applications. Additionally, they often lack awareness of the application's internal characteristics. Kubernetes is only aware of simplistic metrics such as CPU and memory load, often leading to sub-optimal scheduling decisions and interference phenomena between co-located workloads. Cloud Service Providers are aware of this issue and are willing to compromise resource utilization to uphold the Quality of Service class requested by the customer.

In this thesis, we experimentally identify the formerly described challenges in an attempt to design a more efficient resource management mechanism that integrates with Kubernetes. We leverage Kubernetes' extension points for developers, as well as different system monitoring and benchmarking tools and create sophisticated scheduling policies that utilize application profiling in contrast to the baseline CPU and memory affinity enabled policy of the default scheduler. We profile incoming applications based while observing low-level system metrics, e.g. Memory Bandwidth, Instructions per Cycle, L2 and L3 Cache Misses, and apply scheduling decisions with our custom scheduler. Afterwards, we evaluate the effectiveness of our solution by comparing the slowdown of the applications prior and after deploying our custom solution. We conduct experiments using numerous benchmarks that introduce realistic scenarios of stress on the system and demonstrate the impact of our solution in foreseeing resource contention and improving overall system performance.

Keywords — Cloud Computing, Containerization, Kubernetes, Resource Management, Performance Optimization, Interference Mitigation, Application Scheduling, Benchmarking

Ευχαριστίες

Θέλω να ευχαριστήσω από την καρδιά μου το προσωπικό του Εργαστηρίου Υπολογιστικών Συστημάτων (CSLab), και κυρίως τον διδακτορικό φοιτητή Ιωάννη Παπαδάκη και τον καθηγητή Γεώργιο Γκούμα, που μου έδωσαν την ευκαιρία να ασχοληθώ με αυτό το ιδιαίτερα ενδιαφέρον και απαιτητικό έργο. Χωρίς την γνώση, την καθοδήγηση και την υποστήριξή τους, δεν θα είχα την δυνατότητα να εμβαθύνω τόσο στο αντικείμενο, ούτε να βρω λύσεις για ζητήματα που αρχικά έμοιαζαν αδιανόητα.

Στα έξι χρόνια φοίτησής μου στο Πολυτεχνείο, έχω συνάψει αληθινές φιλίες με ανθρώπους που αγαπώ και θαυμάζω ιδιαίτερα. Ζήσαμε αξέχαστες στιγμές, συνεργαστήκαμε, περάσαμε μαζί γιορτές και δυσκολίες, δημιουργήσαμε αναμνήσεις που θα φέρω πάντα μαζί μου. Σας ευχαριστώ πολύ για την αμέριστη συναισθηματική υποστήριξη, και την έμπνευση που μου δώσατε να εξελιχθώ όχι μόνο γνωσιακά, αλλά και ως άνθρωπος.

Τέλος, θα ήθελα να ευχαριστήσω τους γονείς μου Βάλια και Δημήτρη, και την αδερφή μου, Κατερίνα. Παρά τις εντάσεις και τις διαφωνίες μας, η οικογένειά μου δεν σταμάτησε ποτέ να με στηρίζει, να με συμβουλεύει, να με ενθαρρύνει και να δείχνει πίστη και υπομονή, αποτελώντας τον πιο σταθερό πυλώνα της ζωής μου. Τους οφείλω ό,τι έχω καταφέρει μέχρι στιγμής, και καμία λέξη δεν περιγράφει την ευγνωμοσύνη μου για εκείνους.

Στεφανάκης Γεώργιος, Φεβρουάριος 2025

Contents

List of Figures

Chapter 1

Εκτεταμένη Περίληψη στα Ελληνικά

1.1 Εισαγωγικές Έννοιες

Η υπολογιστική νέφους είναι μία αναδυόμενη τεχνολογία που επεκτείνεται συνεχώς σε κάθε μοντέρνο τομέα της πληροφορικής. Το νέφος (cloud) ορίζεται, σύμφωνα με το Εθνικό Ινστιτούτο Προτύπων και Τεχνολογίας (NIST) των ΗΠΑ, ως ένα μοντέλο που επιτρέπει την πρόσβαση σε ένα σύνολο διαμοιραζόμενων και διαμορφώσιμων υπολογιστικών πόρων, διαθέσιμων μέσω του διαδικτύου [15]. Το cloud παρέχει αυτούς τους πόρους με μικρό διαχειριστικό κόστος από την πλευρά του τελικού χρήστη και ελάχιστη διάδραση με τον πάροχο υπηρεσιών νέφους (Cloud Service Provider - CSP). Το νέφος αποτελείται από από ένα δίκτυο κέντρων δεδομένων (datacenters), κάθε ένα από τα οποία στεγάζει χιλιάδες διασυνδεδεμένους υπολογιστές, οι οποίοι μπορούν να χρησιμοποιηθούν για την εκπόνηση διαφόρων εργασιών, όπως την διανομή εφαρμογών, πλατφορμών, και υπηρεσιών στο διαδίκτυο. Σε αντίθεση με το μέχρι τώρα καθιερωμένο μοντέλο της επί τόπου στέγασης υπολογιστικών συστημάτων (on-premise computing), ένας οργανισμός μπορεί να εκμεταλλευτεί του ευέλικτους, και ανοιχτούς σε κλιμάκωση διαθέσιμους πόρους του νέφους για τις υπολογιστικές του ανάγκες, ελαχιστοποιώντας το κόστος συντήρησης και αναβάθμισης φυσικών εξυπηρετητών (servers) στις υποδομές του. Επιπλέον, το νέφος διαθέτει ευέλικτα "pay-as-you-go" σχήματα τιμολόγησης, γεγονός που μειώνει τις προκαταβολικές επενδύσεις των οργανισμών και τους δίνει την δυνατότητα να επεκτείνουν τους πόρους τους κατά το δοκούν. Οι τεχνολογίες εικονικοποίησης υπολογιστικών συστημάτων που παρέχει το cloud είναι ο λόγος που πλέον χρησιμοποιείται κατά κόρον σε μεγάλης κλίμακας εφαρμογές, και για αυτόν τον λόγο η σωστή διαχείριση των διαθέσιμων υπολογιστικών πόρων αποτελεί μία επιτακτική ανάγκη.

Η υπολογιστική νέφους διακρίνεται σε τρία κύρια μοντέλα υπηρεσιών: Λογισμικό ως Υπηρεσία (SaaS), όπου οι χρήστες έχουν πρόσβαση σε εφαρμογές χωρίς εγκατάσταση ή διαχείριση, Πλατφόρμα ως Υπηρεσία (PaaS), που επιτρέπει στους προγραμματιστές να δημιουργούν και να αναπτύσσουν εφαρμογές σε υποδομή νέφους, και Υποδομή ως Υπηρεσία (IaaS), η οποία προσφέρει εικονικοποιημένους υπολογιστικούς πόρους, όπως εικονικές μηχανές (VMs) και αποθηκευτικό χώρο. Η εικονικοποίηση (virtualization) αποτελεί βασικό μηχανισμό της υπολογιστικής νέφους, καθώς επιτρέπει την εκτέλεση πολλαπλών εικονικών μηχανών πάνω σε έναν φυσικό εξυπηρετητή. Το hypervisor-based virtualization παρέχει απομόνωση μεταξύ των εφαρμογών, αλλά συνεπάγεται υψηλό κόστος σε αποθηκευτικό χώρο και επίδοση. Αντίθετα, η τεχνολογία των κοντέινερ (containers) προσφέρει μια ελαφρύτερη λύση, επιτρέποντας την εκτέλεση πολλαπλών εφαρμογών στον ίδιο πυρήνα λειτουργικού συστήματος, χωρίς την ανάγκη για ξεχωριστό εικονικό μηχάνημα. Οι containers είναι πιο αποδοτικοί σε επίδοση και κόστος, διευκολύνοντας τη φορητότητα των εφαρμογών και την αυτοματοποίηση της διανομής τους.

Η ενορχήστρωση πακέτων (container orchestration) αναφέρεται στην αυτοματοποίηση της ανάπτυξης, της κλιμάκωσης (scaling) και της διαχείρισης εφαρμογών που εκτελούνται σε κοντέινερ (containerized applications) μέσα σε ένα σύμπλεγμα εξυπηρετητών (cluster). Ο ενορχηστρωτής αλληλεπιδρά με το περιβάλλον εκτέλεσης κοντέινερ (container runtime) και διαχειρίζεται εργασίες όπως την εκκίνηση, την ενημέρωση, τη μετακίνηση, την κλιμάκωση και την αφαίρεση κοντέινερ.

Ο Κυβερνήτης (Kubernetes) είναι ένας ανοικτού κώδικα ενορχηστρωτής κοντέινερ (container orchestrator) που έχει γίνει το πρότυπο για εφαρμογές μεγάλης κλίμακας [9]. Εκτός από τις βασικές λειτουργίες εκτέλεσης

(deployment), κλιμάκωσης (scaling) και διαχείρισης κοντέινερ, αυτοματοποιεί κρίσιμες λειτουργίες σε επίπεδο υποδομής, όπως την ανακάλυψη υπηρεσιών (service discovery), την εξισορρόπηση φορτίου (load balancing), τη διαχείριση αποθηκευτικού χώρου (storage management), την αναίρεση ενημερώσεων (rollbacks) και την αυτοθεραπεία (self-healing) των εφαρμογών. Αν και το Kubernetes είναι βελτιστοποιημένο για την εκτέλεση εφαρμογών με αρχιτεκτονική μικροϋπηρεσιών (microservices), η ευελιξία του επιτρέπει την εκτέλεση διαφόρων εφαρμογών, όπως αναλύσεις δεδομένων (data analytics), ροές μηχανικής μάθησης (machine learning pipelines) και επιστημονικές εφαρμογές υψηλής επίδοσης (scientific high-performance workloads).

Το Kubernetes παρέχει επίσης επεκτάσιμες, φιλικές προς τους προγραμματιστές (developer-centric) διεπαφές (frameworks), που επιτρέπουν την ενσωμάτωση εφαρμογών για προσαρμοσμένες απαιτήσεις, υπό την μορφή επεκτάσεων (plugins) της ήδη υπάρχουσας λειτουργικότητας ενός Kubernetes cluster. Αυτές οι επεκτάσεις περιλαμβάνουν τη δημιουργία προσαρμοσμένης λογικής δρομολόγησης εφαρμογών (custom scheduling logic), την εισαγωγή προσαρμοσμένων δεδομένων (custom resources) στον API server και την παραγωγή των αντίστοιχων clients, καθώς και τη χρήση εξωτερικών συσκευών (device plugins) που επιτρέπουν στους διαχειριστές να αξιοποιούν εξειδικευμένο υλικό, όπως μονάδες επεξεργασίας γραφικών (GPUs), μέσα στα κοντέινερ τους. Παρόλο που το Kubernetes είναι ένα πολύπλοκο σύστημα, απαιτείται περαιτέρω έρευνα για τη βελτιστοποίηση της κατανομής πόρων CPU και μνήμης στις εφαρμογές. Προς το παρόν, βασίζεται σε τυπικά ποσοστά χρήσης μνήμης και CPU των μηχανημάτων (worker nodes), τα οποία δεν είναι πάντα ακριβή. Αυτό μπορεί να οδηγήσει σε ανταγωνισμό για πόρους (resource contention) και περιορισμό επίδοσης (throttling), προκαλώντας υποβέλτιστη επίδοση εκτέλεσης (sub-optimal performance). Ο προκαθορισμένος scheduler δεν λαμβάνει πάντα βέλτιστες αποφάσεις, καθώς δεν βασίζεται στην κατάσταση του cluster σε πραγματικό χρόνο ή στα χαρακτηριστικά των εφαρμογών. Επιπλέον, το Kubelet (daemon που τρέχει σε κάθε node) δεν διαθέτει έναν εξελιγμένο μηχανισμό για την αποκλειστική παραχώρηση πόρων στις εφαρμογές, οδηγώντας σε ανταγωνισμό για κοινόχρηστους πόρους και τελικά σε υποβάθμιση της επίδοσης.

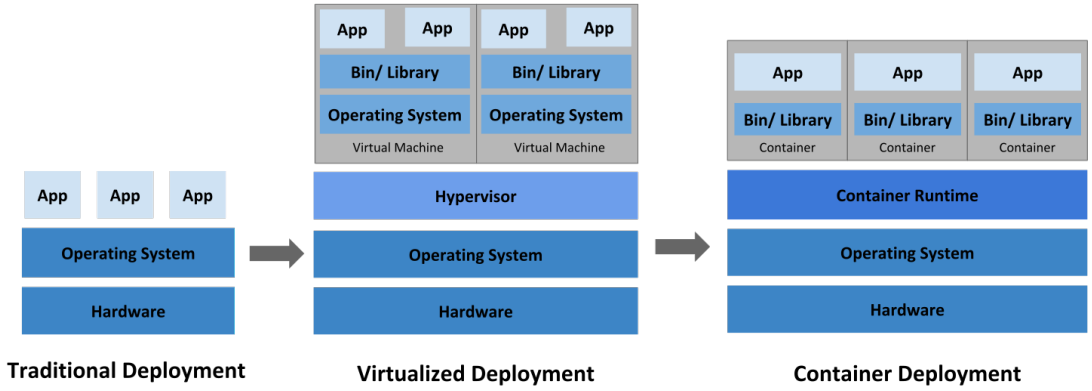


Figure 1.1.1: Εξέλιξη των Τεχνολογιών Εικονικοποίησης

Οι πάροχοι υπηρεσιών νέφους βασίζονται σε συνεχή αναβάθμιση του υλικού (hardware) των πληροφοριακών υποδομών τους, σε συνδυασμό με απλοϊκές πολιτικές δρομολόγησης εφαρμογών και παραχώρησης πόρων για να ελαχιστοποιήσουν το ρίσκο υπολειτουργίας των κρισιμών (latency critical) εφαρμογών. Δεδομένης της ραγδαίας μετατόπισης προς το cloud, οι πάροχοι οφείλουν να λάβουν υπόψην τους πιο αποδοτικούς τρόπους εφοδιασμού πόρων για να μειώσουν περιττά κόστη και να συντηρήσουν την επίδοση των εφαρμογών στο απαιτούμενο εύρος. Ιδιαίτερα σε συστήματα πολλαπλών επεξεργαστικών μονάδων (multi-core processor systems), όπου κάθε επεξεργαστική μονάδα απαρτίζεται από ένα κόμβο μνήμης (NUMA node), πολλαπλούς φυσικούς πυρήνες και επίπεδα κρυφής μνήμης (cache), η συνύπαρξη εφαρμογών διαφόρων ενοικιαστών (multi-tenancy) υπαγορεύει διάφορες προκλήσεις. Μερικές από αυτές αφορούν την ασφάλεια των συνυπάρχουσων εφαρμογών και την εξάλειψη της πιθανότητας διαρροής δεδομένων, ενώ πολύ σημαντική είναι και η βελτιστοποίηση των πόρων που δίνονται στις εφαρμογές, τόσο κατά την εκκίνησή τους, όσο και δυναμικά κατά την εκτέλεσή τους στην περίπτωση εμφάνισης υποβιβασμού της ποιότητας εκτέλεσης. Οι πάροχοι οφείλουν να τηρούν τις συμφωνίες επιπέδου υπηρεσίας (Service-Level Agreement - SLA), οι οποίες περιλαμβάνουν μετρικές επίδοσης που έχουν προσυμφωνηθεί, όπως ρυθμός επεξεργασίας αιτημάτων (throughput) και καθυστέρηση (latency). Η παραβίαση των συμφωνηθέντων

μετρικών οδηγεί σε ποινές (SLA penalties) και υποβιβάζει την ποιότητα εμπειρίας του τελικού χρήστη. Η παρεμβολή από «θορυβώδεις γείτονες» (noisy neighbors) και η διεκδίκηση πόρων (resource contention) μπορεί να οδηγήσουν σε υποβάθμιση της επίδοσης, καθιστώντας δύσκολη τη διατήρηση των υποσχόμενων επιπέδων Ποιότητας Υπηρεσίας (Quality of Service - QoS). Συνεχιζόμενη έρευνα και πρόοδοι στη διαχείριση πόρων, τον προγραμματισμό και τις τεχνικές βελτιστοποίησης είναι απαραίτητες για την αντιμετώπιση αυτών των προκλήσεων και τη βελτίωση της συνολικής QoS στα περιβάλλοντα νέφους. Τέλος, η σωστή διαχείριση των πόρων έχει τεράστια σημασία για μείωση του περιβαλλοντικού αποτυπώματος και την ελαχιστοποίηση του κόστους λειτουργίας. Η κατανάλωση ενέργειας των κέντρων δεδομένων, αυξάνεται ραγδαία, με προβλέψεις να φτάσει τα επίπεδα συνολικής ηλεκτρικής κατανάλωσης της Ιαπωνίας έως το 2026 [4]. Για τη μείωση του περιβαλλοντικού αποτυπώματος και του κόστους λειτουργίας, απαιτούνται τεχνικές βελτιστοποίησης, όπως η δυναμική κατανομή πόρων και η ενσωμάτωση της τεχνητής νοημοσύνης για αποδοτικότερη διαχείριση εφαρμογών με απρόβλεπτη χαρακτηριστικά εκτέλεσης.

Στην παρούσα διπλωματική εργασία, θα αναλύσουμε τα βασικά χαρακτηριστικά του ενορχηστρωτή Kubernetes και τα frameworks που διαθέτει για να επεκτείνουμε τη λειτουργικότητά του με σκοπό την κατασκευή ενός μηχανισμού δρομολόγησης και παραχώρησης πόρων για διαφορετικού τύπου εφαρμογές. Θα χαρακτηρίσουμε διάφορα φορτία δοκιμής επίδοσης (benchmarks) με βάση το αποτύπωμα που παρουσιάζουν στην αξιοποίηση πόρων και θα ελέγξουμε την αποδοτικότητα του μηχανισμού μας στην δρομολόγηση και παραχώρηση συγκεκριμένων αποκλειστικών πόρων σε επίπεδο CPU και μνήμης. Θα συγκρίνουμε τον χρόνο εκτέλεσης των εφαρμογών αυτών πριν και μετά την πρόσληψη του προσαρμοσμένου μας μηχανισμού και θα δείξουμε ότι είναι εφικτό να επιτύχουμε καλύτερους χρόνους εκτέλεσης γνωρίζοντας την φύση των εφαρμογών που επιθυμούμε να δρομολογήσουμε.

1.2 Σχετικές Μελέτες

Στο παρόν κεφάλαιο αναλύθηκαν διάφορες μελέτες και έργα που έχουν διεξαχθεί από ακαδημαϊκούς οργανισμούς αλλά και από επιχειρήσεις, σχετικά με την βελτιστοποίηση διαχείρισης πόρων στο cloud.

Η επέκταση Volcano [22] είναι ένα cloud-native σύστημα διαχείρισης για batch workloads, όπως ροές μηχανική μάθηση, εφαρμογές HPC, και big data. Το Volcano ενσωματώνει γενικευμένα frameworks για batch εφαρμογές, π.χ. Tensorflow, Spark, PyTorch, και MPI, ενώ δίνει τη δυνατότητα στους χρήστες να δρομολογήσουν αυτά τα φορτία υπό διάφορες πολιτικές, όπως co-scheduling, fair-share, gang, και topology-aware scheduling. Ένα άλλο σύστημα δρομολόγησης εφαρμογών που επεκτείνει τον Κυβερνήτη, είναι το Koordinator [8], επιτρέποντας την συνύπαρξη πολυποίκιλων εφαρμογών, συμπεριλαμβανομένων microservices, εφαρμογών AI, και big data. Το Koordinator παρέχει έναν μηχανισμό δρομολόγησης που βασίζεται σε ειδικές QoS κλάσεις που υποστηρίζει. Χρησιμοποιώντας το προσαρμοσμένο πεδίο `koordinator.sh/qosClass` για να ορίσουμε την ποιότητα υπηρεσίας μίας συγκεκριμένης εφαρμογής, σε συνδυασμό με το προφίλ του cluster που ορίζεται στο ClusterColocationProfile, μπορεί να πάρει πιο λεπτομερείς αποφάσεις δρομολόγησης. Επιπλέον, έχει την δυνατότητα να προκαλέσει υπερφόρτωση πόρων (oversubscription) και να ανακτήσει πόρους μη χρησιμοποιούμενους από Pods υψηλής προτεραιότητας, παρέχοντάς τους σε εφαρμογές χαμηλότερης προτεραιότητας. Το Koordinator επιπλέον υποστηρίζει load-aware scheduling, το οποίο εξισορροπεί την κατανομή των εφαρμογών κατά μήκος όλου του cluster, αποτρέποντας υπερφόρτωση και υποχρησιμοποίηση πόρων χρησιμοποιώντας real-time δεδομένα, συλλεγόμενα από τους workers. Το σύστημα προσφέρει λεπτομερή ενορχήστρωση πόρων και απομόνωση για να εξασφαλίσει την επίδοση των εφαρμογών.

Ο ACTiManager [16] είναι ένας διαχειριστής πόρων που χρησιμοποιεί τεχνικές Μηχανικής Μάθησης για να διαχειρίζεται την κατανομή VMs, βελτιώνοντας την ισορροπία μεταξύ κόστους και επίδοσης, επιτυγχάνοντας αύξηση κέρδους έως και 49%. Οι Liu et al. [11] πρότειναν ένα σύστημα δρομολόγησης για εφαρμογές Υψηλής Επίδοσης (HPC) στο Kubernetes, το οποίο μειώνει τον χρόνο απόκρισης κατά 35% και βελτιώνει την επίδοση κατά 34% μέσω μίας αρχιτεκτονικής δύο επιπέδων, με εξειδικευμένους αλγόριθμους κατανομής εργασιών. Το Escra από τους Cusack et al. είναι ένα σύστημα που προσαρμόζει τους πόρους σε χρονικά διαστήματα των 100ms, μειώνοντας τη λανθάνουσα κατάσταση κατά 38% και αυξάνοντας την επίδοση κατά 25,4% [2]. Ο μηχανισμός των Rodriguez et al. συνδυάζει προγραμματισμό, αναπρογραμματισμό και αυτόματη κλιμάκωση, μειώνοντας το κόστος έως 58% σε σχέση με τον προεπιλεγμένο Kubernetes scheduler. Αυτός ο μηχανισμός δεν εστιάζει μόνο στη δρομολόγηση εργασιών, αλλά ενσωματώνει επίσης δυναμική κλιμάκωση (autoscaling) και επανατοποθέτηση κοντέινερ (relocation), επιτρέποντας το να προσαρμόζει τον αριθμό των worker VMs σε

πραγματικό χρόνο, κλιμακώνοντας δυναμικά για να ανταποκριθεί στη ζήτηση και ανακατανέμοντας κοντέινερ ώστε να μειώσει το κόστος και να αξιοποιήσει αποδοτικότερα τους διαθέσιμους πόρους. Οι αλγόριθμοι BDI και BCDI από τους Li et al. βελτιώνουν τη διαχείριση I/O στο Kubernetes, μειώνοντας τον χρόνο απόκρισης των εφαρμογών και εξισορροπώντας τη χρήση CPU και δίσκου. Το JIAGU από τους Liu et al. βελτιώνει την επίδοση serverless περιβαλλόντων, μειώνοντας το κόστος της δρομολόγησης κατά 81-93% και τον χρόνο εκκίνησης κατά 57,4-69,3% [13]. Το ChainsFormer από τους Song et al. χρησιμοποιεί μηχανική μάθηση για να προσαρμόζει τους πόρους δυναμικά, μειώνοντας τον χρόνο απόκρισης κατά 26% και βελτιώνοντας τη συνολική επίδοση κατά 8% [18]. Τέλος, το μοντέλο υβριδικής, δρομολόγησης, με διαμοιραζόμενη κατάσταση, από τους Ungureanu et al., συνδυάζει κεντρικό και κατανεμημένο προγραμματισμό, βελτιώνοντας την ευελιξία και την επίδοση κατανομής εργασιών [21]. Αυτές οι τεχνικές καταδεικνύουν την ανάγκη για schedulers που υπερβαίνουν τις βασικές δυνατότητες του Kubernetes, επιτρέποντας δυναμική προσαρμογή και βελτιστοποιημένη χρήση των υπολογιστικών πόρων.

1.3 Ο Ενορχηστρωτής Kubernetes

Ένας ενορχηστρωτής κοντέινερ είναι υπεύθυνος για την παραχώρηση πόρων, το deployment, την κλιμάκωση και την γενική διαχείριση containerized εφαρμογών σε ένα σύμπλεγμα από εξυπηρετητές. Ο Κυβερνήτης (Kubernetes) είναι η πρότυπη, open-source πλατφόρμα που εξυπηρετεί αυτόν τον σκοπό. Παρέχει ένα δηλωτικό API μέσω του οποίου ο διαχειριστής έχει την δυνατότητα να ορίσει την επιθυμητή κατάσταση του cluster, μέσα από αρχεία διαμόρφωσης (configuration files - manifests) και τελικά να εκτελέσει εφαρμογές οι οποίες κυμαίνονται από deployments εφαρμογών μεγάλης κλίμακας, φορτία υψηλής επίδοσης, και ροές μηχανικής μάθησης και big data. Σε αυτό το κεφάλαιο αναλύονται οι βασικές συνιστώσες ενός Kubernetes cluster και οι προεπιλεγμένες πολιτικές διαχείρισης πόρων που παρέχει. Επιπλέον, αναλύονται κάποια από τα frameworks που μπορούν οι προγραμματιστές να εκμεταλλευτούν προκειμένου να κατασκευάσουν επεκτάσεις στην βασική λειτουργικότητα του Kubernetes.

Ένα Kubernetes cluster αποτελείται από το control plane και ένα σύνολο worker nodes, όπου εκτελούνται containerized εφαρμογές. Το control plane διαχειρίζεται την κατάσταση του cluster και επιβλέπει τα Pods, τις βασικές μονάδες εργασίας των εφαρμογών. Τα κύρια στοιχεία του control plane περιλαμβάνουν τον API Server, ο οποίος είναι η κύρια πηγή αλήθειας του cluster και παρέχει ένα RESTful API για τη διαχείριση των resources, το etcd, που αποθηκεύει την κατάσταση και τις ρυθμίσεις του cluster, τον kube-scheduler, ο οποίος αναθέτει Pods σε nodes βάσει διαθέσιμων πόρων και περιορισμών, και τον kube-controller-manager, που λειτουργεί ως ένας συνεχής βρόχος ελέγχου για τη διατήρηση της επιθυμητής κατάστασης των πόρων. Από την άλλη, τα nodes περιέχουν το kubelet, ένα daemon που διαχειρίζεται την εκτέλεση των Pods σε κάθε node, και τον kube-proxy, που επιτρέπει την επικοινωνία μεταξύ των υπηρεσιών του cluster. Αυτή η αρχιτεκτονική επιτρέπει στο Kubernetes να διαχειρίζεται δυναμικά την εκτέλεση εφαρμογών, εξασφαλίζοντας υψηλή διαθεσιμότητα και αποδοτική αξιοποίηση των πόρων.

Στο Kubernetes, τα βασικά resources ενός cluster περιλαμβάνουν τα Pods, τα οποία είναι οι μικρότερες μονάδες εκτελέσιμου workload και αποτελούνται από έναν ή περισσότερους containers που μοιράζονται αποθηκευτικούς και δικτυακούς πόρους. Για την ομαλή διαχείριση των Pods, χρησιμοποιούνται ανώτερα επίπεδα ελέγχου όπως τα Deployments, που εξασφαλίζουν αναπαραγωγή, scaling και αυτοίαση των εφαρμογών, τα StatefulSets για διαχείριση καταστάσεων με διατήρηση ταυτότητας και αποθήκευσης, καθώς και τα DaemonSets που διασφαλίζουν ότι ένα συγκεκριμένο Pod εκτελείται σε κάθε node. Τα Services παίζουν κεντρικό ρόλο στη δικτύωση του cluster, επιτρέποντας την επικοινωνία μεταξύ των Pods και την πρόσβαση σε εφαρμογές, ακόμα και αν τα Pods αλλάζουν διευθύνσεις. Οι βασικοί τύποι περιλαμβάνουν το ClusterIP για εσωτερική επικοινωνία, το NodePort για έκθεση σε εξωτερικά δίκτυα, και το LoadBalancer για διαχείριση εξωτερικής κίνησης μέσω cloud παρόχων. Όσον αφορά την αποθήκευση, το Kubernetes παρέχει Volumes για προσωρινή αποθήκευση δεδομένων μεταξύ container επανεκκινήσεων, ενώ για μόνιμη αποθήκευση χρησιμοποιεί Persistent Volumes (PVs) και Persistent Volume Claims (PVCs). Αυτά επιτρέπουν τη διατήρηση δεδομένων ανεξάρτητα από τη διάρκεια ζωής των Pods και μπορούν να υλοποιηθούν μέσω StorageClasses που διαχειρίζονται αυτόματα τους πόρους αποθήκευσης, διασφαλίζοντας αποδοτικότητα και ανεξαρτησία από το υποκείμενο σύστημα αρχείων.

Ο Kubernetes επιτρέπει στους χρήστες να καθορίσουν απαιτήσεις πόρων (CPU, μνήμη κ.λπ.) για κάθε container μέσα σε ένα Pod, διασφαλίζοντας ότι οι εφαρμογές λαμβάνουν τους απαραίτητους πόρους για τη λειτουργία τους. Οι πόροι χωρίζονται σε "requests", που αντιπροσωπεύουν τον ελάχιστο εγγυημένο πόρο που θα δεσμευθεί,

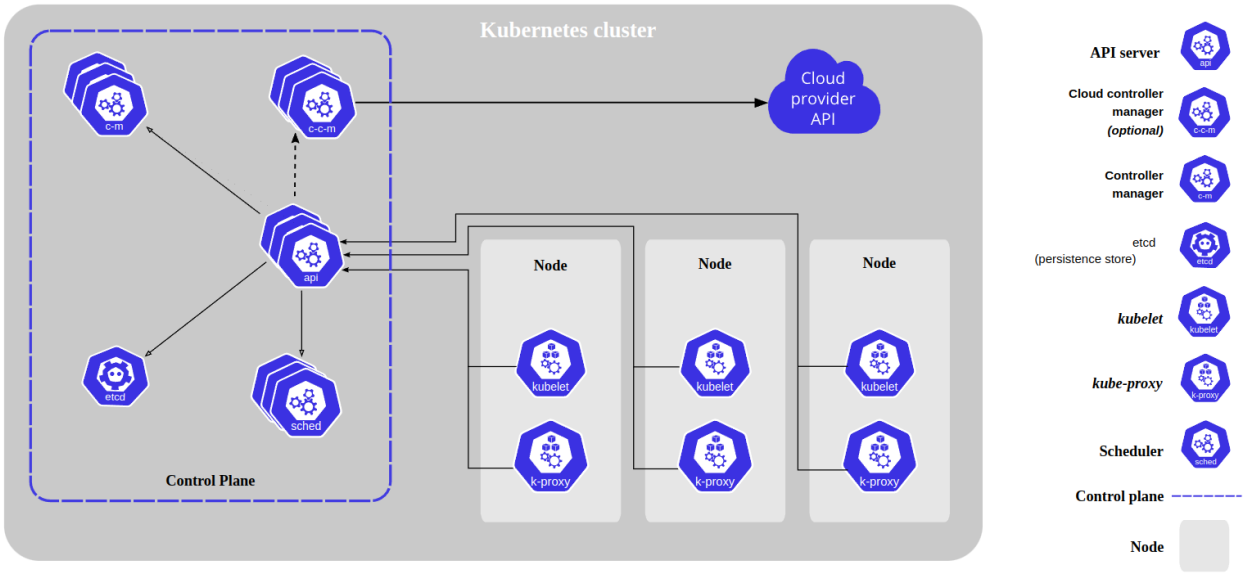


Figure 1.3.1: Βασικές Συνιστώσες Ενός Kubernetes Cluster

και "limits", που θέτουν το ανώτατο όριο κατανάλωσης. Ο kube-scheduler χρησιμοποιεί τα requests για να τοποθετήσει Pods σε κατάλληλους κόμβους, ενώ το kubelet επιβάλλει τα limits, αποτρέποντας τα containers από το να υπερβούν τους καθορισμένους περιορισμούς. Αυτή η διαχείριση διασφαλίζει την αποτελεσματική αξιοποίηση των πόρων και αποτρέπει την εξάντληση των διαθέσιμων υποδομών.

Ο Kubernetes ταξινομεί τα Pods σε τρεις κατηγορίες Quality of Service (QoS) βάσει των καθορισμένων requests και limits των πόρων τους. Τα Guaranteed Pods έχουν τόσο requests όσο και limits ίσα μεταξύ τους για όλους τους containers, λαμβάνοντας την υψηλότερη προτεραιότητα και τη χαμηλότερη πιθανότητα έξωσης (eviction). Τα Burstable Pods έχουν τουλάχιστον ένα request καθορισμένο, επιτρέποντάς τους να εκμεταλλεύονται διαθέσιμους πόρους δυναμικά. Τα BestEffort Pods δεν έχουν καθορισμένα requests ή limits, με αποτέλεσμα να έχουν τη χαμηλότερη προτεραιότητα και να είναι τα πρώτα που εκδιώκονται σε περίπτωση έλλειψης πόρων.

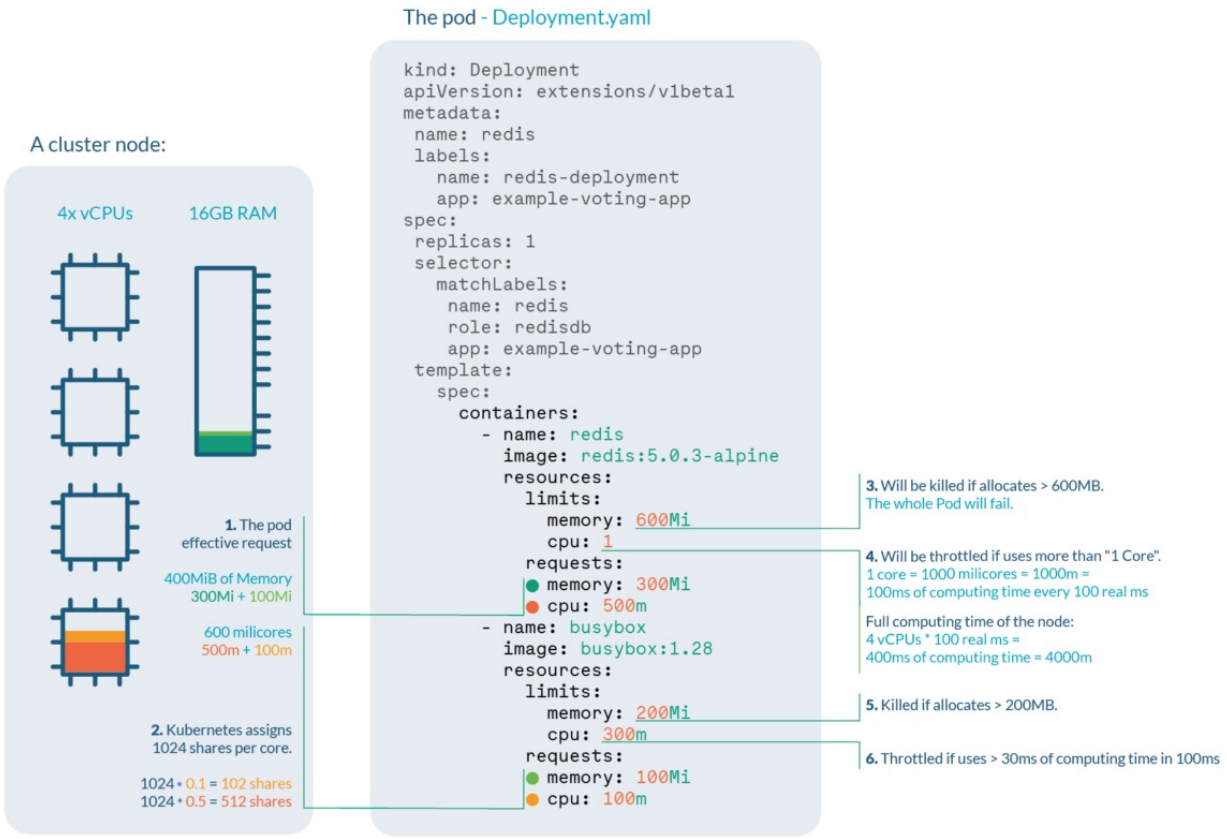


Figure 1.3.2: Resource Requests και Limits ενός Pod

Το kubelet είναι το βασικό agent που τρέχει σε κάθε κόμβο Kubernetes και είναι υπεύθυνο για την εκτέλεση των Pods, καθώς και για την επιβολή των περιορισμών πόρων. Χρησιμοποιεί τα cgroups του Linux για να εφαρμόσει ελέγχους σε CPU, μνήμη και διεργασίες, εξασφαλίζοντας ότι κάθε container λειτουργεί μέσα στα όρια που έχουν οριστεί. Επιπλέον, ο CPU Manager μπορεί να εκχωρεί αποκλειστικούς πυρήνες CPU σε Pods με υψηλές απαιτήσεις επίδοσης, ενώ ο Topology Manager ευθυγραμμίζει την κατανομή των πόρων με την αρχιτεκτονική NUMA, μειώνοντας την καθυστέρηση πρόσβασης στη μνήμη. Αυτοί οι μηχανισμοί διασφαλίζουν αποδοτική χρήση των πόρων και σταθερή επίδοση των εφαρμογών. Ωστόσο, παρουσιάζει ορισμένους περιορισμούς, καθώς δεν επιτρέπει αλλαγές στη διαμόρφωση του χωρίς επανεκκίνηση του kubelet, γεγονός που μπορεί να διαταράξει τα τρέχοντα workloads. Επίσης, δεν υποστηρίζει ταυτόχρονα πολλαπλές πολιτικές κατανομής CPU, κάτι που τον καθιστά λιγότερο ευέλικτο για ετερογενή workloads που απαιτούν διαφορετικές στρατηγικές, όπως κατανομή με γνώμονα τη NUMA αρχιτεκτονική ή απομόνωση πυρήνων. Επιπλέον, δεν προσαρμόζει δυναμικά την κατανομή πόρων μετά την εκτέλεση των containers, περιορίζοντας την ικανότητα αντιμετώπισης επιβραδύνσεων κατά την εκτέλεση. Λόγω αυτών των περιορισμών, υπάρχει ανάγκη για πιο ευέλικτους αλγορίθμους διαχείρισης πόρων που μπορούν να προσαρμοστούν στις μεταβαλλόμενες απαιτήσεις των εφαρμογών.

1.4 Προκλήσεις Διαχείρισης Πόρων σε Συστήματα Υψηλής Επίδοσης

Τα υπολογιστικά συστήματα υψηλής επίδοσης (high-performance computing systems) είναι σχεδιασμένα για την αποδοτική εκτέλεση απαιτητικών φορτίων. Όμως, υπάρχουν διάφοροι παράγοντες που μπορούν να οδηγήσουν στην χειροτέρευση της απόδοσής τους, οι οποίοι μπορούν να εντοπιστούν, είτε στις δικτυακές διασυνδέσεις των κόμβων του cluster, είτε εσωτερικά σε κάθε κόμβο, στους multi-core επεξεργαστές τους. Οι schedulers των HPC clusters δρομολογούν τις εφαρμογές με βάση μη-λεπτομερών μετρικών όπως πυρήνες, μνήμη, χώρος

δίσκου και swap μνήμης, κ.τλ. Σε αυτό το κεφάλαιο μελετήθηκαν διάφοροι παράγοντες που σχετίζονται με την ευαισθησία των εφαρμογών σε συνύπαρξη με άλλες εφαρμογές σε έναν κόμβο.

Η «επίδραση του θορυβώδους γείτονα» είναι ένα αναπόφευκτο φαινόμενο στις υποδομές cloud λόγω της κοινής χρήσης πόρων και της multi-tenant φύσης τους. Όταν πολλές εφαρμογές ή εικονικές μηχανές φιλοξενούνται στον ίδιο server, η συνύπαρξή τους μπορεί να οδηγήσει σε υποβάθμιση της επίδοσης. Αυτό συμβαίνει λόγω της διεκδίκησης περιορισμένων υπολογιστικών πόρων, όπως η κρυφή μνήμη του επεξεργαστή. Για παράδειγμα, η ταυτόχρονη φόρτωση δεδομένων από διαφορετικές εφαρμογές μπορεί να προκαλέσει «μόλυνση» της κρυφής μνήμης, αυξάνοντας τις αστοχίες (cache misses) και επιβραδύνοντας τον χρόνο εκτέλεσης. Άλλοι παράγοντες που οδηγούν στη διακύμανση της επίδοσης των εφαρμογών αποτελούν ο αριθμός λειτουργιών ανάγνωσης/εγγραφής μνήμης, οι πράξεις κινητής υποδιαστολής (FLOPs) και η επικοινωνία μεταξύ νημάτων μέσω locks και barriers. Οι πάροχοι υπηρεσιών cloud πρέπει να διαχειρίζονται προσεκτικά την δρομολόγηση και την κατανομή πόρων για τη μείωση της παρεμβολής και την τήρηση των συμφωνιών επιπέδου υπηρεσιών.

Στο πλαίσιο της Cloud-Native τεχνολογίας, όπου χρησιμοποιούνται containers και μηχανισμοί ενορχήστρωσης, έχουν δημοσιευτεί πολλές έρευνες για τη βελτιστοποίηση των πόρων σε multi-tenant περιβάλλοντα. Οι μελέτες αυτές καλύπτουν διάφορους τομείς, όπως η βαθιά μάθηση (Deep Learning), το βιομηχανικό IoT, το Batch Processing και η εκτέλεση ετερογενών εφαρμογών στο ίδιο cluster [24] [7] [5].

Η υποβάθμιση της επίδοσης λόγω ανταγωνισμού για πόρους μπορεί να φτάσει έως και το 200% όταν εκτελούνται πολλαπλά νήματα ή διεργασίες σε πολυπύρηνους επεξεργαστές [1]. Αυτό οφείλεται στη διεκδίκηση κοινών πόρων, όπως η κρυφές μνήμες τελευταίου επιπέδου (last-level caches - LLC), οι ελεγκτές μνήμης (memory controllers), και το memory bandwidth. Παρόλο που η συνύπαρξη των φορτίων βελτιώνει την αξιοποίηση του υλικού, μπορεί επίσης να οδηγήσει σε σοβαρή επιβράδυνση, αναιρώντας τα οφέλη, παραβιάζοντας τις απαιτήσεις ποιότητας υπηρεσιών (QoS) και ενδεχομένως σπαταλώντας ενέργεια.

Οι Hutcheson et al., σε μία δημοσίευσή τους, περιγράφουν τις διαφορές μεταξύ δύο τύπους εφαρμογών, τις memory-bound και τις CPU-bound [6]. Οι εφαρμογές memory-bound περιορίζονται από την ταχύτητα μεταφοράς δεδομένων μεταξύ μνήμης και επεξεργαστή, με αποτέλεσμα ο επεξεργαστής να μένει ανεκμετάλλευτος, ενώ οι compute-bound εφαρμογές περιορίζονται από την υπολογιστική ισχύ του επεξεργαστή, δημιουργώντας συμφόρηση στους υπολογισμούς. Οι πρώτες περιλαμβάνουν απλές πράξεις σε μεγάλα σύνολα δεδομένων, όπως αντιγραφή ή πρόσθεση διανυσμάτων, ενώ οι δεύτερες αφορούν έντονες υπολογιστικές εργασίες, όπως η παραγωγή τυχαίων αριθμών. Η διάκριση αυτή είναι κρίσιμη για τη βελτιστοποίηση επίδοσης, και στην παρούσα εργασία προτείνεται ένας μηχανισμός δρομολόγησης που αξιοποιεί αυτήν την διάκριση για αποδοτικότερη εκτέλεση εργασιών σε cloud περιβάλλοντα.

1.5 Maestro - Ένας Διαχειριστής Πόρων για Kubernetes

Σε αυτό το κεφάλαιο αναλύουμε την υλοποίηση ενός ενορχηστρωτή και δρομολογητή εφαρμογών ως επέκταση για Kubernetes clusters, τον Maestro. Το framework αυτό επιτρέπει διαχειριστές cluster αλλά και μελλοντικούς ερευνητές να πειραματιστούν με διαφορετικές πολιτικές παραχώρησης πόρων σε πιο λεπτομερές επίπεδο από εκείνο που επιτρέπει ο η βασική λειτουργικότητα του Kubernetes. Εκμεταλλευόμαστε διάφορα frameworks για προγραμματιστές για να κατασκευάσουμε έναν δηλωτικό μηχανισμό για CPU pinning και απομόνωση εφαρμογών πάνω σε συγκεκριμένους πόρους επεξεργαστή και μνήμης, σε συνδυασμό με έναν προσαρμοσμένο scheduler ο οποίος μπορεί να πάρει αποφάσεις με βάση τα χαρακτηριστικά των εφαρμογών και των πόρων που ήδη έχουμε παραχωρήσει σε αυτές. Ο σχεδιασμός του framework φροντίσαμε να είναι επεκτάσιμος έτσι ώστε να επιτρέπει την υλοποίηση πιο περίπλοκων τεχνικών δρομολόγησης και παραχώρησης πόρων από μελλοντικούς ερευνητές.

Στην ενότητα αυτή παρουσιάζονται δύο προσαρμοσμένα Custom Resource Definitions που επιτρέπουν την εποπτεία των διαθέσιμων υπολογιστικών πόρων σε κάθε κόμβο του cluster, καθώς και την τρέχουσα παραχώρηση πόρων σε κάθε Pod. Αυτό εξασφαλίζει ότι οι διαχειριστές του cluster γνωρίζουν σε πραγματικό χρόνο ποια Pods εκτελούνται σε συγκεκριμένους CPU και κόμβους μνήμης, καθώς και το αν οι πόροι που χρησιμοποιούν είναι αποκλειστικοί. Επιπλέον, αν παρατηρηθεί υποβέλτιστη κατανομή ή περιορισμός πόρων, οι διαχειριστές μπορούν να προσαρμόσουν δυναμικά τα custom resources μέσω τροποποίησης των σχετικών manifests.

Το πρώτο custom resource, `NodeCPUTopology`, περιγράφει τη διαθεσιμότητα CPU και NUMA των nodes, περιλαμβάνοντας πληροφορίες για λογικούς και φυσικούς πυρήνες, sockets και αρχιτεκτονική NUMA. Οι πόροι

αυτοί δημιουργούνται αυτόματα κατά την εκκίνηση του controller manager, επιτρέποντας στον προσαρμοσμένο scheduler να λαμβάνει αποφάσεις κατανομής μέσω του custom resource `PodCPUBinding`.

Το δεύτερο custom resource, `PodCPUBinding`, περιγράφει τους πόρους CPU και μνήμης που έχουν παραχωρηθεί σε συγκεκριμένα Pods στο cluster. Ο διαχειριστής του cluster ή εφαρμογές που λαμβάνουν αποφάσεις κατανομής πόρων μπορούν να εφαρμόσουν αυτά τα manifests. Μια σημαντική παράμετρος είναι το exclusivenessLevel, το οποίο καθορίζει το επίπεδο απομόνωσης των πόρων (None, CPU, Core, Socket, NUMA), με στόχο τη βελτιστοποίηση της επίδοσης για latency-critical εφαρμογές.

Ο Controller Manager είναι υπεύθυνος για τη διαχείριση των `NodeCPUTopology` και `PodCPUBinding` resources, συγχρονίζοντας την τρέχουσα κατάσταση του cluster με την επιθυμητή. Επικυρώνει και εφαρμόζει δεσμεύσεις CPU για Pods, αποτρέποντας μη έγκυρες διαμορφώσεις και παράγοντας αντίστοιχα συμβάντα στο API server.

Ο Daemon εκτελείται ως DaemonSet και παρέχει πληροφορίες CPU τοπολογίας και μηχανισμούς δέσμευσης πόρων μέσω δύο υπηρεσίες gRPC. Αναλαμβάνει τη δυναμική διαχείριση CPU και μνήμης, διασφαλίζοντας τη σωστή κατανομή των διαθέσιμων πόρων σε κάθε κόμβο.

Ο Scheduler χρησιμοποιεί το `WorkloadAware` plugin για τη βελτιστοποίηση της κατανομής Pods, λαμβάνοντας υπόψη το είδος του φόρτου εργασίας (CPU-bound, Memory-bound, IO-bound, Best-effort). Προσφέρει πολιτικές κατανομής όπως `MaximumUtilization` και `Balanced` για ευέλικτη διαχείριση των διαθέσιμων πόρων.

```yaml
apiVersion: cslab.ece.ntua.gr/v1alpha1
kind: PodCPUBinding
metadata:
  finalizers:
  - cslab.ece.ntua.gr/pod-cpu-binding-finalizer
  name: streamcluster-4-pcb
  namespace: benchmarks
spec:
  cpuSet:
  - cpuID: 0
  - cpuID: 2
  - cpuID: 4
  - cpuID: 6
  exclusivenessLevel: NUMA
  podName: streamcluster-4
status:
  nodeName: node-4
  resourceStatus: Applied
```

Listing 1.1: Παράδειγμα `PodCPUBinding` Manifest

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: graph-analytics-8
  labels:
  cslab.ece.ntua.gr/workload-type: cpu-bound
spec:
  schedulerName: maestro
  containers:
    - name: graph-analytics-container
      image: graph-analytics:latest
      imagePullPolicy: Always
      command: [ "/root/entrypoint.sh" ]
      args: [ "pr", "--driver-memory", "4g", "--executor-memory", "9g" ]
      resources:
      requests:
          cpu: "8"
          memory: "14Gi"
      limits:
          cpu: "8"
          memory: "14Gi"
```

Listing 1.2: Παράδειγμα Δρομολόγησης Εφαρμογής με Maestro

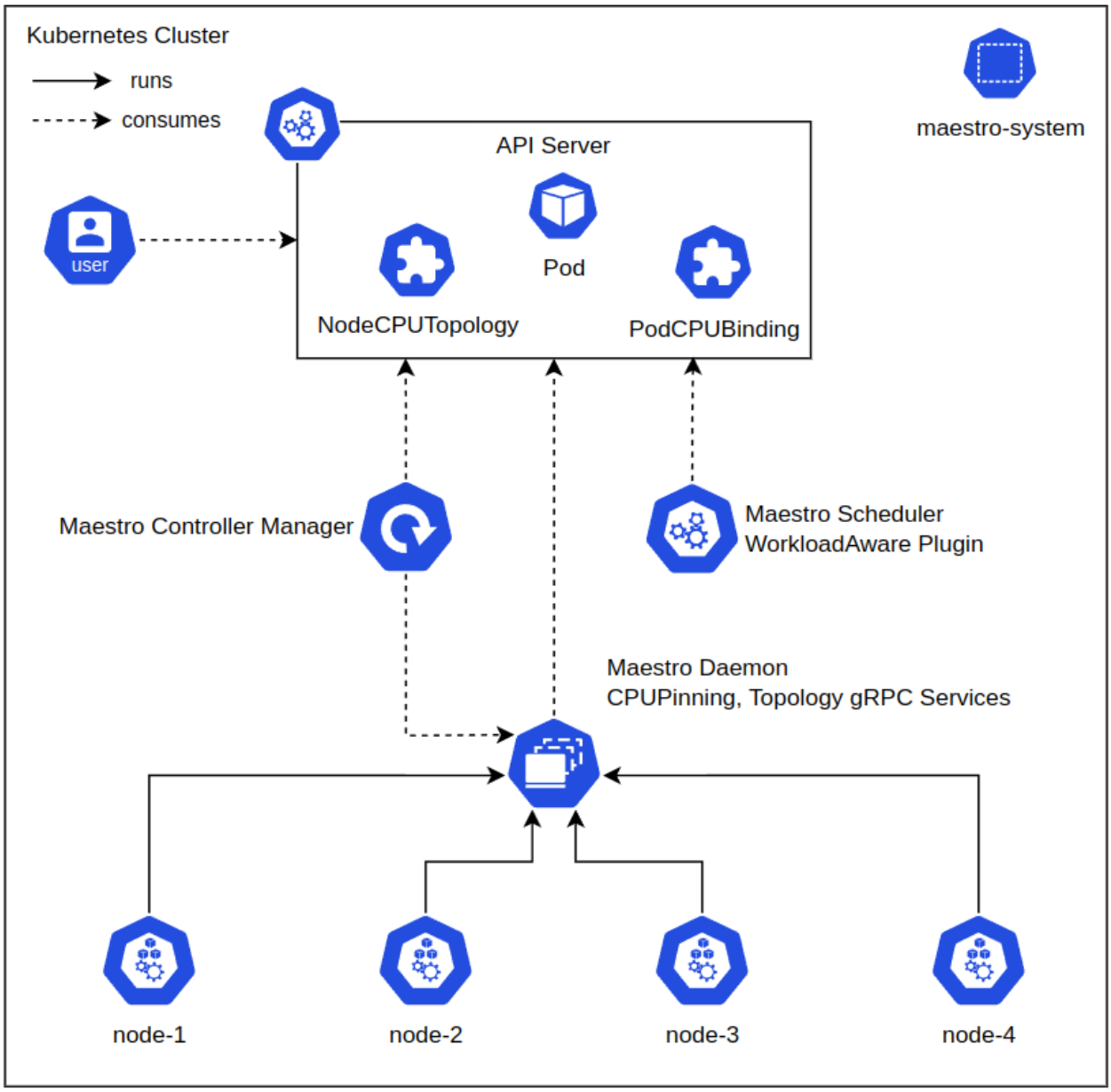


Figure 1.5.1: Αρχιτεκτονική του μηχανισμού Maestro

1.6 Πειραματική Διάταξη

Στην πειραματική μας διάταξη χρησιμοποιήσαμε τέσσερις εικονικές μηχανές με Ubuntu Server 18.04, κατανεμημένες σε δύο φυσικούς υπολογιστές. Το cluster Kubernetes (v1.31.0) δημιουργήθηκε με kubeadm, ενώ για απαιτητικά workloads προσθέσαμε έναν ισχυρότερο dual-socket worker. Η κατανομή των CPU και μνήμης έγινε με σταθερό CPU pinning, διασφαλίζοντας τη βέλτιστη επίδοση για multi-NUMA workloads. Τα χαρακτηριστικά των εικονικών μηχανών παρουσιάζονται στον παρακάτω πίνακα.

Τα benchmarks που χρησιμοποιούμε αξιολογούν την επίδοση του συστήματός μας σε διαφορετικά υπολογιστικά φορτία. Η σουίτα CloudSuite επικεντρώνεται στην προσομοίωση πραγματικών cloud εφαρμογών. Περιλαμβάνει το Data Serving, το οποίο μετρά την επίδοση της NoSQL βάσης δεδομένων Cassandra, το Web Serving, που συνδυάζει MariaDB, Memcached και Elgg για την προσομοίωση ενός web server, και το Media Streaming, που χρησιμοποιεί έναν Nginx server για μετάδοση βίντεο. Επιπλέον, το Graph Analytics εκτελεί ανάλυση μεγάλων

Table 1.1: Διαμόρφωση Εικονικών Μηχανημάτων

Host	CPU Model	VM Name	Sockets	Cores/Socket	vCPUs	Memory (GB)	NUMA Nodes	OS
termi10	Intel Xeon E5645	node-1	1	2	4	16	1	Ubuntu Server 18.04
		node-2	1	4	8	16	1	Ubuntu Server 18.04
		node-3	1	4	8	16	1	Ubuntu Server 18.04
		nfs	1	2	4	8	1	Ubuntu Server 18.04
termi9	Intel Xeon E5645	node-4	2	10	20	64	2	Ubuntu Server 18.04

δεδομένων με Spark και GraphX, ενώ το In-Memory Analytics αξιολογεί αλγορίθμους collaborative filtering με Apache Spark και MLlib. Το PARSEC έχει σχεδιαστεί για την αξιολόγηση πολυπύρηνων αρχιτεκτονικών (CMPs) και περιλαμβάνει ποικιλία εφαρμογών από διαφορετικούς τομείς. Μεταξύ αυτών, το blackscholes υπολογίζει τιμές χρηματοοικονομικών επιλογών, το canneal βελτιστοποιεί δρομολόγηση σε σχεδιασμό chip, το fluidanimate προσομοιώνει ρευστά για animation, και το ferret εκτελεί αναζήτηση ομοιότητας σε βάσεις εικόνων. Το facesim προσομοιώνει κινήσεις προσώπου, ενώ το x264 κωδικοποιεί βίντεο με τον H.264/AVC αλγόριθμο. Αυτά τα benchmarks εστιάζουν σε παραλληλισμό, μοτίβα πρόσβασης στη μνήμη και απαιτήσεις συγχρονισμού, αξιολογώντας τη συνολική επίδοση πολυπύρηνων επεξεργαστών. Το STREAM είναι ένα benchmark ειδικά σχεδιασμένο για τη μέτρηση του θεωρητικού μέγιστου memory bandwidth ενός συστήματος. Περιλαμβάνει τέσσερις βασικές λειτουργίες σε μεγάλα διανύσματα: Copy, Scale, Sum και Triad. Αυτές οι λειτουργίες αποφεύγουν την επαναχρησιμοποίηση δεδομένων από την cache ή τους καταχωρητές, μετρώντας το raw bandwidth της κύριας μνήμης. Το STREAM βοηθά στην ανίχνευση πιθανών bottlenecks στη μνήμη που επηρεάζουν την επίδοση εφαρμογών με έντονη χρήση δεδομένων.

Για την αποτίμηση του μηχανισμού μας, κατηγοριοποιήσαμε τις εφαρμογές σε τέσσερεις βασικές κατηγορίες, με βάση τον τρόπο με τον οποίο αξιοποιούν πόρους επεξεργαστή και μνήμης κατά την εκτέλεσής τους, σε αρχιτεκτονικές NUMA.

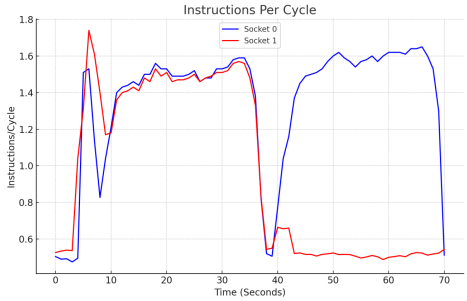
1. Packed-Friendly εφαρμογές λειτουργούν καλύτερα όταν τα νήματα του προγράμματος παραμένουν σε έναν NUMA κόμβο, μειώνοντας την επικοινωνία μεταξύ κόμβων. Χαρακτηρίζονται από υψηλή υπολογιστική ένταση και περιορίζονται από τη δυνατότητα του επεξεργαστή να εκτελεί αριθμητικές πράξεις, όπως προσομοιώσεις Monte Carlo και αλγόριθμοι δυναμικού προγραμματισμού.

2. Spread-Friendly εφαρμογές επωφελούνται από τη διασπορά των νημάτων σε πολλούς NUMA κόμβους, ώστε να αξιοποιήσουν αυξημένο memory bandwidth. Αυτές οι εφαρμογές είναι κυρίως memory-bound και υποφέρουν από χαμηλή υπολογιστική ένταση, όπως πράξεις σε μεγάλους διανύσματα.

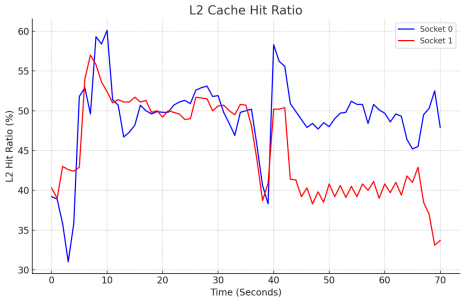
3. Isolation-Friendly εφαρμογές αποδίδουν καλύτερα όταν εκτελούνται ανεξάρτητα, χωρίς ανταγωνισμό για πόρους όπως CPU, cache και memory bandwidth. Αυτές περιλαμβάνουν latency-sensitive εργασίες, όπως κρυπτογραφία και real-time επεξεργασία δεδομένων.

4. Agnostic εφαρμογές παρουσιάζουν παρόμοια επίδοση ανεξάρτητα από τη διαρρύθμιση των πόρων τους. Είναι ισορροπημένες μεταξύ υπολογιστικής και μνημονικής έντασης, γεγονός που τις καθιστά κατάλληλες για δυναμικά περιβάλλοντα κατανομής πόρων.

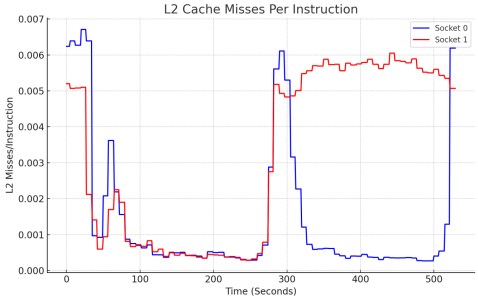
Για την κατηγοριοποίηση των εφαρμογών, προτείνεται ένας αλγόριθμος που συγκρίνει τις εκτελέσεις τους σε διαφορετικές διαμορφώσεις και υπολογίζει τις καθυστερήσεις (slowdown). Αυτή η ταξινόμηση επιτρέπει τη βελτιστοποίηση της εκτέλεσης μέσω προσαρμοσμένου προγραμματισμού και κατανομής πόρων, εξασφαλίζοντας χαμηλότερο slowdown και καλύτερη συνολική επίδοση.



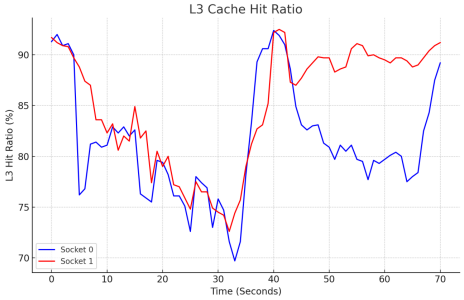
(a) Instructions Per Cycle



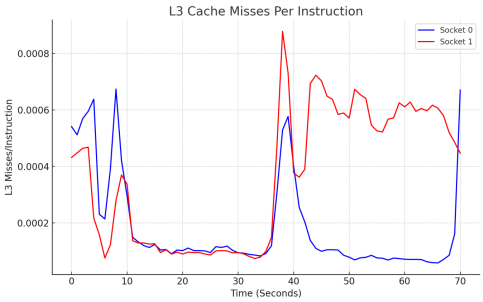
(b) L2 Cache Hit Ratio



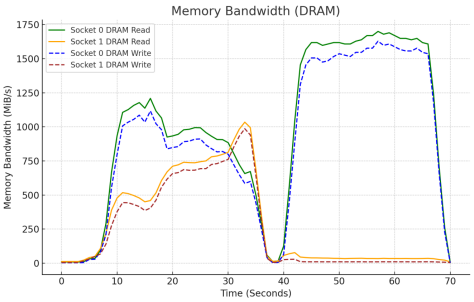
(c) L2 Cache Misses Per Instruction



(d) L3 Cache Hit Ratio



(e) L3 Cache Misses Per Instruction



(f) Memory Bandwidth (DRAM)

Figure 1.6.1: Μετρικές επίδοσης δύο συνεχόμενων εκτελέσεων του benchmark `in-memory-analytics` με δύο νήματα. Στην πρώτη εκτέλεση τα νήματα είναι κατανεμημένα στα διαθέσιμα NUMA nodes, μειώνοντας το απαιτούμενο memory bandwidth και διπλασιάζοντας τα instructions per cycle.

1.7 Αποτίμηση

Παρακάτω παραθέτουμε τα αποτελέσματα του classification που εφαρμόσαμε στα benchmarks που αναφέραμε, εκτελώντας τα κάτω από διαφορετικές συνθήκες παραχώρησης πόρων και συγκρίνοντας τις καθυστερήσεις (slowdowns) που παρουσιάζουν σε κάθε διάταξη.

Benchmark	Packed-Friendly	Spread-Friendly	Classification
canneal-2	0.78	1.59	Packed Friendly
canneal-4	0.83	1.04	Packed Friendly
canneal-8	0.90	0.96	Packed Friendly
ferret-2	0.87	0.96	Packed Friendly
ferret-4	0.87	0.92	Packed Friendly
ferret-8	1.09	0.84	Spread Friendly
fluidanimate-2	0.80	0.98	Packed Friendly
fluidanimate-4	0.98	1.00	Agnostic
fluidanimate-8	1.15	0.88	Spread Friendly
freqmine-2	0.88	0.94	Packed Friendly
freqmine-4	0.96	0.95	Agnostic
freqmine-8	1.35	1.00	Spread Friendly
ga-2	0.98	0.98	Agnostic
ga-4	1.01	0.98	Agnostic
ga-8	1.68	0.91	Spread Friendly
in-mem-2	0.98	1.02	Agnostic
in-mem-4	0.98	1.01	Agnostic
in-mem-8	1.31	0.96	Spread Friendly
stream-2	0.83	0.94	Packed Friendly
stream-4	1.44	0.94	Spread Friendly
stream-8	1.16	0.97	Spread Friendly
streamcluster-2	0.79	0.98	Packed Friendly
streamcluster-4	0.86	0.98	Packed Friendly
streamcluster-8	0.91	0.85	Spread Friendly

Table 1.2: Κατηγοριοποίηση των Benchmarks σε Τρεις Βασικές Κατηγορίες

Κατασκευάσαμε πολλαπλά σενάρια συνύπαρξης εφαρμογών με διαφορετικά χαρακτηριστικά. Δεδομένου ότι τα benchmarks δρομολογήθηκαν από τον scheduler μας στον ίδιο testing κόμβο, συγκρίναμε την απόδοσή τους κάτω από την επίδραση της προεπιλεγμένης παραχώρησης πόρων του kubelet, τις στατικές πολιτικές SpreadFriendly και PackedFriendly, και της επιλογής που έκανε ο Maestro ύστερα από τον χαρακτηρισμό των εφαρμογών. Επιλέξαμε μια ποικιλία benchmarks με διαφορετικά μεγέθη εισόδων, εκτελώντας τα με δύο, τέσσερα και οκτώ threads, ώστε να αναλύσουμε την επίδραση των παραμέτρων εκτέλεσης στην συνολική αποδοτικότητα.

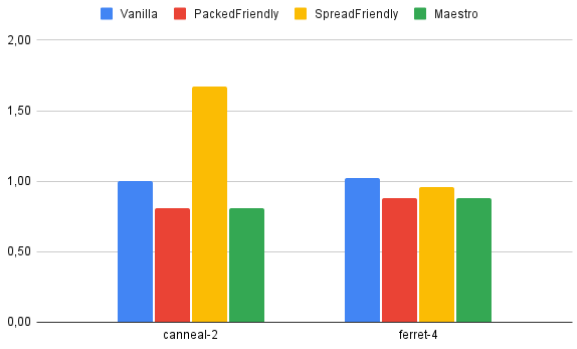


Figure 1.7.1: PackedFriendly vs. PackedFriendly (1)

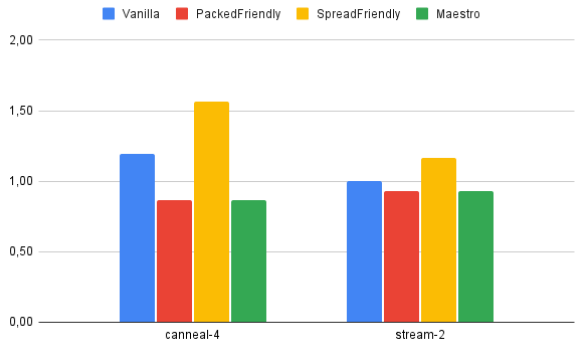


Figure 1.7.2: PackedFriendly vs. PackedFriendly (2)

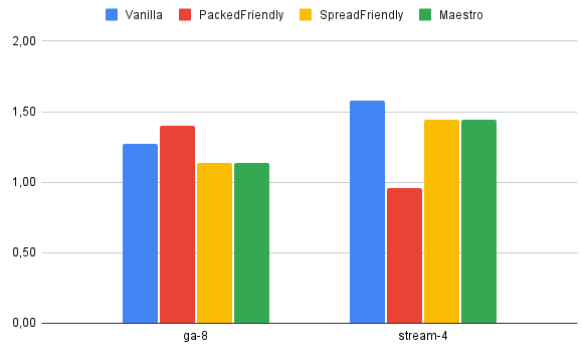


Figure 1.7.3: SpreadFriendly vs. SpreadFriendly (1)

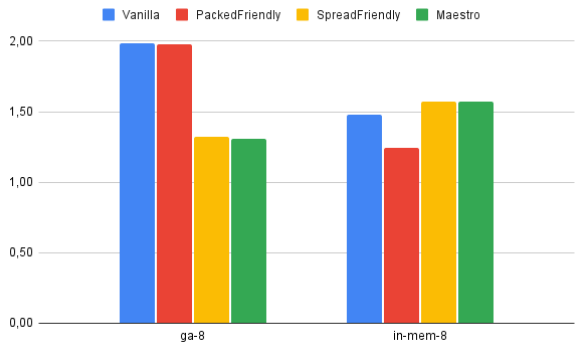


Figure 1.7.4: SpreadFriendly vs. SpreadFriendly (2)

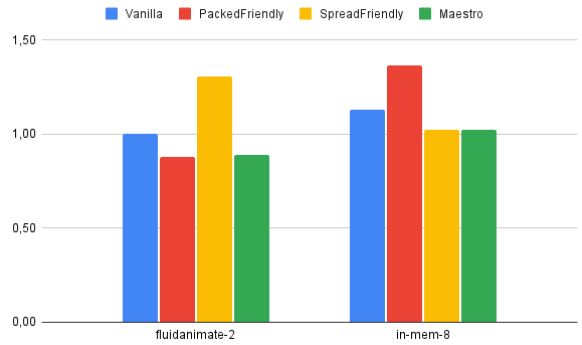


Figure 1.7.5: PackedFriendly vs. SpreadFriendly (1)

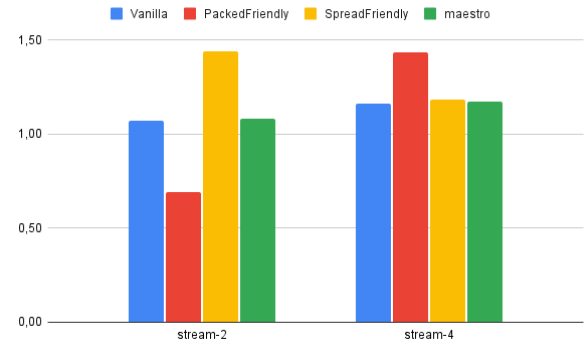


Figure 1.7.6: PackedFriendly vs. SpreadFriendly (2)

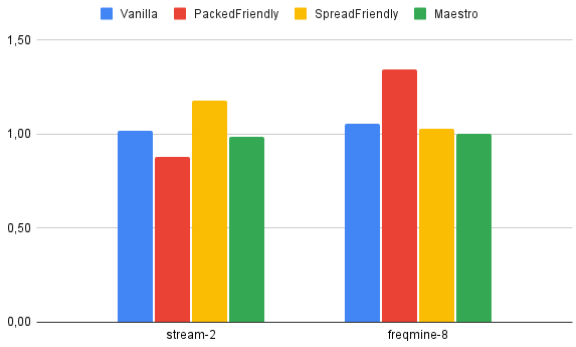


Figure 1.7.7: PackedFriendly vs. SpreadFriendly (3)

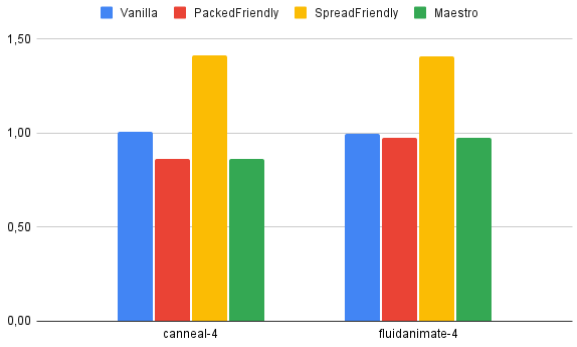


Figure 1.7.8: PackedFriendly vs. Agnostic (1)

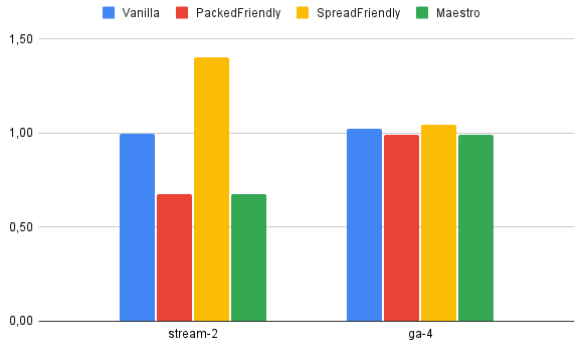


Figure 1.7.9: PackedFriendly vs. Agnostic (2)

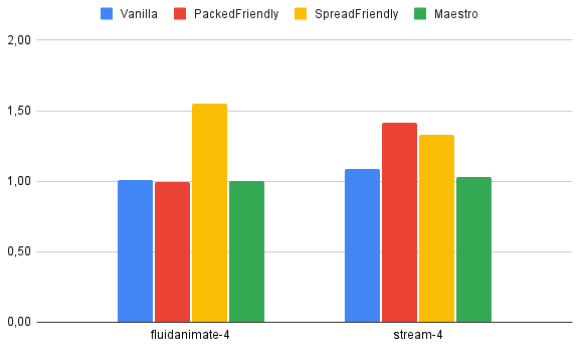


Figure 1.7.10: SpreadFriendly vs. Agnostic (1)

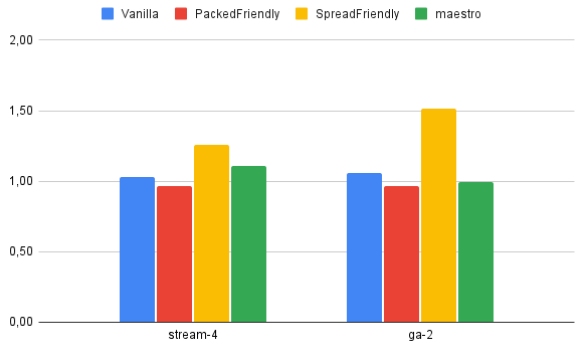


Figure 1.7.11: SpreadFriendly vs. Agnostic (2)

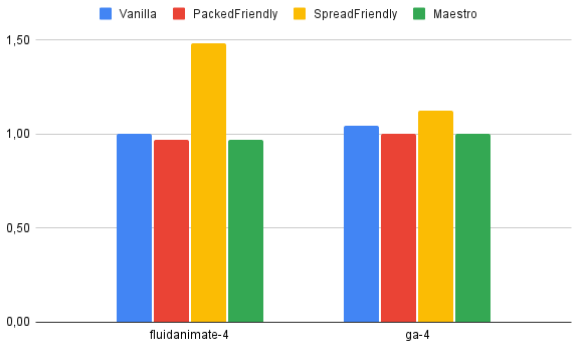


Figure 1.7.12: Agnostic vs. Agnostic

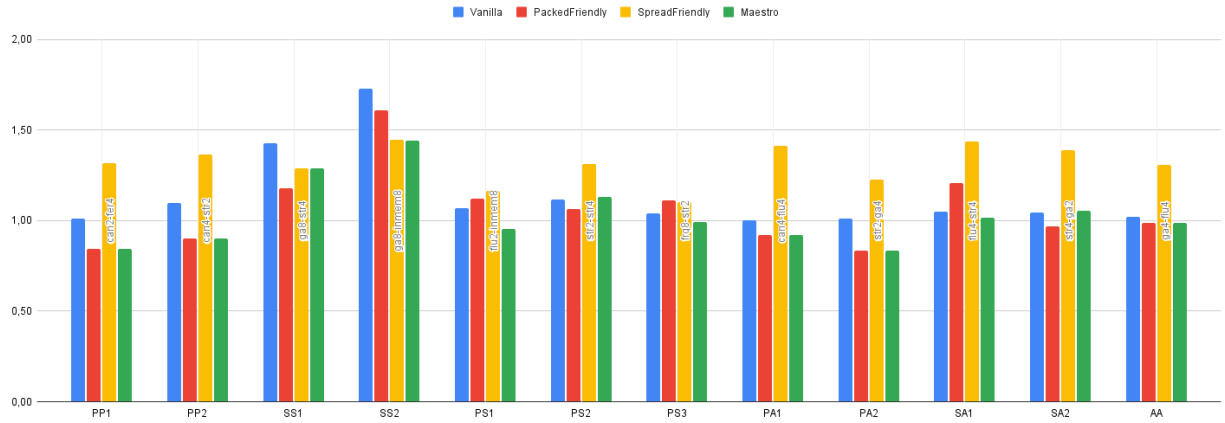


Figure 1.7.13: Συγκεντρωμένα Αποτελέσματα

Το διάγραμμα 1.7.13 παρουσιάζει τα συνολικά αποτελέσματα από όλα τα σενάρια που αναλύθηκαν, συνοψίζοντας τον μέσο slowdown και των δύο εφαρμογών για κάθε μία από τις τέσσερις διαμορφώσεις. Το διάγραμμα αποδεικνύει ότι ο Maestro διατηρεί σταθερά χαμηλότερο μέσο slowdown σε σύγκριση με τις υπόλοιπες στρατηγικές παραχώρησης πόρων. Σε αρκετές περιπτώσεις, η βελτίωση είναι σημαντική, με μείωση του slowdown κατά 29%, 20%, 18% και 17% έναντι του vanilla Kubernetes. Τα αποτελέσματα επιβεβαιώνουν ότι ο Maestro λαμβάνει κατά κανόνα ορθές αποφάσεις κατανομής CPU, μειώνοντας τον χρόνο εκτέλεσης μέσω τεχνικών CPU pinning, isolation, και κατανομής των threads στα sockets.

1.8 Συζήτηση και Μελλοντικό Έργο

Στην παρούσα εργασία, αναλύονται λεπτομερώς τα φαινόμενα της μείωσης επίδοσης και της υποχρησιμοποίησης πόρων σε συστήματα cloud. Τα multi-tenant συστήματα αντιμετωπίζουν προκλήσεις όσον αφορά την απομόνωση και τη διαχείριση των πόρων, γεγονός που οδηγεί σε μη βέλτιστη επίδοση. Οι πάροχοι cloud, προκειμένου να διατηρήσουν τα συμφωνημένα επίπεδα υπηρεσιών (SLAs), συχνά υποχρησιμοποιούν το διαθέσιμο υλικό, κάτι που έχει σημαντικές περιβαλλοντικές επιπτώσεις. Έρευνα διεξάγεται συνεχώς για τη βελτίωση των πολιτικών κατανομής πόρων, με στόχο τη βελτίωση της επίδοσης και της αποδοτικότητας. Το Kubernetes, ως ο πιο διαδεδομένος ενορχηστρωτής containers, δεν λαμβάνει υπόψη τα ιδιαίτερα χαρακτηριστικά κάθε φόρτου εργασίας, γεγονός που περιορίζει την αποτελεσματικότητα των αποφάσεων κατανομής. Με την υλοποίηση ενός προσαρμοσμένου μηχανισμού, αποδείχθηκε ότι μια απλοϊκή στρατηγική χαρακτηρισμού εφαρμογής και κατανομής πόρων μπορεί να βελτιώσει την επίδοση έναντι των προεπιλεγμένων μηχανισμών του Kubernetes, επιτυγχάνοντας έως και 29% ταχύτερη εκτέλεση.

Η έρευνα αυτή αποτελεί μόνο το πρώτο βήμα προς την ανάπτυξη ενός εξελιγμένου μηχανισμού κατανομής πόρων, ο οποίος θα μπορεί να πραγματοποιεί τόσο στατική όσο και δυναμική κατανομή πόρων. Για τη βελτίωση της κατηγοριοποίησης των φόρτων εργασίας, μπορεί να ενσωματωθεί ένας πιο προηγμένος μηχανισμός κατηγοριοποίησης εφαρμογών (workload profiling) που θα βασίζεται σε hardware performance counters, low-level system metrics και τεχνικές μηχανικής μάθησης. Μέσω εκπαίδευσης νευρωνικών δικτύων με δεδομένα από offline profiling, είναι εφικτό να ταξινομηθούν οι εφαρμογές με βάση τα μοτίβα χρήσης μνήμης, τα cache misses, καθώς και τη συμπεριφορά τους υπό συνθήκες συνύπαρξης (co-execution). Επιπλέον, η ανίχνευση φαινομένων παρεμβολών σε πραγματικό χρόνο μπορεί να επιτευχθεί με ενσωμάτωση online monitoring εργαλείων, όπως Intel PCM και eBPF-based tracing, ώστε να παρθούν αποφάσεις σε σενάρια όπως κορεσμό memory bandwidth ή μείωση των instructions per cycle (IPC).

Για μελλοντική έρευνα, η ενσωμάτωση ενός ενισχυτικού μηχανισμού μάθησης (reinforcement learning) μπορεί να επιτρέψει την προσαρμοστική ανακατανομή των πόρων με βάση τα δεδομένα παρακολούθησης κατά την εκτέλεση, βελτιώνοντας δυναμικά την επίδοση των φόρτων εργασίας. Επιπλέον, θα μπορούσε να μελετηθεί η ανάπτυξη ενός lightweight agent εντός των Kubernetes nodes, ο οποίος θα συλλέγει χαμηλού επιπέδου μετρήσεις και θα επικοινωνεί με τον scheduler για προσαρμογές σε πραγματικό χρόνο. Αυτή η αρχιτεκτονική

μπορεί να αναπτυχθεί σε cloud-native περιβάλλοντα, χωρίς την ανάγκη εξωτερικών προγραμμάτων, όπως το Intel PCM στους hypervisors. Η αξιοποίηση αυτών των τεχνικών μπορεί να βοηθήσει τους cloud providers να μειώσουν τη σπατάλη υπολογιστικών πόρων, ελαχιστοποιώντας παράλληλα το περιβαλλοντικό αποτύπωμα μέσω πιο αποδοτικής κατανομής φόρτων εργασίας στα datacenters.

Chapter 2

Introduction

2.1 Cloud Computing

The Cloud is an emerging technology that is constantly expanding in every modern field of computation. As defined by the National Institute of Standards and Technology (NIST), Cloud Computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction [15]. The cloud consists of a network of datacenters, each housing thousands of interconnected computers designed to perform various tasks. These datacenters enable users to access a wide range of applications, platforms, and services over the Internet. In contrast with the preceeding paradigm of On-Premise Computing, organizations can leverage scalable, flexible, virtualized resources over the Internet for their computational needs, eliminating the cost and maintenance complexity of physical servers in their own facilities. Furthermore, it promotes "pay-as-you-go" pricing schemas, which can result to smaller upfront investments while limiting and scaling resources based on their current demand. Flexible pricing and resource elasticity along with on-demand, broad network access, scalability and virtualization are the essential characteristics of the cloud paradigm. As the adoption of Cloud Computing has become a norm on a large-scale enterprise level, the improvement of resource sharing is a logical next step.

Taxonomy of Cloud Services

Cloud Computing primarily comprises three types of services: Software as a Service (SaaS), Platform as a Service (PaaS), and Infrastructure as a Service (IaaS).

- **SaaS (Software as a Service)**: Consumers can utilize the provider's applications via the web, allowing access to software without installation or maintenance, and enabling seamless updates and scalability. SaaS applications are typically delivered on a subscription basis, offering flexibility and convenience across various devices.
- **PaaS (Platform as a Service)**: Built on cloud infrastructure, PaaS provides consumers with tools and services for developing and deploying applications using supported platforms and programming languages. It offers a higher-level platform, such as Google App Engine and Microsoft Azure, enabling developers to create customized applications without managing the underlying cloud infrastructure.
- **IaaS (Infrastructure as a Service)**: IaaS provides consumers with virtual computing resources, including Virtual Machines (VMs) (e.g., Amazon EC2), storage (e.g., Amazon S3), and other essential computing capabilities. This model allows consumers to deploy and run arbitrary software, including applications and operating systems, on these virtual resources, offering significant flexibility and control.

2.2 Virtualization Techniques

Prior to the existence of virtualization, organizations used physical servers (bare-metal machines) to run their applications. However, this lead to under-utilization of computing resources and made it difficult to move applications between different environments. Virtualization refers to a set of software technologies that allow software to run on virtual hardware.

Hypervisor-Based Virtualization

Traditional virtualization enables multiple Virtual Machines (VMs) to run on a single physical server. Each VM acts like a complete machine with its own operating system running on virtual hardware, on which the user has the flexibility to adjust computational resources such as CPU, memory and storage. This allows better scalability, portability and reduced hardware costs. Furthermore, the separate operating systems create clear boundaries between the applications, making VMs the industry standard for application multi-tenancy on the cloud.

Containerization

A container is a virtual runtime environment that operates on top of a single operating system kernel, emulating an operating system rather than the underlying hardware. While containers often run within a VM, they do not require the provisioning of an operating system. Containers improve the sharing of hardware resources by eliminating the hypervisor infrastructure layer.

A container engine is a managed environment for deploying containerized applications. The container engine allocates cores and memory to containers, enforces spatial isolation and security, and provides scalability by enabling the addition of containers. Docker [3] is a container platform that provides solutions for most layers of the container technology stack. These layers include the container engine, scheduling, orchestration, image management, and configuration management. Due to Docker's widespread adoption and comprehensive range of solutions, the platform has become synonymous with container technology.

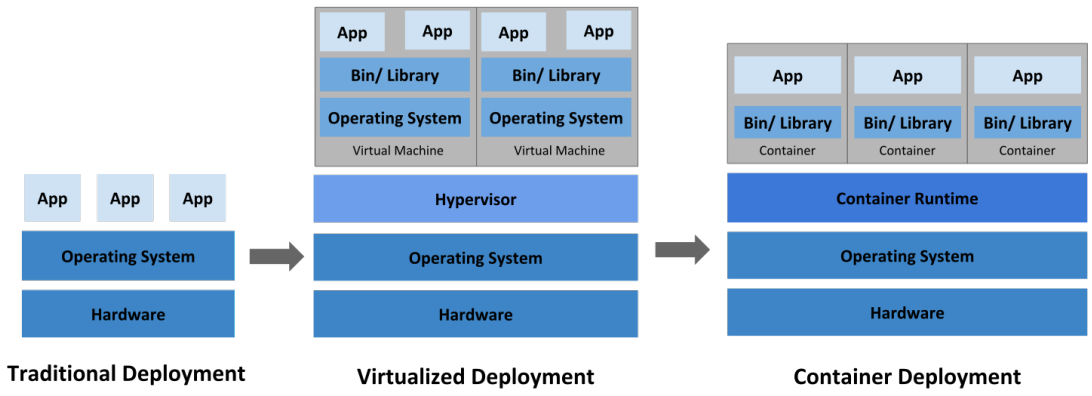


Figure 2.2.1: Evolution of Virtualization

Hybrid Container Architecture

A hybrid container architecture is an architecture combining virtualization by both virtual machines and containers, i.e., the container engine and associated containers execute on top of a virtual machine. Use of a hybrid container architecture is also known as hybrid containerization, and it's the most common practice for cloud servers.

Advantages of Containers Over Virtual Machines

- **Hardware Costs**: Containers enhance hardware utilization by allowing co-located software to fully utilize the concurrency of multi-core architectures. By eliminating the overhead of virtual machines,

such as excessive storage and resource usage, containers offer a lightweight and cost-effective solution for efficient application deployment.

- **Scalability**: A single container engine can efficiently manage large numbers of containers, enabling additional containers to be created as needed.
- **Spatial Isolation**: The Linux kernel features `cgroups` and `namespaces` allow us to provide each container with its own resources (e.g., core processing unit, memory, and network access) and container-specific namespaces.
- **Performance**: Compared to VMs, containers increase performance (throughput) of applications because they do not emulate the underlying hardware.
- **Storage**: Compared to virtual machines, containers are more lightweight in terms of storage, as the applications within containers can share binaries and libraries, reducing duplication and saving space.
- **Security**: When dealing with a cybersecurity compromise, the container engine is able to restore the container to its healthy state while applying various security software rules on every hosted container.

2.3 Container Orchestration and Kubernetes

Container Orchestration refers to automating the deployment, scaling and management of containerized applications within a cluster. The orchestrator interacts with the container runtime and handles tasks like launching, updating, moving, scaling, and removing containers.

Kubernetes [9] is an open-source container orchestrator that has become the standard for enterprise-level applications. Beyond its core functions of deploying, scaling, and managing containers, it automates critical infrastructure tasks like service discovery, load balancing, storage management, rollbacks, and self-healing. While Kubernetes is optimized for deploying microservices, its flexibility allows users to deploy a wide range of applications, including data analytics and machine learning pipelines, as well as scientific high-performance workloads, among many others.

Kubernetes also ships with highly extensible, developer-centric interfaces that enable developers to integrate their custom business requirements in their clusters. These extension points include implementing custom scheduling logic, creating custom resources on the API server and their respective clients, as well as device plugins which enable the cluster administrator to leverage vendor-specific hardware, such as GPUs, within their containers.

Although Kubernetes is a highly complex system, further research is needed to optimize the allocation of CPU and memory resources to applications. It currently relies on typical memory and CPU usage percentages of worker nodes, which are not always accurate. This can result in resource contention and throttling of applications, leading to sub-optimal performance. The default scheduler's decisions are not always optimal, as they are not based on the real-time state of the cluster or the specific runtime characteristics of the applications being scheduled. Additionally, the daemon running inside each worker node lacks a sophisticated mechanism for assigning exclusive resources to applications, causing them to compete for shared resources and interfere with each other, ultimately degrading performance.

2.4 Resource Utilization Concerns in Datacenters

Cloud Service Providers (CSPs) traditionally rely on continuous hardware upgrade of their IT facilities, in combination with simplistic scheduling and resource allocation policies to minimize the risk of their applications under-performing. Nowadays, such practice is considered unsustainable especially given the rapid shift from on-premise to cloud environments and slowingly recessing technology scaling [14]. Having to handle a very large amount of diverse workloads, CSPs should consider carefully how to provision resources in order to reduce unnecessary cost, while keeping performance within the required margin. While the breakthrough of multi-core processors enabled the CSPs to co-locate multiple tenants' services in the same host machine, reducing the need to aggressively invest in new hardware, there is still a lot to improve regarding resource utilization on multi-core systems.

Datacenter Server Architecture

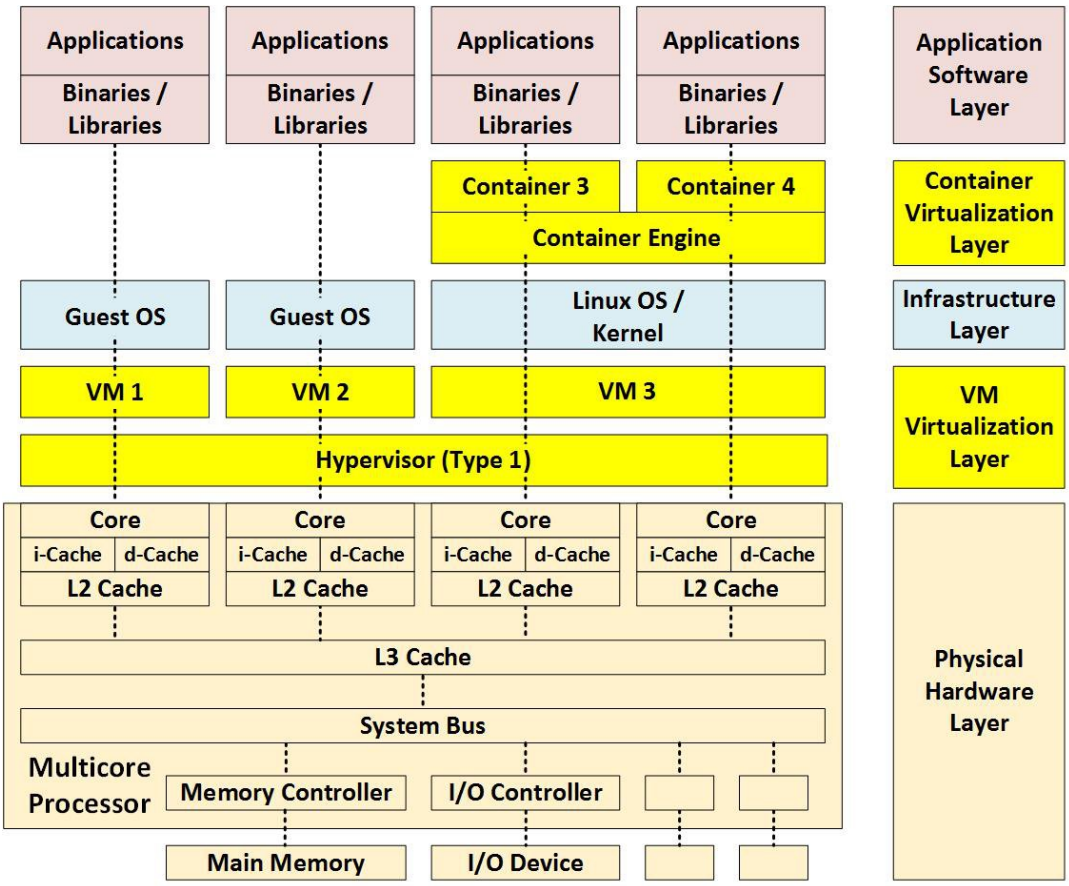


Figure 2.4.1: Hybrid Cloud Server Architecture

Figure 2.4.1 provides a detailed illustration of the architectural layers of a datacenter machine with a single multi-core socket. Each multi-core processor is directly linked to a dedicated memory node, and each physical core has its own isolated memory space for Level 1 (L1) and Level 2 (L2) caches. In the diagram, the L1 cache is split into the instruction cache (i-Cache), which holds program instructions, and the data cache (d-Cache), which stores the data needed for execution. All physical cores within the socket share a common Level 3 (LLC) cache. Additionally, if Simultaneous Multi-Threading (SMT) is enabled, each physical core may consist of two logical cores (also known as threads or CPUs), which share the core's L1 and L2 caches.

A cloud server typically operates within a Non-Uniform Memory Access (NUMA) architecture, where multiple sockets are each linked to their own memory node. In this setup, each processor in a NUMA node has faster access to its local memory compared to the memory attached to other processors in different NUMA nodes as the cross-node communication through the interconnect bus is significantly slower than local memory access. This architecture is critical for optimizing performance in workloads that require large amounts of memory and high processing power, such as in-memory databases and high-performance computing (HPC) tasks. Cloud Service Providers like Amazon Web Services (AWS), Google Cloud Platform (GCP), and Microsoft Azure offer computing instances that comprise of one, two, or four NUMA nodes depending on the instance type and workload requirements.

Multi-Tenancy, Workload Consolidation and Shared Resources

The term Multi-Tenancy refers to the resource sharing between multiple organizations (tenants) within the infrastructure of a specific Cloud Service Provider, and the associated complexity and challenges that emerge. Since any resource object is reusable in the Cloud infrastructure, CSPs have to carefully secure them and

eliminate any possible vulnerability that might occur, such as data leakage, while minimizing cost and keeping up with the customer's requirements. While security concerns are, and should be, one of the most important considerations, CSPs have to optimize their available resources to avoid under-utilization of their machines and reduce financial, as well as environmental cost. This includes being able not only to efficiently provision the requested services, but also to dynamically scale the amount of resources allocated to each tenant at runtime.

Workload Consolidation, is an effective method to achieve multi-tenancy in the cloud, enabling the CSP to place multiple workloads in the same server, increasing resource utilization. The constraints on performance metrics, such as throughput and latency, are stated in the Service-Level Agreement which is associated with each tenant, and their violation results in SLA penalties, and client dissatisfaction.

Quality of Service and Service-Level Agreement Guarantees

Quality of Service (QoS) refers to the performance level of a service as perceived by the user. In the context of Cloud Computing, QoS encompasses various performance metrics such as availability, latency, throughput, and reliability. Ensuring high QoS is crucial for Cloud Service Providers to meet customer expectations and maintain competitive advantage.

Despite these efforts, challenges remain in consistently meeting SLA guarantees, particularly in multi-tenant environments where resources are shared among multiple customers. Interference from "noisy neighbors" and resource contention can lead to performance degradation, making it difficult to maintain the promised QoS levels. Ongoing research and advancements in resource management, scheduling, and optimization techniques are essential to address these challenges and improve the overall QoS in cloud environments.

Kubernetes classifies each running Pod into a specific QoS class that is inferred from the resource requests and limits of the Containers within the Pod. The three QoS classes of Kubernetes, `Guaranteed`, `Burstable`, and `BestEffort` guide the orchestrator to make the least destructive decisions when evicting applications due to node pressure. While this is a valid approach for prioritizing applications and guaranteeing resources to the Pods, it often leads to under-utilization and false scheduling decisions, as the CPU and memory utlization are naive indicators and do not always reflect the real utilization state of the cluster.

Energy Consumption and Cost Efficiency

While cloud systems offer a range of benefits, their resource utilization and environmental impacts raise concerns. A key concern is the substantial energy consumption of data centers, which are fundamental to cloud infrastructure. Globally, data centers consume an estimated 460 terawatt-hours (TWh) of electricity, and this figure is projected to double by 2026, reaching a consumption level equivalent to Japan's total electricity use [4]. The inefficient allocation of resources within cloud environments can worsen this already significant energy demand, contributing to higher operational costs and a larger carbon footprint. For example, data centers in Ireland, a growing hub for this sector, consumed 17% of the country's total electricity in 2022, a figure expected to double by 2026, potentially reaching 32% of national electricity consumption driven by data center expansion and increasing AI applications [4]. A huge need emerges for optimization tools and techniques, such as rightsizing and runtime optimization, to ensure efficient resource utilization and mitigate the environmental impact of cloud systems [20]. Moreover, the integration of AI and automation is crucial for enhancing efficiency and enabling proactive optimization strategies. Transitioning to a sustainable cloud ecosystem requires a collective effort to prioritize resource efficiency and minimize the environmental burden associated with these powerful technologies.

2.5 Thesis Overview

In this thesis, we address the fundamental challenges of resource sharing during the co-execution of workloads on cloud systems. In chapter 2, we list related academic and enterprise work on cloud resource optimization. Following a detailed introduction on Kubernetes and its extension points for developers, in chapter 3, we enumerate, different performance degradation factors in HPC workloads and provide suggested methods to profile incoming applications based on their intrinsic characteristics, in chapter 4. In chapter 5, we techincally describe the different components of our custom solution and the scheduling and resource binding

algorithms that were implemented. In chapter 6 we present the experimental setup we created to develop, test, and monitor our custom Kubernetes solution for scheduling and allocating CPU and memory resources on workloads, based on the workload family the are classified as. In chapter 7, we evaluate our solution on our setup, by conducting single and multi-node experiments, while testing different scheduling strategies and features that were implemented. In the final chapter 8, we summarize the current work's findings and suggest possible corrections and extensions that would make our mechanism more efficient and more eligible for production cloud-native environments.

Chapter 3

Related Work

Running latency-critical workloads in multi-tenant cloud environments requires efficient resource management strategies, especially when considering scalability and performance requirements. Kubernetes, a widely adopted container orchestration platform, provides a default scheduling mechanism, but its capabilities often fall short in scenarios requiring fine-grained resource allocation, dynamic scaling, and interference-aware scheduling. This chapter reviews notable advancements in this area, including Kubernetes-based enterprise schedulers like Volcano and Koordinator, as well as academic research on fine-grained scheduling policies, autoscaling mechanisms, and load-aware optimization strategies.

3.1 Enterprise Solutions for Cloud Resource Management

Volcano

Volcano [22] is a cloud-native batch system built on Kubernetes designed for high-performance workloads, including machine learning, bioinformatics, and big data applications. It provides powerful batch scheduling capabilities that Kubernetes lacks. Volcano integrates with generalized domain frameworks like TensorFlow, Spark, PyTorch, and MPI, allowing users to run applications without needing significant modifications. Volcano supports diverse scheduling policies such as co-scheduling, fair-share, gang scheduling, and topology-aware scheduling. It also provides enhanced job management, supports multiple runtimes like Singularity and GPU accelerators, and offers robust monitoring with logging, metrics, and dashboards. Volcano is an incubating project of the Cloud Native Computing Foundation (CNCF) and has been adopted by numerous companies and institutions worldwide. It is designed to optimize the performance of compute-intensive jobs by converting them to Kubernetes workloads that are scheduled in batches.

Koordinator

Koordinator is a scheduling system that enhances Kubernetes by enabling the efficient co-location of diverse workloads including microservices, AI, and big data applications, aiming to improve resource utilization and workload performance [8]. It provides a QoS-based scheduling mechanism allowing different types of pods to run on the same node, and uses an independent field, `koordinator.sh/qosClass`, to define pod service quality. The ClusterColocationProfile blueprint facilitates the injection of Koordinator QoS and priority into Pods without modifying existing controllers. Koordinator also supports resource oversubscription to maximize utilization by reclaiming unused resources from high-priority pods for low-priority pods, with the SLO Controller dynamically adjusting the overcommitment ratio based on node status.

Koordinator also features load-aware scheduling that balances pod distribution across nodes, preventing under- or over-utilization by using real-time node metrics collected by the koordlet, a daemon running on each node. The system provides fine-grained resource orchestration and isolation to ensure the performance of latency-sensitive workloads while running batch jobs. The koordlet component is crucial for resource profiling, interference detection, and QoS management. Koordinator also has a flexible job scheduling mechanism for

supporting specialized workloads. The system is designed to be compatible with Kubernetes and can be used as a sidecar without invasive modifications. Furthermore, Koordinator includes monitoring, troubleshooting, and operational tools.

3.2 Academic Work on Efficient Resource Allocation in the Cloud

Several custom mechanisms have been developed to overcome the limitations of the default Kubernetes scheduler, which often results in suboptimal resource allocation and performance. These custom solutions leverage Kubernetes' extension points to implement more advanced scheduling algorithms that consider a broader range of factors such as I/O load, CPU utilization, and resource contention. These extension points, including the Filter (Predicates) and Scoring (Priorities) phases, allow for the integration of custom logic into the scheduling process, to address issues such as disk I/O bottlenecks, imbalanced CPU and disk I/O usage, and the need for more nuanced workload distribution. For example, the presentation on fine-grained cgroup resource scheduling in Kubernetes by Wang and Kan at the 2020 KubeCon and CloudNativeCon, shows that implementing cgroup based resource management can result in a 30% increase in insert TPS for MySQL workloads, a 10% increase in select TPS for MySQL workloads, and a 20-30% increase in QPS for Java/Go web applications with a 20% reduction in time consumption [23]. Below we will present a brief overview of academic work on fine-grained resource allocation and scheduling based on more sophisticated metrics and decision methods.

Virtual Machine Based Resource Management in the Cloud

Psomadakis et al. have proposed ACTiManager, a practical, interference-aware resource manager for cloud environments [16]. It operates without requiring offline application profiles and is transparent to applications. It has a two-tiered architecture, with ACTiManager.external handling VM allocation across servers and ACTiManager.internal managing fine-grained resource allocation within servers. ACTiManager characterizes VMs in a "quiet neighborhood" to build interference and healthy state models using Support Vector Machines and Principal Component Analysis techniques, respectively. It prioritizes applications using gold and silver pricing models and makes decisions that maximize datacenter profit. Experimental results show that ACTiManager is able to achieve a very good balance in metrics, excelling over alternative policies in the majority of configurations and achieving profit increases ranging from 12% to 49%.

Fine-Grained Scheduling Policies in Kubernetes Clusters

Liu et al. have explored fine-grained scheduling policies for containerized High Performance Computing (HPC) workloads within Kubernetes clusters [11]. The authors note that while containerization and Kubernetes are widely used in cloud computing, they are not optimized for the performance requirements of HPC applications. They argue that current scheduling methods do not consider application-specific information or the benefits of multi-container deployments for HPC workloads. The authors' motivation is to develop an optimized management framework that improves the performance of HPC workloads by enabling fine-grained deployment and leveraging containerization and orchestration technologies.

To achieve this goal, they present a two-layer scheduling architecture, comprised of an application manager and a resource manager. In the application layer, a Scanflow [12] agent determines the wrapping granularity (number of workers and nodes) of the HPC workload based on application characteristics. In the infrastructure layer, an MPI-aware plugin and a task-group scheduling scheme are implemented within a containerized platform scheduler. This approach allows for partitioning each job into a suitable multi-container deployment according to the application profile. The results of their experiments showed that the proposed fine-grained scheduling policies outperform baseline policies, reducing overall response time by 35% and improving makespan by 34%. The authors also demonstrate that their policies provide better usability and flexibility for HPC workloads compared to other frameworks.

Fine-Grained Sub-Second Resource Allocation with Dynamic Telemetry

Escra, introduced by Cusack et al., is a container orchestration system that performs fine-grained, event-based resource allocation across multiple nodes [2] by dynamically adjusting container resources on sub-second

intervals. The Resource Allocator uses windowed statistics of unused runtime and throttles to update per-container limits, as often as every 100ms. The goal is to keep container limits just above usage. Additionally, Escra uses kernel hooks in the memory allocation function to catch containers before they are killed for exceeding their memory limits. If a container exceeds its memory limit, it can request more memory from the controller. The Resource Allocator decides how to allocate additional memory based on node state and application needs. If there is available memory, it can be allocated to the container. If the node is under memory pressure, a memory reclamation process is launched. This approach ensures that resources are right-sized to current demands, rapidly reacting to CPU throttles or out-of-memory (OOM) events, and avoiding performance degradation caused by inaccurate predictions. Escra achieves an average 38% decrease in latency and a 25.4% increase in throughput compared to statically allocated applications while zeroing Out-Of-Memory Killed (OOMKilled) events across all experiments.

While serveral studies have focused on fine-grained performance optimization through container-based resource scheduling algorithms, they often lack features such as autoscaling and rescheduling in order to mitigate interference phenomena. In contrast, the mechanism proposed by Rodriguez et al. integrates scheduling, rescheduling, and autoscaling to achieve a more comprehensive approach to resource management in cloud environments [17].

The core of the mechanism by Rodriguez et al. consists of three key objectives: optimized initial container placement, autoscaling of worker VMs, and container rescheduling. The initial placement of containers is optimized to minimize the number of worker VMs needed while fulfilling the memory and CPU requirements of the containerized applications. This involves efficiently scheduling containers onto the available resources.

The mechanism also includes autoscaling the number of worker VMs at runtime based on the cluster's current workload. This involves both scaling out, which increases the capacity to meet resource demands and reduce container waiting times, and scaling in, which relocates applications to consolidate them and shut down underutilized VMs to reduce infrastructure costs. A rescheduling mechanism further supports the efficient use of resources by consolidating applications onto fewer VMs when possible. This helps to avoid unnecessary scaling out operations and encourages scaling in, thus optimizing overall resource utilization.

The implementation of this mechanism involves Kubernetes scheduler plugins and was evaluated on an Australian national cloud infrastructure. The results of their experiments showed that the proposed approaches can reduce costs by up to 58% compared to the default Kubernetes scheduler. The authors also noted that the binding autoscaler, combined with either a binding or non-binding rescheduler, consistently leads to the lowest costs and, in most cases, the lowest scheduling durations. The study underscores the importance of having heuristics that aim to avoid unnecessary scaling out operations, such as considering instances that are currently being provisioned as potential hosts before provisioning a new node. This integrated approach allows specific resource management policies to be plugged into the system.

IO-Aware Scheduling

While Kubernetes supports CPU and memory affinity enabled scheduling policies, it lacks I/O pressure awareness, introducing bottlenecks in I/O intensive workloads. The Balanced-Disk-IO-Priority (BDI) algorithm and the Balanced-CPU-Disk-IO-Priority (BCDI) algorithm, proposed by Li et al., are two promising solutions that aim to improve I/O balance and reduce response times [10]. BDI focuses on improving disk I/O balance across nodes by dynamically by sensing I/O load using Prometheus. By monitoring the real-time disk I/O of nodes, BDI can make more informed scheduling decisions, thus avoiding I/O bottlenecks. BDI achieved a reduction in the standard deviation of disk I/O on cluster worker nodes, going from 31.884 to 3.949, and reduced node application average response time from 15.14 to 5.88. The BCDI algorithm addresses load imbalances on a single node by balancing both CPU and disk I/O usage, ensuring that no single node is overloaded in terms of either resource type. BCDI achieved a reduction in the sum of the distance between disk I/O and CPU usage of each node from 81.696 to 40.888. Both algorithms act in the Post-Scoring phase to influence scheduling decisions based on real-time resource usage, in contrast to the default Kubernetes scheduler, which relies on static resource requests.

Fine-Grained Scheduling on Serverless Environments

JIAGU by Liu et al., is another approach that focuses on optimizing resource utilization in serverless computing environments by decoupling prediction and decision-making to reduce scheduling latency [13]. JIAGU introduces pre-decision scheduling with a capacity table that stores pre-calculated performance predictions, allowing for fast scheduling decisions. JIAGU also implements dual-staged scaling to efficiently manage resources under load fluctuations, by decoupling resource releasing and instance eviction, thus avoiding cold start overheads. JIAGU achieved a 54.8% improvement in deployment density over commercial clouds (with Kubernetes), 81.0%–93.7% lower scheduling costs, and a 57.4%–69.3% reduction in cold start latency compared to existing QoS-aware schedulers.

Resource Provisioning for Microservices

ChainsFormer, proposed by Song et al., is a framework for microservices that focuses on chain latency-aware resource provisioning [18]. It dynamically scales CPU and memory resources by analyzing microservice inter-dependencies to identify critical chains and nodes. ChainsFormer uses online telemetry data to capture system state, and employs machine learning (ML) and reinforcement learning (RL) models to adapt to system variances, reducing the need for manual intervention. It first identifies the critical chain using a calling graph and a decision tree to find the critical node that has a significant impact on microservice performance. ChainsFormer utilizes RL for efficient decisions regarding vertical and horizontal scaling. Compared to other approaches, ChainsFormer does not require a centralized graph database, which enhances its scalability. ChainsFormer was evaluated on Kubernetes using realistic applications and traces. Experimental results show that ChainsFormer can reduce response time by up to 26% and improve processed requests per second by 8% compared with other techniques.

Distributed-Agent Scheduling

Finally, a hybrid shared-state scheduling framework, proposed by Ungureanu et al., combines distributed scheduling agents with a master agent that synchronizes the cluster state [21]. This approach uses a scheduling correction (SC) function to process unscheduled and unprioritized jobs, addressing issues like collocation interference and priority preemption. This framework employs a lock-free optimistic concurrency to resolve conflicts between service agents (SAs), ensuring optimal scheduling based on the synchronized cluster state and the SC function's processing of task execution time for each pod. The hybrid shared-state scheduler addresses limitations in other Kubernetes schedulers, such as issues with node over/under-utilization and node failures, and offers better support for inter-pod affinity/anti-affinity, taints/tolerations, baseline scheduling, and priority preemption.

These custom scheduling mechanisms underscore the need for more advanced schedulers that go beyond the default Kubernetes capabilities. These solutions can dynamically adapt to changing conditions and optimize resource allocation for better performance and cost-effectiveness.

Chapter 4

The Kubernetes Container Orchestrator

4.1 Container Orchestration

A Container Orchestrator automatically provisions, deploys, scales, and manages containerized applications within a cluster of worker nodes. It offers a declarative API, typically through manifests (configuration files), enabling cluster administrators to define the desired state of the cluster. The orchestrator then automatically executes the necessary actions to achieve and maintain this state.

With the rise of containerized microservices in software development, cloud-based applications have greatly benefited from Kubernetes, that has become the de facto industry-standard when it comes to container orchestration. Because of its ability to run containers, it gives the end-user the ability to deploy diverse workloads, including large-scale app deployments, high-performance computing, machine learning, and Big Data workloads.

In this section, we will provide a detailed overview of the key components of a Kubernetes cluster and the default resource management policies it offers. Furthermore, we will explore numerous extension points for developers the platform provides, which we will utilize to create our custom scheduler and resource allocator.

4.2 Cluster Architecture

A Kubernetes cluster is composed of a control plane and a set of worker machines, known as nodes, which run containerized applications. Each cluster requires at least one worker node to host and execute Pods, the fundamental units of application workloads. The control plane oversees the worker nodes and manages the Pods within the cluster. In production environments, the control plane typically spans multiple machines, and clusters often consist of several nodes to ensure fault tolerance and high availability.

4.2.1 Control Plane Components

The Control Plane consists of one or more nodes and is responsible for managing the overall state of the cluster. The main source of truth of what resources (e.g. Deployment, DaemonSet, Job) are applied in the cluster is the API Server, where the cluster-wide state management takes place. The API Server provides a RESTful API that enables the cluster administrator to manage the deployed resources, either by interacting directly with the API, using a CLI tool like `kubectl`, or by applying configuration manifest files.

`kube-apiserver`

The API Server is the primary component of a Kubernetes cluster. It exposes a RESTful API that enables administrators to query, configure, and manage Kubernetes objects (resources). The API Server is the source of truth of all the resources that have been applied within the cluster. It also serves as the communication hub for other critical components, such as the scheduler and kubelet, which monitor events published by the API Server and take action accordingly.

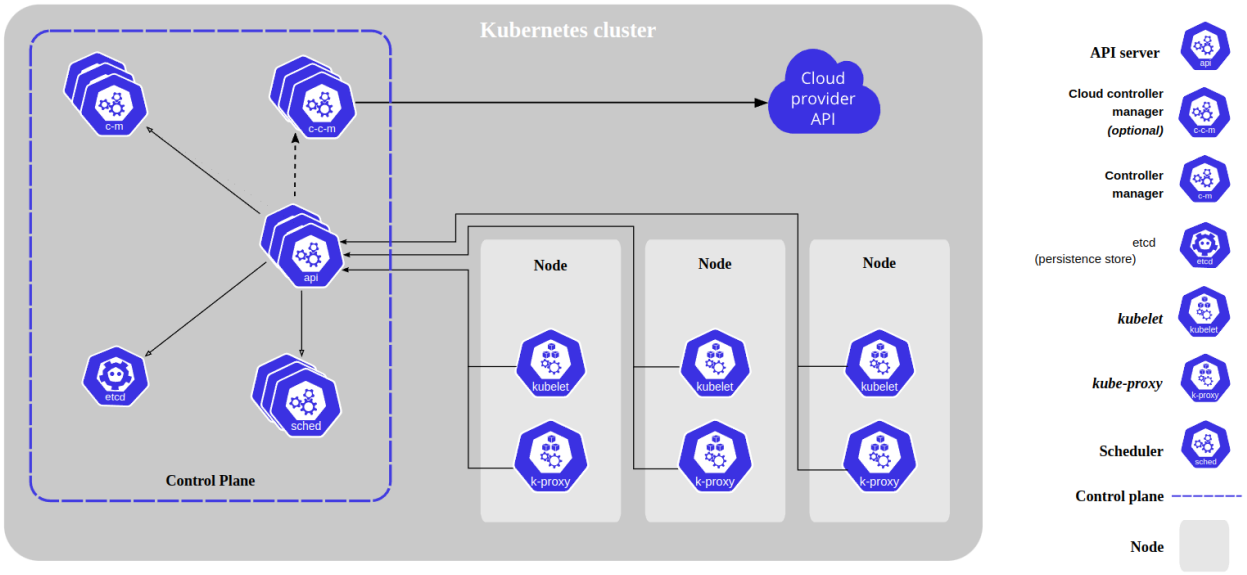


Figure 4.2.1: Kubernetes Components

`etcd`

`etcd` is an open-source, distributed key-value database and the standard storage system used in Kubernetes. It is responsible for storing not only metadata about the Kubernetes objects such as Pods, Services, and Deployments, but also the actual cluster state and configuration.

`kube-scheduler`

The Kubernetes scheduler is a control plane process which assigns Pods to Nodes. The scheduler determines which Nodes are valid placements for each Pod in the scheduling queue according to constraints and available resources. The scheduler then ranks each valid Node and binds the Pod to a suitable Node. Multiple schedulers may be used within a cluster and `kube-scheduler` is the default implementation.

`kube-controller-manager`

The Controller Manager (also called `kube-controller-manager`) is a control plane process that acts as a continuous control loop in a Kubernetes cluster. The controller monitors the current state of the cluster via event streams from API Server, and works to reconcile it with the desired state defined by the cluster administrator through manifests.

The Controller Manager packages controllers that correspond to the default API resources. Each controller (e.g. the DaemonSet controller that manages indentical Pods running as daemons on each worker) utilizes the Operator Pattern to reconcile the objects that are applied on the API server (the DaemonSet manifests applied by the administrator) and bring the cluster to the desired state. These controllers use Informers, a programming interface that allows an application to retrieve objects from the API Server and listen to create/update/delete events, enabling them to take action upon these events.

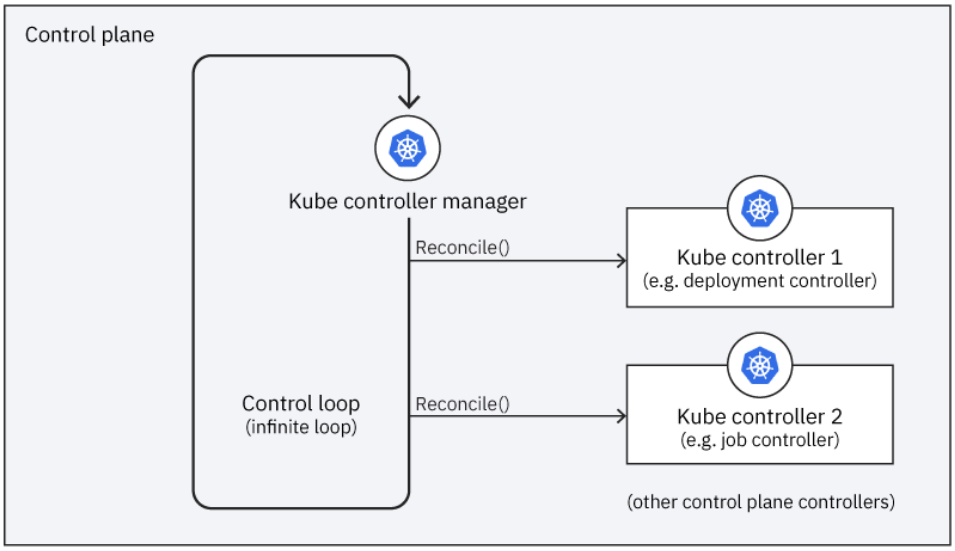


Figure 4.2.2: The Controller Manager and the Operator Pattern

Understanding how the default Controller Manager of Kubernetes works is key as on a later chapter we will introduce our custom Controller Manager that reacts on events occuring on our own Custom Resource Definitions (CRDs). The Controller Manager we developed is an important component of our custom resource allocator, as we will discover later.

4.2.2 Node Components

`kubelet`

The kubelet is the primary node agent that runs on each node. It can register the node with the API server, and ensures that containers scheduled to its Node are running and healthy.

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 are provided through various mechanisms (primarily through the API server) and manages their execution and lifecycle. The kubelet doesn't manage containers which were not created by Kubernetes.

`kube-proxy`

The `kube-proxy` of each node is a network proxy for facilitating Kubernetes networking services. The kube-proxy handles network communications inside or outside of the cluster, relying either on the operating system's packet filtering layer, or forwarding the traffic itself.

4.3 Workloads

Pod

Pods are the smallest deployable units of computing in Kubernetes. A Pod is a group of one or more containers, with shared storage and network resources, and a specification for how to run the containers. A Pod's contents are always co-located and co-scheduled, and run in a shared, isolated context. The isolation is achieved using Linux cgroups, namespaces, and potentially other facets of isolation - the same things that isolate a container.

All containers within a Pod are co-located and co-scheduled, operating in a shared context, with common resources and dependencies. Pods can include init containers that run during startup to set up necessary

conditions before the main application containers begin execution.

Generally, Kubernetes manages Pods through higher-level workload resources like Deployments and StatefulSets, which handle Pod replication, scaling, and lifecycle management, ensuring applications are resilient and scalable.

```
apiVersion: v1
kind: Pod
metadata:
  name: example-pod
  labels:
    app: demo
spec:
  containers:
  - name: demo-container
    image: nginx:1.25.2
    ports:
    - containerPort: 80
    resources:
      requests:
        memory: "64Mi"
        cpu: "250m"
      limits:
        memory: "128Mi"
        cpu: "500m"
```

Listing 4.1: Pod Manifest Example

Deployment

A Deployment is a higher-level resource that manages the lifecycle of applications by overseeing ReplicaSets and the Pods within them. It enables declarative updates, allowing users to define the desired state of an application, including the number of replicas, the container image version, and update strategies. The Deployment controller ensures that the actual state matches the desired state by creating or removing Pods as needed, facilitating seamless rollouts and rollbacks of application versions. This mechanism provides robust scaling, self-healing capabilities, and efficient management of stateless applications, ensuring high availability and reliability in dynamic environments.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: example-deployment
  labels:
    app: demo
spec:
  replicas: 3
  selector:
    matchLabels:
      app: demo
  template:
    metadata:
      labels:
        app: demo
    spec:
      containers:
      - name: demo-container
        image: nginx:1.25.2
        ports:
        - containerPort: 80

```

Listing 4.2: Deployment Manifest Example

ReplicaSet, StatefulSet, DaemonSet

In Kubernetes, ReplicaSets, StatefulSets, and DaemonSets are controllers designed to manage specific workload requirements. A ReplicaSet ensures a defined number of identical, stateless Pods are running, automat-

ically creating or deleting Pods to match the desired count. It is often used indirectly through Deployments, which add update and rollback capabilities. In contrast, a StatefulSet manages stateful applications requiring unique identities, persistent storage, and ordered deployment or scaling—ideal for databases and distributed systems. Lastly, a DaemonSet ensures a copy of a Pod runs on every node (or selected nodes) in the cluster, making it suitable for tasks like logging, monitoring, or networking services that need node-level presence. These controllers provide scalability, persistence, and reliability, addressing diverse application needs in distributed environments.

```yaml
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: fluentd-elasticsearch
  namespace: kube-system
  labels:
    k8s-app: fluentd-logging
spec:
  selector:
    matchLabels:
      name: fluentd-elasticsearch
  template:
    metadata:
      labels:
        name: fluentd-elasticsearch
    spec:
      tolerations:
      - key: node-role.kubernetes.io/control-plane
        operator: Exists
        effect: NoSchedule
      containers:
      - name: fluentd-elasticsearch
        image: quay.io/fluentd_elasticsearch/fluentd:v2.5.2
        resources:
          limits:
            memory: 200Mi
          requests:
            cpu: 100m
            memory: 200Mi
        volumeMounts:
        - name: varlog
          mountPath: /var/log
      volumes:
      - name: varlog
        hostPath:
          path: /var/log

```

Listing 4.3: DaemonSet Manifest Example

In a later chapter, we will discover how our custom resource allocation mechanism utilizes the Deployment and DaemonSet workloads to create a distributed communcation between the Controller Manager of our CRDs and the daemon that's running inside each worker node in order to pin and isolate the Pods on specific CPU resources.

4.4 Services, Load Balancing, and Networking

Kubernetes provides robust networking and service management through Services, which abstract access to a group of Pods, ensuring stable communication despite dynamic Pod lifecycles. Services support types like ClusterIP for internal access, NodePort for exposing services externally via node IPs, and LoadBalancer for integrating external load balancers. Kubernetes assigns each Pod a unique IP, enabling direct communication without Network Address Translation (NAT). Additionally, Ingress manages external HTTP and HTTPS traffic, offering load balancing and SSL termination, enabling scalable and resilient application networking.

ClusterIP, NodePort, LoadBalancer, ExternalName

There are several types of Services, each serving a specific purpose:

- **ClusterIP**: This is the default Service type, providing a stable internal IP address accessible only within the Kubernetes cluster. It facilitates internal communication between services without exposing them externally.
- **NodePort**: This Service type exposes the Service on a static port on each node's IP address, allowing external traffic to access the Service. While it provides external access, it is generally used for development or testing due to limited security and scalability.
- **LoadBalancer**: This Service type integrates with cloud providers to provision an external load balancer that distributes incoming traffic across the Pods. It is commonly used for production workloads requiring reliable external access.
- **ExternalName**: This Service type maps a Service to a DNS name, allowing Kubernetes to return a CNAME record with the external name. It is useful for integrating external services into a Kubernetes cluster without complex configurations.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: clusterip-service
spec:
  type: ClusterIP
  selector:
    app: my-app
  ports:
  - port: 80
    targetPort: 8080
```

Listing 4.4: ClusterIP

```yaml
apiVersion: v1
kind: Service
metadata:
  name: nodeport-service
spec:
  type: NodePort
  selector:
    app: my-app
  ports:
  - port: 80
    targetPort: 8080
    nodePort: 30080
```

Listing 4.5: NodePort

```yaml
apiVersion: v1
kind: Service
metadata:
  name: loadbalancer-service
spec:
  type: LoadBalancer
  selector:
    app: my-app
  ports:
  - port: 80
    targetPort: 8080
```

Listing 4.6: LoadBalancer

```yaml
apiVersion: v1
kind: Service
metadata:
  name: externalname-service
spec:
  type: ExternalName
  externalName: example.com
```

Listing 4.7: ExternalName

4.5 Storage

By default, container storage is ephemeral, meaning that data is lost when a container restarts or terminates. Volumes address this limitation by offering a shared storage space accessible to all containers in a Pod, ensuring data persistence across container restarts and facilitating inter-container communication through shared files.

Persistent Volumes (PVs) are storage resources provisioned independently of Pods, offering a decoupled and persistent storage solution within the cluster. Administrators can set up PVs manually or enable dynamic provisioning through StorageClasses, which automate the creation of storage based on predefined parameters. This abstraction allows users to request storage without needing to understand the underlying infrastructure, promoting a clear separation between storage provisioning and consumption.

Users interact with PVs by creating PersistentVolumeClaims (PVCs), which specify the desired storage capacity and access modes. The Kubernetes control plane then matches these claims to available PVs, binding them accordingly. This mechanism ensures that storage resources are efficiently allocated and managed, providing a consistent and reliable storage experience for applications running within the cluster.

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: nfs-csi
provisioner: nfs.csi.k8s.io
parameters:
  server: nfs
  share: /kubecluster
reclaimPolicy: Delete
volumeBindingMode: Immediate
mountOptions:
  - nfsvers=4.1
```

Listing 4.8: NFS StorageClass

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: benchmarks-logs-pvc
  namespace: benchmarks
spec:
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 5Gi
  storageClassName: nfs-csi
```

Listing 4.9: PersistentVolumeClaim

In listings 4.8 and 4.9 how Kubernetes dynamically provisions storage using a StorageClass and a PersistentVolumeClaim (PVC). The StorageClass defines the storage backend (NFS in this case) and parameters for provisioning, while the PVC requests storage with specific requirements, such as size and access mode. Once the PVC is bound to a provisioned volume, it can be mounted by a Pod as a persistent storage location, ensuring data durability across container restarts.

4.6 Resource Management

When defining a Pod, users can optionally specify the resource requirements for each container within it (figure 4.1). The most commonly specified resources are CPU and memory (RAM), although other resources can also be defined based on the third-party devices that are installed in the cluster.

Resource requests and limits play a crucial role in efficient resource allocation and enforcement. Resource requests indicate the minimum amount of a resource that a container requires. The `kube-scheduler` uses this information to determine which node has sufficient available resources to host the Pod. In contrast, resource limits define the maximum amount of a resource that a container is allowed to consume. These limits are enforced by the kubelet, which prevents a container from exceeding its specified allocation. Additionally, the kubelet guarantees that the requested resources are reserved exclusively for the container, ensuring predictable performance and minimizing resource contention.

Resource Types

CPU

CPU is a measurable resource that represents the computational power required by a container. CPU resources are specified in millicores, where 1000m equals one logical core. Containers can request a specific amount of CPU, ensuring that the Kubernetes scheduler places them on nodes with sufficient capacity. Additionally, limits can be set to cap CPU usage, preventing any single container from monopolizing processing

power. When a container exceeds its allocated CPU limit, it is throttled rather than terminated, and it is subject to under-performing.

Memory

Memory, measured in bytes, represents the working set required by a container to operate effectively. Unlike CPU, memory usage cannot be throttled; therefore, exceeding a container's memory limit results in its termination and restart. Kubernetes allows specifying both requests and limits for memory, ensuring predictable allocation and preventing resource contention. Requests guarantee a minimum memory allocation, while limits define the upper boundary, enabling efficient utilization of cluster resources without risking out-of-memory errors.

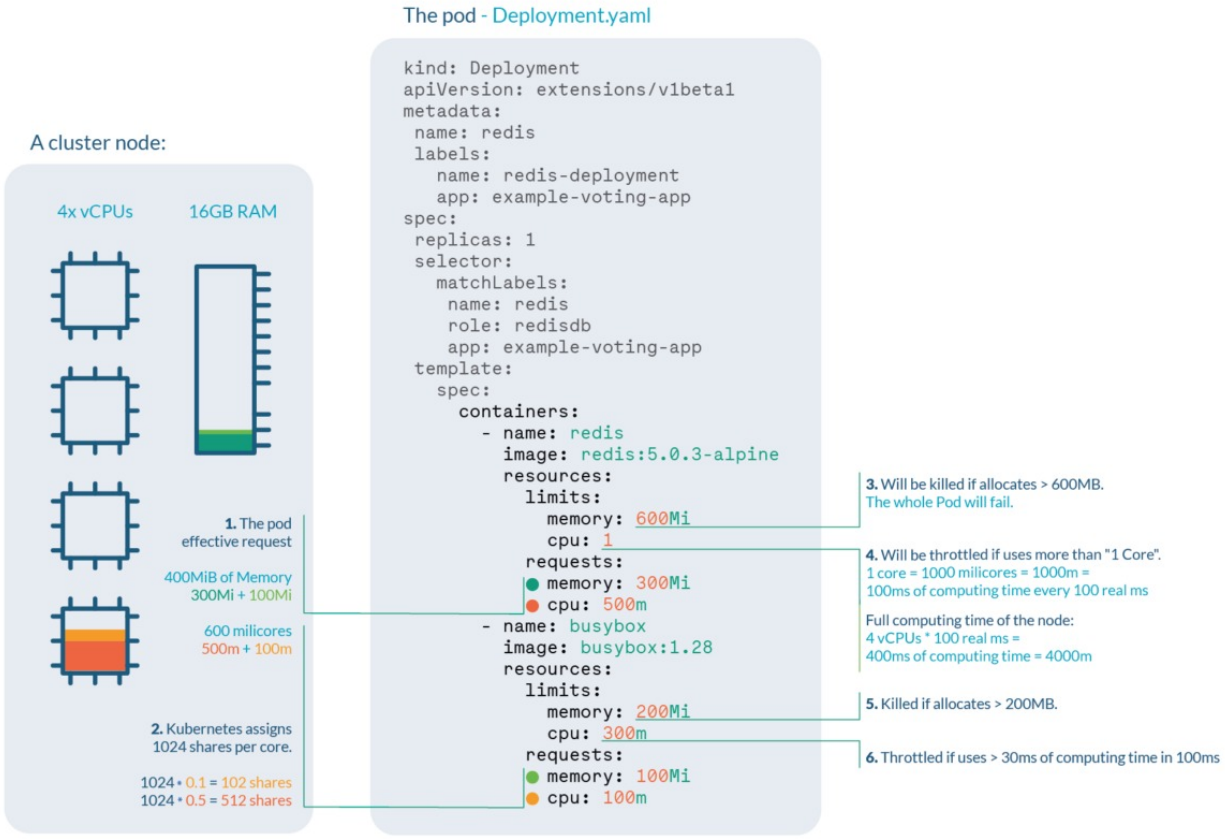


Figure 4.6.1: Resource Requests and Limits of a Pod

Pod Quality of Service - QoS

Quality of Service (QoS) classes are assigned to Pods based on their specified resource requests and limits. These classes guide the system in resource allocation and eviction decisions, especially during resource constraints. Kubernetes defines three QoS classes to manage resource prioritization and ensure application reliability within the cluster.

- **Guaranteed**: A Pod is classified as Guaranteed if every container within it has both memory and CPU limits and requests set, with each limit equal to its corresponding request. This ensures that the Pod receives the highest priority, making it the least likely to be evicted during resource shortages.
- **Burstable**: Pods that do not meet the Guaranteed criteria but have at least one container with a memory or CPU request fall under the Burstable class. These Pods have reserved resources but can also utilize excess capacity when available, offering a balance between resource assurance and flexibility.

- **BestEffort**: Pods where no containers have specified memory or CPU requests or limits are classified as BestEffort. These Pods are the first to be considered for eviction under resource constraints, as they have no guaranteed resource reservations.

Pod Scheduling and Eviction

Pod scheduling is the process of assigning newly created Pods to suitable Nodes within the cluster. The kube-scheduler, Kubernetes' default scheduler, evaluates each Pod's resource requirements, constraints, and policies to determine the optimal Node for deployment. Pod eviction refers to the removal of Pods from Nodes, which can occur due to various factors, including resource contention, Node maintenance, or policy enforcement. Kubernetes prioritizes Pods for eviction based on their assigned Quality of Service (QoS) classes and Pod Priority. Pods with lower priority or BestEffort QoS are more susceptible to eviction under resource pressure, ensuring that critical applications maintain their required performance levels.

Kubelet Resource Management

The kubelet of each node ensures that containers have the necessary resources to operate effectively while enforcing the resource constraints declared in the Pod's specification. It utilizes a Linux kernel feature named `cgroups` which applies restrictions on running processes, for resources such as memory, CPU, block I/O and huge pages.

The kubelet mainly modifies the following `cgroups` controllers for the managed Pods:

- **Memory Controller (`memory`):** Manages memory usage by setting limits (`memory.max`, `memory.min`) and enforcing guarantees. If the container consumes all of its entitled memory, the Pod terminates with an OOMKilled (Out-Of-Memory Killed) error.
- **CPU Controller (`cpu`):** Applies CPU limits for processes requesting CPU resources. `cpu.weight` (or `cpu.shares`) describes how much CPU time will the container get in comparison to other containers. Additionally, `cpu.max` (a combination of the older `cpu.cfs_quota_us` and `cpu.cfs_period_us`) applies a hard limit on how much computing time a process is entitled within a CFS period. The container's performance might be degraded (throttled) due to resource contention. When Static CPU Manager Policy is enabled, the kubelet also allocates specific CPUs to Guaranteed Pods, by setting the `cpuset.cpus` of the `cpuset` controller [1].
- **PIDs Controller (`pids`):** Restricts the number of processes using `pids.max` to prevent resource exhaustion caused by runaway processes.

Furthermore, the container runtime utilizes another Linux kernel feature called `namespaces`. Namespaces are a feature of the Linux kernel that partitions kernel resources such that one set of processes sees one set of resources while another set of processes sees a different set of resources. Containers are isolated into their set of namespaces, such as user, PID, network, mount and IPC namespaces ensuring they are running independently of the rest of the system.

Kubernetes employs a suite of resource managers within the kubelet to optimize node-level resource allocation, particularly for workloads with specific requirements for CPUs, devices, and memory. The two most important resouce management modules of the kubelet are the CPU Manager and the Topology Manager.

CPU Manager

The CPU Manager is a component of the Kubelet responsible for managing CPU resource allocation to containers within a node. It offers two primary policies: the default `none` policy, which does not perform any special CPU pinning, and the `static` policy, which provides enhanced CPU affinity and exclusivity for certain workloads.

Static Policy Options The static policy can be fine-tuned using several options to serve to specific workload requirements:

[1] https://martinheinz.dev/blog/91

- `full-pcpus-only`: Allocates only full physical cores, avoiding sharing between containers and reducing the noisy neighbors problem. Pods failing to meet this requirement are marked as `Failed` with an `SMTAlignmentError`.
- `distribute-cpus-across-numa`: Ensures CPUs are evenly distributed across NUMA nodes when more than one node is required, minimizing bottlenecks and improving parallel performance for workloads relying on synchronization primitives.
- `align-by-socket`: Allocates CPUs aligned at the socket boundary instead of the NUMA boundary, reducing performance degradation caused by cross-socket resource allocation. This option is incompatible with the Topology Manager's `single-numa-node` policy.
- `distribute-cpus-across-cores`: Spreads virtual cores (hardware threads) across multiple physical cores to reduce contention and improve performance for workloads sensitive to CPU sharing. This may be less effective under high system loads.
- `prefer-align-cpus-by-uncorecache`: Allocates CPUs within the same uncore cache block (LLC) to optimize cache usage and reduce inter-cache latency, improving performance for workloads sensitive to cache contention.

While providing fine-grained control over CPU allocation through its static policy and configurable flags, the CPU Manager has several limitations. Firstly, its configuration cannot be adjusted on the fly, requiring a kubelet restart to apply changes, which can disrupt running workloads. This prevents dynamic adaptation to varying workload requirements. Additionally, the CPU Manager does not support multiple allocation policies simultaneously, making it unsuitable for clusters running heterogeneous workloads that might benefit from different strategies, such as NUMA-aware distribution for one application and core isolation for another. Furthermore, the CPU Manager lacks the ability to dynamically adjust resource allocations for containers after they are scheduled, leaving no mechanism to mitigate performance degradation on runtime. These limitations make the CPU Manager less flexible for heterogeneous workloads with evolving resource needs, motivating us to develop a more flexible resource allocator that solves the described limitations.

Topology Manager

The Topology Manager in Kubernetes is a component of the kubelet that attempts to achieve optimal resource allocation by aligning CPU, memory, and device assignments with the underlying hardware topology, particularly in systems with NUMA architectures. By coordinating resource allocation, it enhances application performance and reduces latency for workloads sensitive to resource locality. The Topology Manager supports various policies, such as `none`, `best-effort`, `restricted`, and `single-numa-node`, which determine the strictness of resource alignment.

The Topology Manager coordinates with the Memory Manager to ensure resource alignment across NUMA nodes. It retrieves topology hints, such as "10" for a single NUMA node or "11" for a multi-NUMA group, based on available memory. After finalizing hints, memory is pre-allocated, and cgroups are updated via the CRI API to enforce resource assignments efficiently.

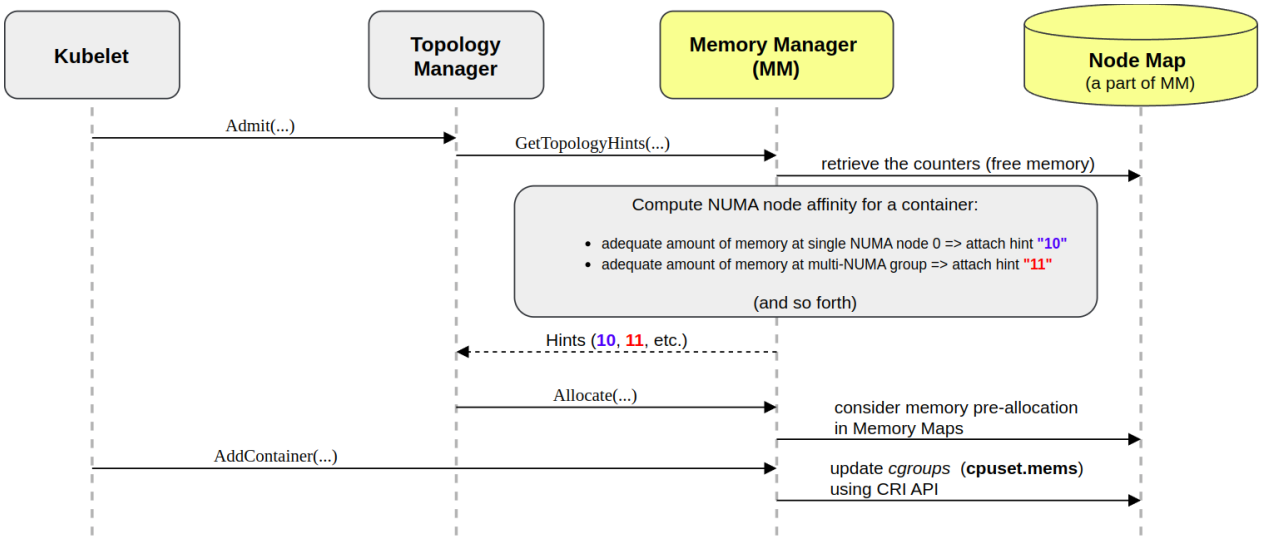


Figure 4.6.2: Kubelet's Memory Manager Workflow

4.7 The Kubernetes Scheduler

The Kubernetes Scheduler runs as part of the control plane and watches for newly created Pods that have no node assigned. By filtering the Nodes, the scheduler comes up with a set of feasible nodes that meet the Pod's requirements. The Pod's containers can have different resource requirements, and the scheduler is responsible for selecting the optimal node to run it. If none of the nodes are suitable, the Pod remains unscheduled until the scheduler is able to place it.

The scheduler finds feasible nodes for a Pod and then runs a set of functions to score the feasible nodes and picks a node with the highest score among the feasible ones to run the Pod. The scheduler then notifies the API server about this decision in a process called binding.

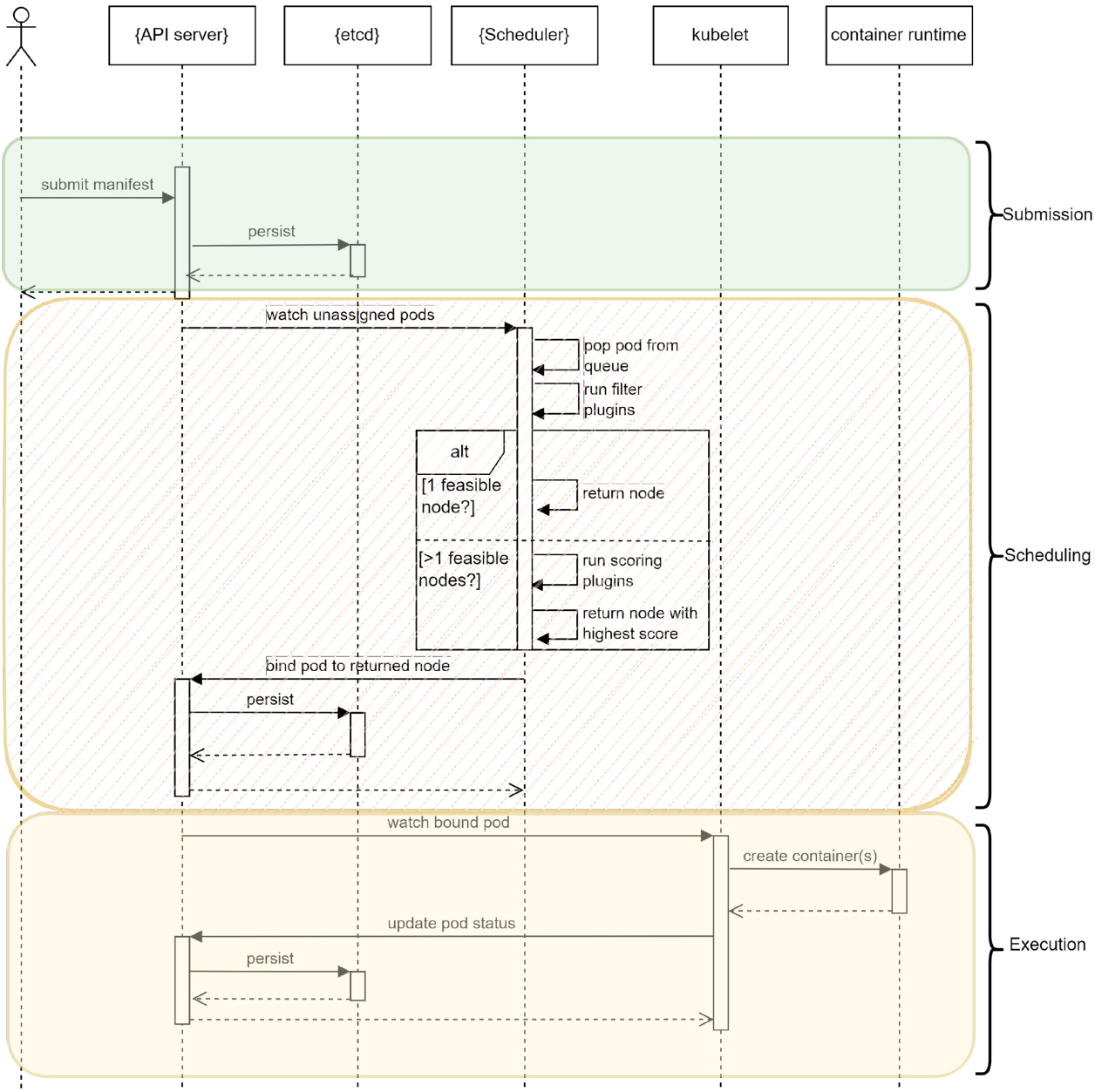


Figure 4.7.1: Lifecycle of a Pod

Factors that need to be taken into account for scheduling decisions include individual and collective resource requirements, hardware / software / policy constraints, affinity and anti-affinity specifications, data locality, inter-workload interference, among others.

1. **QueueSort**: This stage defines an ordering function to sort pending Pods in the scheduling queue. Only one queue sort mechanism can be enabled at a time.
2. **PreFilter**: This stage pre-processes or validates information about a Pod or the cluster before filtering. It can also mark a Pod as unschedulable.
3. **Filter**: Equivalent to predicates in a scheduling policy, this stage filters out nodes that cannot run the Pod. Filters execute sequentially, and a Pod is marked unschedulable if no nodes satisfy all filters.
4. **PostFilter**: Invoked when no feasible nodes are found, this stage can make Pods schedulable by modifying the scheduling decision. If any part of this stage marks a Pod as schedulable, remaining

steps are skipped.

5. **PreScore**: An informational stage used to prepare data or perform pre-scoring tasks before scoring nodes.
6. **Score**: This stage assigns scores to nodes that pass the filtering phase. The node with the highest weighted score is selected.
7. **Reserve**: This stage notifies the system when resources are reserved for a Pod. It includes an Unreserve step, triggered in case of failure during or after reservation.
8. **Permit**: This stage can either delay or prevent a Pod from being bound to a node.
9. **PreBind**: This stage executes any required preparation work before binding a Pod to a node.
10. **Bind**: Responsible for binding a Pod to a node, this stage executes sequentially. Once a binding is complete, remaining steps are skipped. At least one binding mechanism is mandatory.
11. **PostBind**: An informational stage invoked after a Pod has been successfully bound.

The `kube-scheduler` is a plugin-based application, meaning that it bundles serveral scheduling plugins that implement one or more of the above extension points. The `kube-scheduler` utilizes a range of default plugins to handle scheduling decisions, each designed to address specific requirements. For instance, plugins like `ImageLocality` and `NodeResourcesFit` prioritize nodes based on image availability and resource requirements, while `TaintToleration` and `NodeAffinity` enforce constraints related to taints, tolerations, and node affinity rules. Others, such as `PodTopologySpread` and `InterPodAffinity`, ensure even workload distribution and affinity/anti-affinity compliance. Plugins like `VolumeBinding` and `NodeVolumeLimits` handle storage constraints, and `PrioritySort` and `DefaultPreemption` define sorting and preemption mechanisms. These plugins are enabled on the initialization of any cluster and they encapsulate the scheduling policies that are applied by default.

To configure the different stages of scheduling, the user may pass a scheduler configuration manifest as a command line argument to the scheduler. The `KubeSchedulerConfiguration` manifest contains information about scheduling profiles, including enabled plugins, plugin configurations, and arguments to customize the scheduling behavior.

For example, in the scheduler configuration below, we have defined two scheduling profiles, the `default-scheduler` which has all the default plugins enabled, and the `no-scoring-scheduler`, which disabled all plugins that implement the `Score` extension point.

```
apiVersion: kubescheduler.config.k8s.io/v1
kind: KubeSchedulerConfiguration
profiles:
  - schedulerName: default-scheduler
  - schedulerName: no-scoring-scheduler
    plugins:
      preScore:
        disabled:
        - name: '*'
      score:
        disabled:
        - name: '*'
```

Listing 4.10: `KubeSchedulerConfiguration` Example Manifest

Kubernetes enables developers to create their own scheduler plugins by utilizing the Scheduling Framework. We will describe in depth how our scheduler plugin works in a later chapter.

4.8 Kubernetes Interface Standards

Kubernetes uses several interface standards to ensure flexibility and compatibility with different systems. These standards allow Kubernetes to work with a variety of tools for containers, networking, and storage without requiring major changes to its core components.

The **Container Runtime Interface (CRI)** provides a way for Kubernetes to communicate with different container runtimes. This means Kubernetes can manage containers using different technologies, as long as they follow the CRI standard. Examples of CRI implementations include `containerd` and `CRI-O`. This flexibility allows Kubernetes to support emerging container technologies without needing to rewrite its code.

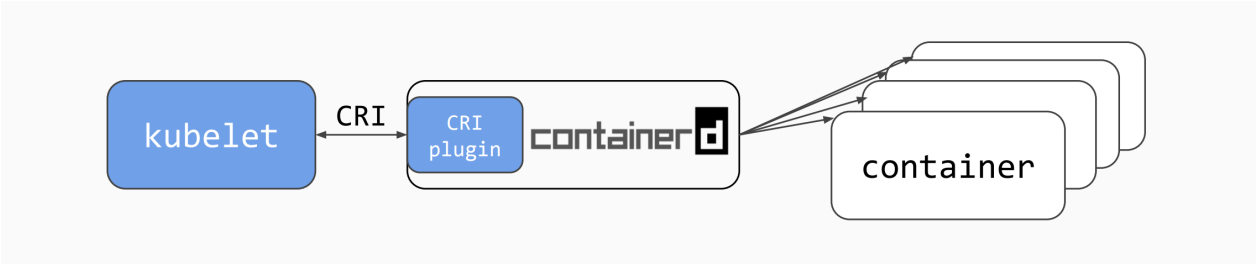


Figure 4.8.1: The CRI specification defines the interface between the kubelet and the container runtime

For networking, Kubernetes uses the **Container Network Interface (CNI)**. The CNI standard allows administrators to use various networking solutions to connect pods and manage network policies. Popular CNI implementations include Calico, Flannel, and Cilium. These plugins help provide networking features such as IP address management, routing, and security policies.

In terms of storage, Kubernetes integrates the **Container Storage Interface (CSI)** to manage storage systems. The CSI standard lets Kubernetes work with different storage providers without requiring changes to its core code. Examples of CSI drivers include Amazon AWS EBS CSI Driver, Google Compute Engine Persistent Disk CSI Driver, and Ceph CSI. This makes it easier to use various storage backends, whether they are cloud-based or on-premises, while maintaining compatibility with Kubernetes.

In our setup, we use the `csi-driver-nfs` plugin to persist data on an NFS server.

4.9 Frameworks for Developers

While Kubernetes offers a wide range of features out of the box, cluster operators might need some more advanced, hardware-specific functionality to integrate with their system. Kubernetes is designed to be highly customizable by providing development tools for interacting and extending the API server, creating custom scheduling policies, installing and advertisting custom hardware resources such as GPUs and FPGAs, and placing custom CSI and CNI plugin implementations. In this section we will analyze in depth the most significant extension points for developers and the ones we utilized to create our custom resource allocation mechanism.

Custom Resource Definitions - CRDs

Custom Resource Definitions (CRDs) enable Kubernetes users to define and manage their own resources, extending the Kubernetes API to support domain-specific objects. By creating a CRD, developers can introduce new resource types that behave like native Kubernetes objects. Once registered, these custom resources can be managed using familiar Kubernetes tools, such as `kubectl` and YAML manifests. That way, organizations are able to define abstractions tailored to their workloads and manage them in a declarative manner, just as they would manage their Deployments, StatefulSet, and so on.

```
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  name: crontabs.stable.example.com
spec:
  group: stable.example.com
  versions:
    - name: v1
      served: true
      storage: true
```

```
    schema:
      openAPIV3Schema:
        type: object
        properties:
          spec:
            type: object
            properties:
              cronSpec:
                type: string
              image:
                type: string
              replicas:
                type: integer
  scope: Namespaced
  names:
    plural: crontabs
    singular: crontab
    kind: CronTab
    shortNames:
    - ct
```

Listing 4.11: CustomResourceDefinition Example

The CustomResourceDefinition described in 4.11 will create a new RESTful endpoint on the API server of the cluster under `/apis/stable.example.com/v1/namespaces/*/crontabs/...`. This allows cluster administrators to create and manage custom resources of the specified schema. For example, the following manifest defines a custom resource that matches the above definition:

```
apiVersion: "stable.example.com/v1"
kind: CronTab
metadata:
  name: my-new-cron-object
spec:
  cronSpec: "* * * * */5"
  image: my-cron-image
```

Listing 4.12: Custom Resource Manifest Example

Operator Pattern and Frameworks

Unlike the resources available in the API server by default, custom resources that are installed through CRDs do not have a corresponding controller that reconciles the cluster state with the applied manifests. Kubernetes' operator pattern concept lets you extend the cluster's behaviour without modifying the code of Kubernetes itself by linking controllers to one or more custom resources. Operators are clients of the Kubernetes API that act as controllers for a Custom Resource. Custom controllers follow the Operator Design Pattern, listening on create/update/delete events for the custom resources of the API server and taking the required actions that brings the cluster's state to the desired one.

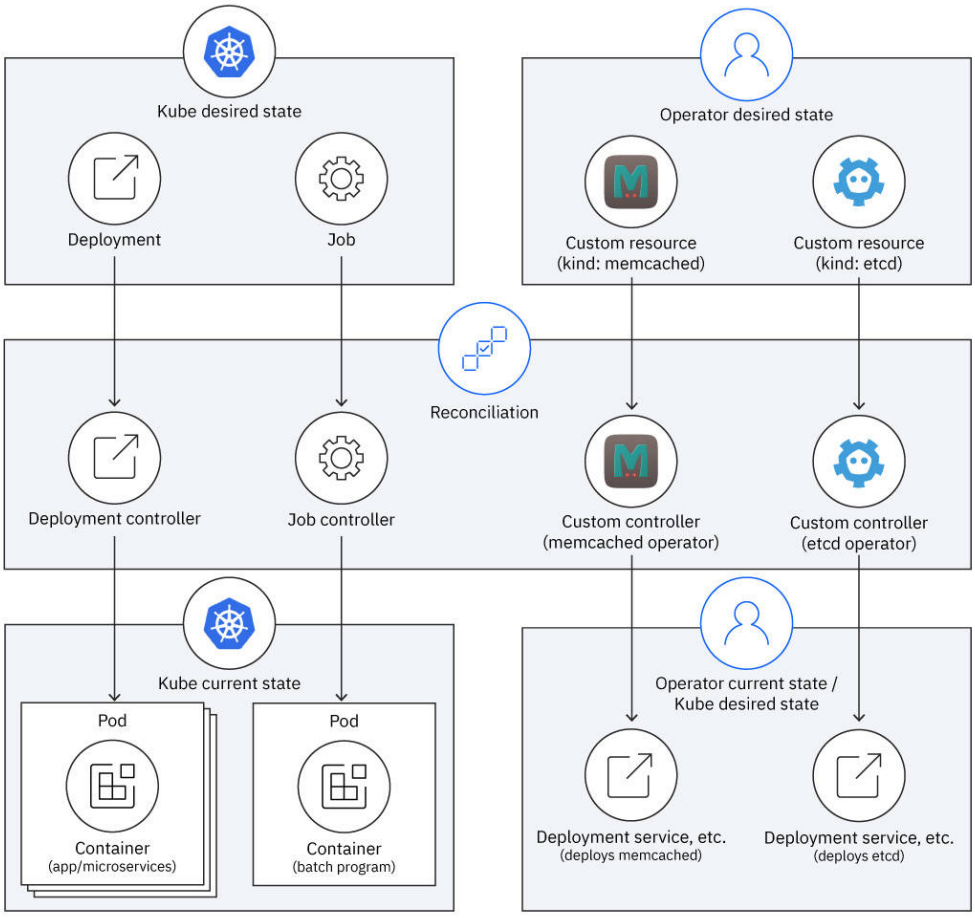


Figure 4.9.1: Custom Operators Within a Kubernetes Cluster

Kubebuilder is a framework developed by the Kubernetes Special Interest Group (SIG) for building Kubernetes APIs and controllers. It provides scaffolding, code generation, and integration with Kubernetes conventions. Kubebuilder supports advanced features such as webhook validation, admission controllers, API versioning, and custom reconciliation loops, enabling the development of scalable and production-ready extensions. It is built on top of the Kubernetes `controller-runtime` library, ensuring compatibility with native Kubernetes APIs. In this thesis, Kubebuilder was used to implement the controller manager, the application that manages the creation and lifecycle of our custom resources, automating the process of allocating specific resources to our Pods.

Scheduling Framework

The scheduler of Kubernetes is implemented with a pluggable architecture and consists of a set of "plugin" APIs that are compiled directly into the scheduler, as described before. The framework enables developers to follow the same plugin-based architecture and build their own scheduler flavors with custom scheduler plugins that meet their specific needs.

The Scheduling Framework defines several extension points, both in the scheduling cycle (stages that correspond to the decision making of the scheduler), as well as the binding cycle (stages related to the binding of resources to the Pod). Together, a scheduling cycle and binding cycle are referred to as a "scheduling context".

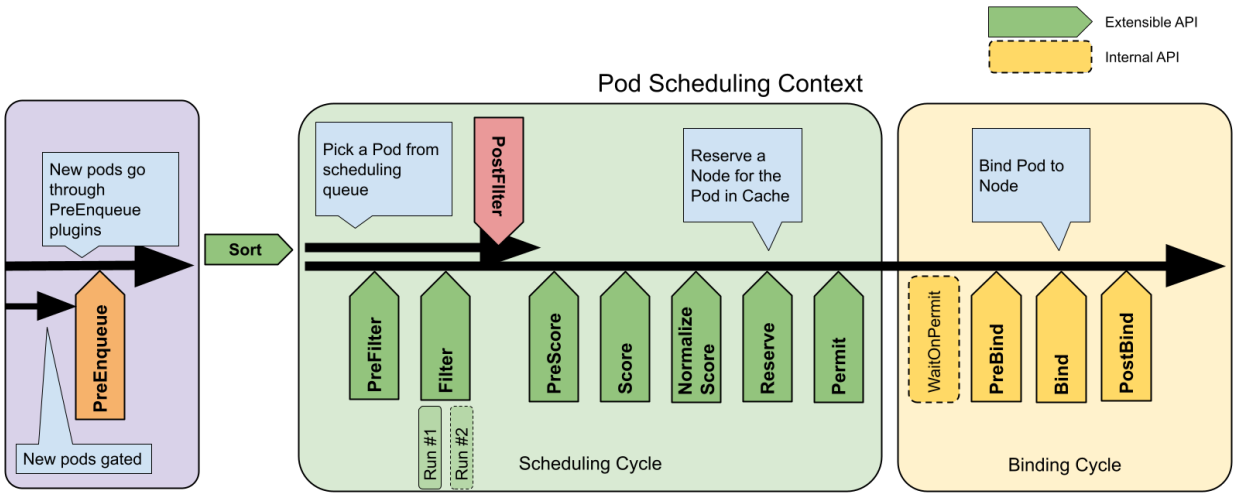


Figure 4.9.2: Scheduling Framework Extension Points

The scheduling workflow comprises of different extension points (stages) that run in the following order:

- **QueueSort**: The plugin sorts Pods in the scheduling queue by providing a function to determine the processing order. Only one QueueSort plugin can be enabled at a time.
- **PreFilter**: The plugin pre-processes Pod information and verifies cluster or Pod requirements before proceeding. An error returned by this plugin aborts the scheduling cycle.
- **Filter**: The plugin filters out nodes that do not meet the Pod's requirements. Nodes are evaluated in sequence, and once marked infeasible, subsequent plugins are skipped for that node.
- **PostFilter**: The plugin is executed when no feasible nodes are found, and it attempts to make the Pod schedulable by actions such as preempting lower-priority Pods.
- **PreScore**: The plugin prepares data needed by Score plugins to rank nodes. Errors in this phase abort the scheduling cycle.
- **Score**: The plugin ranks feasible nodes based on predefined metrics and scoring rules. Node scores are combined and weighted to determine the final placement.
- **NormalizeScore**: The plugin adjusts raw scores generated by Score plugins to ensure uniform scaling and ranking. Errors during this phase abort the scheduling cycle.
- **Reserve**: The plugin temporarily reserves resources on a selected node to avoid conflicts. It also handles cleanup in case of failure through the Unreserve phase.
- **Permit**: The plugin can approve, deny, or delay Pod binding, adding conditions that must be met before proceeding. Delays may include timeouts, reverting to deny if not resolved.
- **PreBind**: The plugin performs tasks required before binding, such as provisioning volumes or setting configurations. Errors during this phase cause the Pod to return to the scheduling queue.
- **Bind**: The plugin handles the actual binding of a Pod to a Node. It skips subsequent plugins once a Pod is successfully bound.
- **PostBind**: The plugin executes tasks after a Pod has been successfully bound, such as resource cleanup or status updates. This marks the end of the binding cycle.

Users can configure the `KubeSchedulerConfiguration` manifest to enable or disable default, as well as custom scheduler plugins, based on their requirements.

Device Plugins

Device plugins in Kubernetes provide a mechanism for advertising and managing specialized hardware resources, such as GPUs, FPGAs, and network adapters, to the kubelet. These plugins enable Kubernetes to discover and allocate hardware devices in a consistent and extensible manner without requiring changes to the core Kubernetes codebase.

Device plugins run as gRPC services on each node, registering themselves with the kubelet. They report the available resources and monitor their health. The kubelet then makes these resources available for scheduling by exposing them as extended resources in the Node API. Pods can request these resources using resource requests and limits, ensuring that workloads are assigned to nodes with the required hardware capabilities.

This approach decouples hardware-specific logic from Kubernetes, making it easier to integrate new types of devices. For example, GPU vendors like NVIDIA provide device plugins to enable seamless scheduling and monitoring of GPU workloads. Administrators can deploy and manage these plugins as DaemonSets, simplifying operations in large clusters.

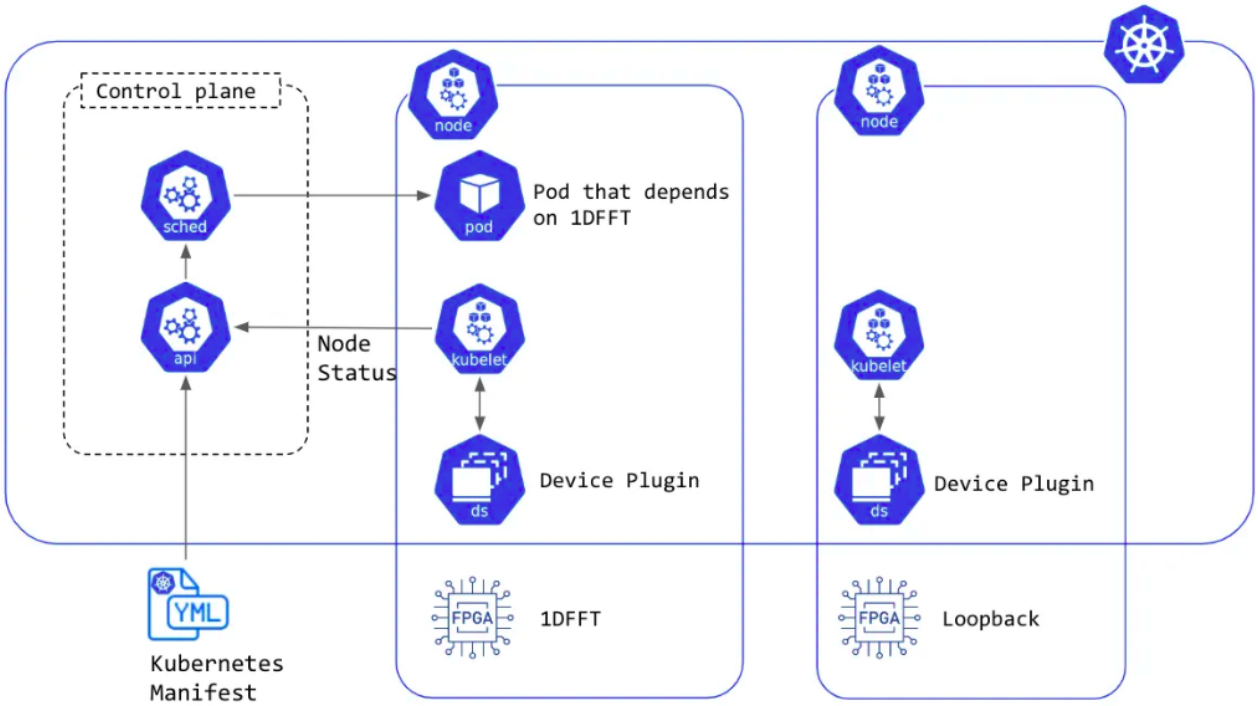


Figure 4.9.3: Kubernetes Cluster with FPGA Device Plugin Installed

Chapter 5

Performance of High Performance Computing Systems

5.1 Performance Degradation Factors in HPC Workloads

High-Performance Computing (HPC) systems are designed for the efficient execution of complex and demanding workloads. However, various factors can lead to performance degradation, resulting in longer execution times and reduced efficiency. Identifying and understanding these factors are critical for optimizing resource utilization and improving overall system performance. Performance bottlenecks in HPC clusters can occur both within the cluster multicore nodes and in the cluster interconnects. While cluster schedulers typically consider coarse-grained resource demands such as the number of nodes, processors, physical memory, swap space, and disk space, they often lack mechanisms to address the sensitivity of jobs to resource contention [1].

5.2 The Noisy Neighbor Effect

The "Noisy Neighbor Effect" is an inevitable consequence of shared infrastructure and multi-tenancy on the Cloud. When multiple applications or virtual machines are hosted inside the same physical server, the co-location of them may lead to increased performance degradation of the others. Furthermore, an application might be prone to performance degradation when co-executed with other workloads. This phenomenon occurs due to contention of limited processing resources of the physical host. One example is cache contamination, where two or more applications simultaneously load their data into the cache, competing with each other for the finite cache capacity of the processor. This results in applications having a "polluted" cache workspace where the blocks fetched from memory might be replaced by blocks of other co-running processes, leading to increased cache misses and degraded execution times.

In addition, when considering that in the cloud, applications may be activated or deactivated at arbitrary times, an increased variation in the performance of applications is observed. It is obvious that this effect heavily depends on the nature of the applications, e.g. if the program instructions are dominated by the memory read/write operations or the number of floating point operations (FLOPs) in the processor, or if there is aggressive data sharing and heavy utilization of communication primitives such as locks and barriers, between the threads. However, CSPs should carefully schedule and select resources for co-running applications to mitigate interference and meet the SLA requirements.

When it comes to Cloud-Native technology, where containerization and orchestration of workloads is utilized, numerous works have been published that attempt to tackle the problem of resource optimization in multi-tenant clusters. Research on scheduling and optimizing various workloads has been conducted across different domains, including Deep Learning, Industrial IoT, Batch Processing, and heterogeneous applications running within the same cluster [24] [7] [5].

5.3 Estimating Performance Bottlenecks on Multicore Architectures

Slowdown in Workload Execution

Performance degradation due to resource contention can reach as high as 200% when multiple threads or processes run concurrently on a multicore CPU [1]. This degradation stems from competition for shared resources such as last-level caches, memory controllers, system request queues, and prefetch bandwidth. While workload consolidation improves hardware utilization, it can also lead to performance degradation. Severe performance degradation can negate the benefits of consolidation, violate customer QoS constraints, cause unacceptable application slowdowns, and potentially waste energy despite power savings. Traditional methods for estimating performance degradation, such as analytical modeling based on the performance counter values of the utilized processor, heuristic models, and trial-and-error methods, have limitations in effectively handling the complexities of modern CPUs.

Machine learning has emerged as a promising alternative for modeling performance degradation on multicore systems. Machine learning excels at discovering complex relationships between various factors and filtering out irrelevant ones. This capability makes it well-suited for analyzing the numerous performance events available on modern CPUs to identify those correlated with sharing-induced degradation. Instead of perturbing workload execution or requiring prior knowledge of the applications, performance degradation can be estimated online using performance counter values obtained from the consolidated workload. This approach allows for real-time assessment of degradation without the need for isolation or disruption of ongoing execution.

Contentiousness and Sensitivity

Tang et al. [19] aim to characterize workloads based on the interference phenomena they impose or suffer from by introducing two properties, Contentiousness and Sensitivity respectively.

Contentiousness refers to an application's potential to cause performance degradation in co-running applications due to its demand for shared resources. It can be measured by quantifying the performance degradation experienced by other applications when co-scheduled with the target application. Factors contributing to contentiousness include high cache line requests per millisecond (LLC Lines In/ms), indicating heavy memory bandwidth usage.

Sensitivity, on the other hand, denotes an application's potential to suffer performance degradation due to interference from co-running applications. It is measured by quantifying the performance degradation the application experiences when co-scheduled with contentious applications. Sensitivity is influenced by the application's reliance on shared resources and its data reuse patterns.

Contentiousness and sensitivity are distinct characteristics of an application and are not strongly correlated. An application can be highly contentious but not sensitive to contention, and vice versa. For instance, streaming applications are considered contentious due to their high memory bandwidth usage, but they may not be significantly affected by cache contention. However, they are highly sensitive to memory bandwidth contention.

This distinction arises because contentiousness is directly related to the pressure an application exerts on shared resources, while sensitivity is determined by its dependence on those resources. The degree of pressure and reliance can vary significantly depending on the specific resource and the application's characteristics. For example, an application may occupy a substantial portion of the last-level cache (LLC) but not experience a high miss rate if its working set fits within the cache. In this scenario, the application would be considered contentious due to its high cache usage but not necessarily sensitive to contention as long as its cache needs are met. Similarly, an application may generate many prefetch requests but derive minimal benefit from them. This indicates high prefetcher usage but low reliance, meaning the application is contentious in terms of prefetcher utilization but not highly sensitive to contention.

Understanding the contention characteristics of an application requires a holistic view of the entire memory subsystem, as contention can occur in various components beyond just the LLC, including memory bandwidth,

prefetchers, and memory controllers.

Memory Bound and Compute Bound Applications

Hutcheson et al. describe the differences between two types of workloads, memory-bound and compute-bound, in an attempt to analyze performance in HPC workloads [6]. Memory-bound applications are limited by the speed at which data can be transferred to and from memory. In these situations, the processor is often idle because it is waiting for the memory controller to fulfill its requests, thus the processor cannot use its full computational potential. These types of applications typically involve low computational intensity, where the number of operations performed per memory access is minimal. For example, applications that perform simple vector operations, like copying, adding, or scalar operations on large arrays of data, are often memory-bound because the processor can perform the operations much faster than the memory controller can provide the data. In contrast, compute-bound applications are limited by the rate at which the processor can perform arithmetic operations. This occurs when the processor cannot keep up with the rate at which data is delivered by the memory subsystem, resulting in a computational bottleneck. Compute-bound problems often involve a large number of computations on a limited amount of data. Examples of compute-bound tasks include random number generation and dynamic programming, where extensive calculations are performed on relatively small datasets. The distinction between these two types of applications is crucial for optimization, as it allows programmers to identify the limiting factor in their algorithms.

In this thesis, we propose a scheduling mechanism inspired by the classification of workloads into memory-bound and compute-bound categories. Our approach leverages offline profiling of various benchmarks to characterize applications based on the performance bottlenecks they impose. We will co-execute different categories of workloads and observe how our resource allocator performs compared to the default behavior of the cluster. This profiling-driven approach seeks to improve scheduling efficiency and overall performance in multi-tenant cloud environments.

Chapter 6

Maestro: Fine-Grained Scheduling and Resource Allocation in Kubernetes

6.1 Overview

In this thesis, we introduce a framework that enables cluster administrators and future researchers to experiment with fine-grained resource allocation policies, aiming to improve workload performance while optimizing hardware utilization. We leverage Kubernetes' extensible architecture to create a mechanism for workload pinning and isolation across CPU and memory resources in a declarative manner. We describe the design and implementation of our framework, including a controller manager for our custom resources, a node-level daemon that interacts with the API server and Linux kernel to enforce manifests, and a custom scheduler plugin that schedules Pods and applies resource bindings based on the workload type. Our framework is designed to be extensible, enabling the implementation of more sophisticated resource allocation policies in the future for enhanced resource optimization and cost-efficiency.

Custom Resource Definitions

We introduce two supervisory Custom Resource Definitions (CRDs) that provide detailed information about the available CPU and memory resources on each node, as well as the resource allocations associated with each Pod. This ensures that the cluster and the administrators are always aware of which Pods are running on specific CPUs and memory nodes, as well as the exclusivity of their resource allocations. Furthermore, if workloads are observed to be throttled or suboptimally placed, administrators can dynamically reallocate them to other resources by modifying the applied manifests.

`NodeCPUTopology`

A `NodeCPUTopology` custom resource describes the available CPUs and NUMA nodes of each cluster node. This includes logical cores, physical cores, sockets, as well as NUMA architecture information. Listing 6.1 shows the `NodeCPUTopology` of a node in our cluster. It describes a node with four logical cores and two physical cores, contained in one NUMA node.

`NodeCPUTopology` resources of all the nodes in our cluster are automatically created on the startup of the controller manager we developed. Our custom scheduler is aware of all the available CPU resources of each node by querying the API server, and is able to make resource binding decisions by consuming the `PodCPUBinding` API, which we describe in the following section.

```
apiVersion: cslab.ece.ntua.gr/v1alpha1
kind: NodeCPUTopology
metadata:
  labels:
    node-role.kubernetes.io/control-plane: ""
  name: node-1-cputopology
```

```
spec:
  nodeName: node-1
  topology:
    cpus:
    - 0
    - 1
    - 2
    - 3
    numaNodes:
      "0":
        cpus:
        - 0
        - 1
        - 2
        - 3
    sockets:
      "0":
        cores:
          "0":
            cpus:
            - 0
            - 1
          "1":
            cpus:
            - 2
            - 3
        cpus:
        - 0
        - 1
        - 2
        - 3
status:
  internalIP: 100.92.105.6
  resourceStatus: Fresh
```

Listing 6.1: `NodeCPUTopology` Manifest Sample

`PodCPUBinding`

The `PodCPUBinding` is a custom resource that describes the CPU and memory resources allocated to a specific Pod that's running within the cluster. Listing 6.2 shows a `PodCPUBinding` manifest that can be applied both by the cluster's administrator, as well as an application that is able to make resource allocation decisions.

```
apiVersion: cslab.ece.ntua.gr/v1alpha1
kind: PodCPUBinding
metadata:
  finalizers:
  - cslab.ece.ntua.gr/pod-cpu-binding-finalizer
  name: streamcluster-4-pcb
  namespace: benchmarks
spec:
  cpuSet:
  - cpuID: 0
  - cpuID: 2
  - cpuID: 4
  - cpuID: 6
  exclusivenessLevel: NUMA
  podName: streamcluster-4
status:
  nodeName: node-4
  resourceStatus: Applied
```

Listing 6.2: `PodCPUBinding` Manifest Sample

The field `exclusivenessLevel` allows users to specify different levels of resource isolation for the requested cores and memory nodes.

- `None` – The Pod does not hold exclusive access to any of the requested resources.

- `CPU, Core, Socket, NUMA` – The Pod exclusively reserves the specified resource type, ensuring that any other Pods requesting resources at the same exclusiveness level are rejected by the controller manager.

This feature proves particularly effective for latency-critical workloads, which often benefit significantly from running in isolation on dedicated resources.

As we will describe in depth in a later section, the controller manager of our mechanism is the entry-point of every applied CPU binding. The `PodCPUBinding` controller is responsible for validating the custom resource and communicating with the Pod running the node via a gRPC service to command the resource allocation that was manifested.

6.2 Application Components

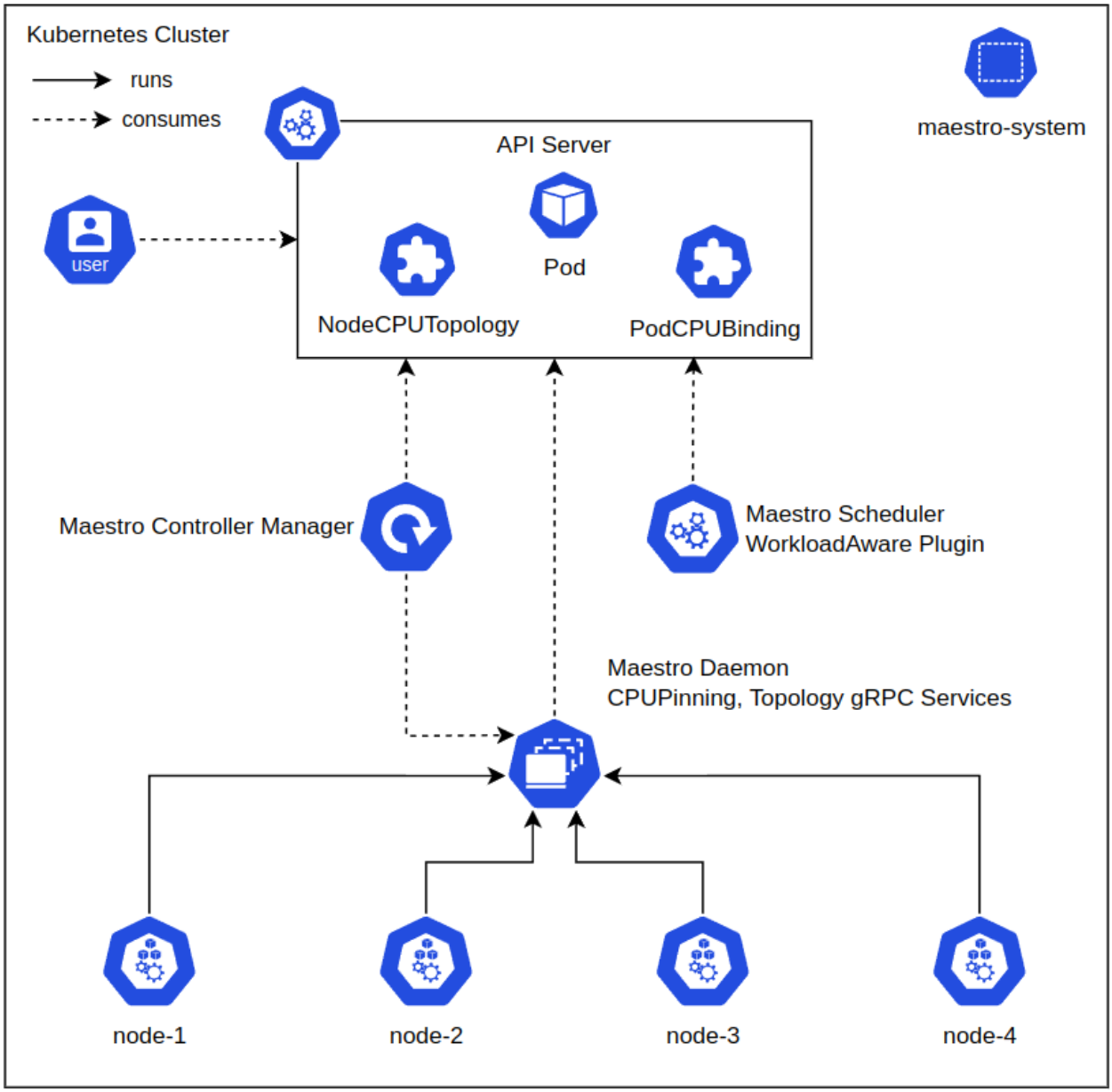


Figure 6.2.1: Maestro Architecture

Figure 6.2.1 displays the different components of Maestro, and how they communicate with each other. The three main components, the controller manager, the daemon and the scheduler consume the cluster's API server to remain aware of all the custom resources that have been applied. Additionally, the DaemonSet's Pods are run on every node in the cluster, and the controller manager consumes the gRPC services that they expose in order to communicate with the Linux kernel of each node.

6.2.1 Controller Manager

The Controller Manager, developed in Go with the Kubebuilder framework, is an application that follows the same principles as the `kube-controller-manager`, for our Custom Resource Definitions. The controller manager packages both `NodeCPUTopology` and `PodCPUBinding` controllers. When a custom resource create/update/delete event occurs on the API server, the corresponding controller is triggered and attempts to fulfill the applied manifest's request, bringing the cluster's current state to the desired one.

A key functionality of the controller manager is validating and processing `PodCPUBinding` manifests. When a manifest is applied or updated, the controller manager validates it against the `NodeCPUTopology` of the node and its currently available resources. If the manifest contains invalid configurations, the controller manager prevents the resource binding from being applied. It also generates events that users can inspect using commands such as `kubectl describe` or `kubectl get events`.

```
$ kubectl describe pcb
Name:         streamcluster-4-pcb
Namespace:    benchmarks
Labels:       <none>
Annotations:  <none>
API Version:  cslab.ece.ntua.gr/v1alpha1
Kind:         PodCPUBinding
Metadata:
  Creation Timestamp:  2024-12-27T19:18:55Z
  Finalizers:
    cslab.ece.ntua.gr/pod-cpu-binding-finalizer
  Generation:        1
  Resource Version:  18805247
  UID:               2b8821a2-4d34-4ccc-8251-e7bfb0545da0
Spec:
  Cpu Set:
    Cpu ID:             0
    Cpu ID:             2
    Cpu ID:             4
    Cpu ID:             6
  Exclusiveness Level:  Core
  Pod Name:             streamcluster-4
Status:
  Node Name:        node-4
  Resource Status:  Applied
Events:
  Type    Reason     Age   From                        Message
  ----    ------     ----  ----                        -------
  Normal  Validated  13s   podcpubinding-controller  CPU binding is validated
  Normal  Applied    8s    podcpubinding-controller  Applied CPUSet [0 2 4 6], MemSet [0]
```

Listing 6.3: Successful `PodCPUBinding`

```
$ kubectl describe pcb
Name:         streamcluster-4-pcb
Namespace:    benchmarks
Labels:       <none>
Annotations:  <none>
API Version:  cslab.ece.ntua.gr/v1alpha1
Kind:         PodCPUBinding
Metadata:
  Creation Timestamp:  2024-12-27T19:18:55Z
  Finalizers:
    cslab.ece.ntua.gr/pod-cpu-binding-finalizer
  Generation:        3
  Resource Version:  18805528
```

```
  UID:               2b8821a2-4d34-4ccc-8251-e7bfb0545da0
Spec:
  Cpu Set:
    Cpu ID:             0
    Cpu ID:             2
    Cpu ID:             4
    Cpu ID:             26
  Exclusiveness Level:  Core
  Pod Name:             streamcluster-4
Status:
  Node Name:        node-4
  Resource Status:  InvalidCPUSet
Events:
  Type     Reason             Age                From                         Message
  ----     ------             ----               ----                         -------
  Warning  InvalidCPUSet      1s                 podcpubinding-controller  CPUs [0 2 4 26]
    do not exist in node node-4-cputopology
```

Listing 6.4: Failed `PodCPUBinding`

Additionally, when a `PodCPUBinding` is created, the Finalizer `cslab.ece.ntua.gr/pod-cpu-binding-finalizer` is attached to it. This ensures that when a resource is deleted, the finalizer attempts to release the resources that were assigned to the Pod.

6.2.2 Daemon

The Daemon is a node-level application deployed as a DaemonSet in the cluster. It exposes two gRPC services to interact with the controller manager. The `Topology` service allows the controller manager to gather information about the underlying CPU and memory architecture of each node, while the `CPUPinning` service enforces specific CPU and memory resource allocations for Pods by modifying the host's `cgroups`. The application also periodically reconciles shared resources for non-bound Pods, ensuring consistency and preventing resource drift, with the reconciliation interval configurable via the DaemonSet arguments.

When an `ApplyPinning` request is made from the controller manager, the daemon is responsible for modifying the following `cgroups` values, based on the CPU pinning arguments and the Pod's resource requests and limits:

- `cpuset.cpus` – Specifies the list of CPUs assigned to the Pod, as defined in the manifest.
- `cpuset.mems` – The list of memory nodes that the assigned CPUs belong in.
- `cpu.cfs_quota_us` – Sets a hard limit on the CPU time (in microseconds) that the Pod can use within a scheduling period.
- `cpu.shares` – Specifies the relative weight for CPU allocation. Pods with higher shares get more CPU time when resources are contended.

The `GetTopology` request returns a structured object with CPU information of each node. Internally, it executes the `lscpu` command and returns CPUs arranged by Socket or by NUMA node. When creating a `NodeCPUTopology` resource on the API server, the `NodeCPUTopology` controller queries the specified node to populate the resource with CPU-related information and store it on the server. This enabled other API consumers, such as the custom scheduler to be aware of each node's underlying CPU architecture and make more fine-grained scheduling and resource binding decisions.

6.2.3 Scheduler

The Scheduling Framework allows developers to create and bundle custom scheduler plugins tailored to specific needs. Developers can implement plugins at any stage of the scheduling and binding cycles, introducing custom logic to influence scheduling decisions. Leveraging this framework, we developed the WorkloadAware plugin, which integrates custom resources to enable fine-grained scheduling and resource binding.

Our scheduler plugin requires the Pod to be scheduled to contain a "Workload Type" label. The supported workload type labels are the following:

- `cslab.ece.ntua.gr/workload-type: cpu-bound`

  Workloads with performance that depends on the available CPU resources.

- `cslab.ece.ntua.gr/workload-type: memory-bound`

  Workloads with performance that depends on the available memory bandwidth.

- `cslab.ece.ntua.gr/workload-type: io-bound`

  Workloads that have threads with high IO wait time.

- `cslab.ece.ntua.gr/workload-type: best-effort`

  Workloads with very loose performance requirements.

The plugin decides on how to bind CPU resources to the Pods of the above types based on the following schema:

- **MemoryBound**: Threads are placed on different memory nodes (sockets).
- **CPUBound**: Threads are placed on different, non-utilized cores, on the same socket.
- **IOBound**: Threads are placed on the same physical core, or more cores are utilized if needed.
- **BestEffort**: Every thread is placed on the same logical core.

The WorkloadAware plugin can be configured to employ different scoring policies and feature flags, making its scheduling decisions even more configurable.

The two complementary scoring policies that are supported are the following:

- **MaximumUtilization**: The plugin tries to maximize the utilization of the resources.
- **Balanced**: The plugin places the Pods in a balanced way across the cluster.

The extra supported features are the following:

- **PhysicalCores**: Use only physical cores for allocation (Default `exclusivenessLevel` value is `Core`).
- **MemoryBoundExclusiveSockets**: Allocate memory nodes (sockets) exclusively for MemoryBound workloads (Default `exclusivenessLevel` value for MemoryBound workloads is `Socket`).
- **BestEffortSharedCPUs**: Allow BestEffort workloads to share logical cores.

The WorkloadAware plugin implements the following scheduling cycle stages:

- **PreFilter**: The plugin gathers the custom resources from the API server and initializes the state of the cycle.
- **Filter**: The plugin filters out all the nodes that cannot meet the Pod's resource requirements. Based on the workload type and the enabled features, the filter step might take into account different resource levels (CPU, Core Socket, NUMA) when filtering out unfeasible nodes.
- **Score**: The plugin scores each feasible node based on its current utilization and the selected policy (Balanced or MaximumUtilization).
- **Bind**: The plugin discovers CPU resources that meet the Pod's requirements and binds the Pod to the nominated node and the decided resources (CPUs, memory nodes).

6.3 WorkloadAware Scheduling Algorithm

In this section we will describe the scheduling algorithm the WorkloadAware plugin implements, and how different scoring policies and enabled features influence its decisions.

Let:

- N be the total number of nodes.

- $\text{RunningProgramThreads}_i$ denote the number of currently running program threads on node i.
- LogicalCores_i represent the number of logical cores on node i.
- Sockets_i represent the number of sockets on node i.
- $\text{AllocatableSockets}_i$ represent the number of sockets that can accommodate the Pod's requirements on node i.
- $\text{AllocatableCores}_i$ represent the number of cores that can accommodate the Pod's requirements on node i.
- AllocatableCPUs_i represent the number of CPUs that can accommodate the Pod's requirements on node i.

MemoryBound Workload

For MemoryBound workloads, the score is calculated as:

$$\text{score} = \left\lceil \left(\frac{\text{AllocatableSockets}_i}{\text{Sockets}_i} \times 100 \right) \right\rceil + \text{AllocatableSockets}_i$$

Where:

$$\text{AllocatableSockets}_i = \sum_{j=1}^{n} \text{AllocatableSocket}_j \quad \text{for node } i,$$

$$\text{Sockets}_i = \sum_{j=1}^{n} \text{Sockets}_j \quad \text{for node } i.$$

CPUBound Workload

For CPUBound workloads, the score is calculated as:

$$\text{score} = \left\lceil \left(\frac{\text{AllocatableCores}_i}{\text{Cores}_i} \times 100 \right) \right\rceil + \text{AllocatableCores}_i$$

Where:

$$\text{AllocatableCores}_i = \sum_{j=1}^{n} \text{AllocatableCore}_j \quad \text{for node } i,$$

$$\text{Cores}_i = \sum_{j=1}^{n} \text{Core}_j \quad \text{for node } i.$$

IOBound Workload

For IOBound workloads, the score is calculated as:

$$\text{score} = \left\lceil \left(\frac{\text{AllocatableCores}_i}{\text{Cores}_i} \times 100 \right) \right\rceil + \text{AllocatableCores}_i$$

Where:

$$\text{AllocatableCores}_i = \sum_{j=1}^{n} \text{AllocatableCore}_j \quad \text{for node } i,$$

$$\text{Cores}_i = \sum_{j=1}^{n} \text{Core}_j \quad \text{for node } i.$$

BestEffort Workload

For BestEffort workloads, the score is calculated based on whether the feature `PhysicalCores` is enabled or not:

- If the feature `PhysicalCores` is enabled:

$$\text{score} = \left\lceil \left(\frac{\text{AllocatableCores}_i}{\text{Cores}_i} \times 100 \right) \right\rceil + \text{AllocatableCores}_i$$

- If the feature `PhysicalCores` is not enabled:

$$\text{score} = \left\lceil \left(\frac{\text{AllocatableCPUs}_i}{\text{CPUs}_i} \times 100 \right) \right\rceil + \text{AllocatableCPUs}_i$$

Where:

$$\text{AllocatableCores}_i = \sum_{j=1}^{n} \text{AllocatableCore}_j \quad \text{for node } i,$$

$$\text{Cores}_i = \sum_{j=1}^{n} \text{Core}_j \quad \text{for node } i,$$

$$\text{AllocatableCPUs}_i = \sum_{j=1}^{n} \text{AllocatableCPU}_j \quad \text{for node } i,$$

$$\text{CPUs}_i = \sum_{j=1}^{n} \text{CPU}_j \quad \text{for node } i.$$

Policy Adjustment

Since the score is a metric of capacity, if we want to maximize utilization, we should favor nodes with higher scores. Thus, after calculating the initial score based on workload type, the score may be adjusted based on the configured policy (`args.Policy`):

- For `MaximumUtilization`:

$$\text{score} = \text{math.MaxInt64} - \text{score}$$

- For `Balanced`, no further action is taken on the score.

Node and Resource Binding of Pod

After deciding on which node the Pod should be scheduled, the `WorkloadAware` plugin also overrides the `Bind` step of the scheduling process. In this phase, the scheduler:

- Binds the Pod to the nominated Node.
- Calculates the Pod resources based on the node's available resources and creates the `PodCPUBinding` object of the scheduled Pod which contains the selected resources, as well as the exclusiveness level, based on the workload type.

6.4 Deployment and Configuration

The deployment of the mechanisms consists of two stages:

1. Apply the Custom Resource Definitions that were generated by the Kubebuilder framework
2. Modify the DaemonSet and the KubeSchedulerConfiguration manifests to match the desired configuration

3. Apply the Controller Manager Deployment, the Maestro DaemonSet and the Maestro Scheduler Deployment

```
args:
  - '--node-name=$(NODE_NAME)'
  - '--container-runtime=containerd'  # containerd, docker, kind
  - '--cgroups-path=/cgroup'
  - '--cgroups-driver=systemd'        # systemd, cgroupfs
  - '--reconcile-period=15s'
  - '--verbosity=3'
```

Listing 6.5: Maestro DaemonSet Arguments

```
apiVersion: kubescheduler.config.k8s.io/v1
kind: KubeSchedulerConfiguration
leaderElection:
  leaderElect: false
profiles:
  - schedulerName: maestro
    plugins:
      preFilter:
        enabled:
          - name: WorkloadAware
      filter:
        enabled:
          - name: TaintToleration       # Enabled default plugin
          - name: WorkloadAware
      score:
        enabled:
          - name: WorkloadAware
      bind:
        enabled:
          - name: WorkloadAware
        disabled:
          - name: '*'
    pluginConfig:
      - name: WorkloadAware
        args:
          policy: MaximumUtilization    # MaximumUtilization, Balanced
          features:
            - PhysicalCores
            - BestEffortSharedCPUs
#           - MemoryBoundExclusiveSockets
```

Listing 6.6: `KubeSchedulerConfiguration` for the Maestro Scheduler

Chapter 7

Experimental Setup and Motivational Analysis

7.1 Baseline Hardware and Virtual Machine Configuration

Our experimental setup consists of four virtual machines running Ubuntu Server 18.04 Bionic Beaver, distributed across two host machines within the lab's infrastructure. On these machines, we utilized kubeadm to setup a Kubernetes v1.31.0 cluster. To support benchmarking scenarios involving co-running applications with numerous threads, we also deployed a more powerful, dual-socket worker on a second host machine, termi9.

Our termi10 host consists of 2 sockets, each one of them containing 6 cores with enabled hyperthreading. We reserve 4 cores (8 vCPUS) from socket-0 for the worker node-2 and 4 cores (8 vCPUS) from socket-1 for the worker node-3. The CPU pinnings are configured in a way such that a host's physical core is mapped to a physical core of a virtual machine. The rest of the termi10 domains (node-1, nfs) are mapped to the rest of the available cores in a similar fashion.

To conduct benchmarks for workloads that heavily utilize memory resources, possibly on more than one NUMA node, we extended our cluster's workers to include a Kubernetes node with 2 memory nodes. On the host machine termi9, we spawned a virtual machine node-4 with 5 cores (10 vCPUS) from socket-0 and 5 cores (10 vCPUS) from socket-1. The memory of the virtual machine is distributed across the two memory nodes of the host. The CPU pinnings are configured in a way such that a host's physical core is mapped to a physical core of a virtual machine. This configuration will allow us to run memory-bound workloads that require the use of more than one CPU and memory node.

Table 7.1: Virtual Machine Configurations

Host	CPU Model	VM Name	Sockets	Cores/Socket	vCPUs	Memory (GB)	NUMA Nodes	OS
termi10	Intel Xeon E5645	node-1	1	2	4	16	1	Ubuntu Server 18.04
		node-2	1	4	8	16	1	Ubuntu Server 18.04
		node-3	1	4	8	16	1	Ubuntu Server 18.04
		nfs	1	2	4	8	1	Ubuntu Server 18.04
termi9	Intel Xeon E5645	node-4	2	10	20	64	2	Ubuntu Server 18.04

7.2 Collecting and Monitoring System Metrics

7.2.1 Intel® Performance Counter Monitor

We deployed Intel's Performance Counter Monitor (PCM) on each of the host machines, enabling us to retrieve CPU-related metrics such as DRAM bandwidth, instructions per cycle, L2 and L3 cache misses per instruction, and other relevant statistics. Intel PCM Sensor Server is a daemon that's running inside each

host machine and retrieves performance counters from the hypervisor's hardware. It also exposes an HTTP endpoint that can be consumed by a timeseries database like Prometheus.

7.2.2 Prometheus, Grafana, Node Exporter, cAdvisor

To enable real-time monitoring and visualization of system metrics, we integrated Prometheus, Grafana, Node Exporter, and cAdvisor into our infrastructure.

Prometheus is employed as the primary time-series database, responsible for collecting and storing metrics from various sources. It queries performance data exposed by Intel PCM's HTTP endpoint and other exporters, enabling centralized monitoring and alerting.

Node Exporter is deployed on each host machine to gather detailed system-level metrics, including CPU usage, memory utilization, disk I/O, and network activity. These metrics provide insights into the overall health and performance of the virtual machines and host systems.

cAdvisor (Container Advisor) is used to monitor resource usage and performance statistics of containers running on the cluster's nodes. It provides visibility into CPU, memory, filesystem, and network metrics at the container level, making it particularly useful for containerized workloads.

Finally, Grafana is used as the visualization layer, offering interactive dashboards that allow us to analyze the collected data and monitor trends over time. Grafana seamlessly integrates with Prometheus, enabling us to build custom dashboards for visualizing system and application performance metrics.

7.3 Benchmark Suites

The CloudSuite Benchmark Suite

CloudSuite is a collection of benchmarks designed to evaluate the performance of cloud systems. They use software commonly found in cloud environments to mimic realistic workloads. For instance, the Data Serving benchmark evaluates the performance of the NoSQL database, Cassandra, by populating it with a 10GB table of 1KB records and then subjecting it to different loads. Users can adjust the number of reader and writer threads used by Cassandra. The Web Serving benchmark uses MariaDB, Memcached, and the social networking engine Elgg to emulate a web server handling dynamic and static content. Users can scale this benchmark by modifying the number of users that log in to the server and request pages. The Media Streaming benchmark utilizes Nginx as a streaming server to host videos with resolutions ranging from 240p to 720p. A client program requests videos at a specified rate, putting stress on the server. Users can control the encryption used for requests and define session lists for the client. The Graph Analytics benchmark, leveraging the Spark framework and its GraphX library, performs graph analysis on a large dataset derived from Twitter. It supports algorithms like PageRank, Connected Components, and Triangle Count, with the latter requiring a substantial amount of memory. Finally, the In-Memory Analytics benchmark uses Apache Spark and MLlib to evaluate the performance of a collaborative filtering algorithm. This benchmark relies on the MovieLens dataset, which contains user-movie ratings. This setup allows for the measurement of the time taken to generate movie recommendations, underlining the importance of sufficient memory allocation for in-memory execution.

The PARSEC Benchmark Suite

PARSEC is specifically designed for evaluating the performance and characteristics of chip multiprocessors (CMPs). It comprises a diverse set of applications and kernels chosen from a wide range of application domains, reflecting the shift in computing towards multi-core processors. PARSEC includes benchmarks like blackscholes, which uses the Black-Scholes PDE to calculate prices for a portfolio of European options; bodytrack, a computer vision application that tracks human body pose from multiple camera inputs; canneal, which minimizes routing cost in chip design using simulated annealing; dedup, a data compression kernel that employs deduplication techniques to compress data streams; facesim, an application that simulates facial animations based on muscle activation inputs; ferret, which performs content-based similarity searches on image databases; fluidanimate, an application simulating fluid dynamics for animation purposes; freqmine,

which employs the FP-growth algorithm to mine frequent itemsets from transaction databases; streamcluster, an online clustering benchmark that processes streaming data points; swaptions, a financial application that prices swaptions using the HJM framework and Monte Carlo simulation; vips, an image processing application that performs a sequence of operations on images; and x264, a video encoder that uses the H.264/AVC standard to compress video streams. These applications exhibit different parallelization strategies, memory access patterns, and communication behavior, providing a comprehensive evaluation platform for CMP architectures. They use large datasets and are designed to stress the memory hierarchy, interconnection networks, and synchronization mechanisms of CMPs. The use of state-of-the-art algorithms and techniques ensures that PARSEC reflects the evolving nature of computing workloads in the multi-core era.

The STREAM Benchmark

STREAM is a benchmark focused on measuring the sustainable memory bandwidth of a computer system. This benchmark utilizes four vector kernels: Copy, Scale, Sum, and Triad. These kernels operate on large arrays of data and avoid data reuse in the cache or registers, ensuring that the benchmark remains memory-bound. The key metric in STREAM is the sustainable bandwidth, expressed in MB/s. This metric is calculated based on the time taken to complete the kernels and provides valuable insights into the performance of the memory subsystem. STREAM's utility lies in its ability to stress the memory system and expose performance bottlenecks that may not be apparent in benchmarks that are less memory-intensive.

7.4 Workload Classification

7.4.1 Packed-Friendly

Packed-friendly applications can optimize performance by keeping threads localized within a single NUMA node, minimizing inter-node communication overhead. Packed-friendly workloads are limited by the rate at which the processor can perform arithmetic operations. This occurs when the processor cannot keep up with the data delivered by memory, resulting in a computational bottleneck. These applications often involve a large amount of computation on a limited field of data. Examples include Monte Carlo simulations for random number generation and dynamic programming. In packed-friendly applications, increasing the number of arithmetic operations will cause performance to decrease until the program is completely limited by the computational speed of the processor, regardless of problem size. The benefit of cache memory is reduced as computational intensity increases, until a main memory call takes less time than processing the data.

7.4.2 Spread-Friendly

Spread-friendly workloads, on the other hand, prefer their threads to be distributed across many NUMA nodes. They are applications that are limited primarily by the speed at which data can be transferred to and from memory, a factor that could be mitigated if the program utilized more than one memory node. In these situations, the processor is often idle because it is waiting for the memory controller to fulfill its requests, which is slower than the processor's computational speed. This commonly occurs in applications with low computational intensity, where the number of operations performed per memory access is low. An example of a spread-friendly application would be one that performs simple vector operations on large arrays. The performance of these applications is highly dependent on memory bandwidth. Factors such as the size of the data relative to the cache size and the pattern of memory access (sequential vs. random) can significantly affect whether an application is spread-friendly. When data is accessed randomly, the system is unable to properly utilize the CPU cache, resulting in a much greater percentage of calls to main memory, and thus, a lower effective bandwidth.

In 7.4.1 we observe how allocating the threads of Spread-Friendly workloads to different sockets impacts the performance metrics of the application. On the first execution, both sockets of the worker node are utilized, which leads to significantly less consumed bandwidth, enabling other processes running on the same NUMA node to have faster access to their memory addresses. Furthermore, the Instructions Per Cycle (IPC) seem to double since on the second run, only Socket 0's cores are enabled.

7.4.3 Isolation-Friendly

Isolation-friendly workloads perform best when executed independently, avoiding interference from other threads or processes. These applications benefit from having dedicated resources, such as CPU cores, cache, and memory bandwidth, ensuring consistent performance without contention. Unlike spread-friendly or packed-friendly workloads, isolation-friendly applications do not exhibit significant performance gains when optimized for NUMA configurations. Instead, their performance is primarily influenced by minimizing resource sharing and contention rather than optimizing data locality or memory access patterns. Examples include latency-sensitive tasks like encryption algorithms, real-time processing systems, and single-threaded workloads, where predictability and resource exclusivity outweigh the advantages of NUMA-aware placement strategies.

7.4.4 Agnostic

Agnostic workloads demonstrate negligible performance differences across NUMA configurations, whether deployed as packed, spread, or in their default (vanilla) configuration. These applications exhibit balanced computational and memory access patterns that neither saturate processing power nor memory bandwidth, making them resilient to variations in resource placement. Agnostic workloads often include parallel tasks, such as batch data processing and rendering pipelines, where independent units of work execute with minimal synchronization overhead. Due to their flexibility, these workloads are ideal for environments with dynamic resource allocation or cloud-based infrastructures where NUMA-awareness may not be a primary optimization concern.

Classification Algorithm

Let:

- W be the currently examined workload
- T_V be the average vanilla execution time for workload W
- T_P be the average execution time for W considered as Packed-Friendly
- T_S be the average execution time for W considered as Spread-Friendly

The slowdown of an application is given by the following equation:

$$S = \frac{\text{Execution Time under Current Configuration}}{\text{Baseline Execution Time}} \tag{7.4.1}$$

If we normalize all the execution times of the workload by dividing them with the vanilla execution time T_V, we obtain the following indicators:

- $S_V = 1$
- $S_P = \frac{T_P}{T_V}$, the slowdown of W when considered as Packed-Friendly
- $S_S = \frac{T_S}{T_V}$, the slowdown of W when considered as Spread-Friendly

We propose a profiling algorithm that leverages these metrics to classify the workload in one of the two categories. If there isn't any significant difference between the two metrics, we should decide whether the workload performs better with the isolation provided by our allocator (Isolation-Friendly) or if it is agnostic to any kind of isolation policy.

Algorithm 1: Classify Workload Based on Slowdown Values

Require: s_p: Slowdown in Packed-friendly configuration, s_s: Slowdown in Spread-friendly configuration, e: Decision threshold

Ensure: Workload classification (PackedFriendly, SpreadFriendly, IsolationFriendly, Agnostic)

$s_v \leftarrow 1.0$ {Slowdown in vanilla configuration}
$s_b \leftarrow \min(s_p, s_s)$ {Best configuration slowdown}
$s_w \leftarrow \max(s_p, s_s)$ {Worst configuration slowdown}
tag $\leftarrow$ PackedFriendly **if** $s_b = s_p$ **else** SpreadFriendly
if $s_w - s_b \geq e$ **then**
    {Significant difference between configurations}
    if $|s_v - s_b| < e$ **then**

        return tag
    else

        return tag
    end if
else
    {Small difference between configurations}
    if $s_v - s_b > e$ **then**
        {Best configuration significantly better than vanilla}
        if $s_v - s_w > e$ **then**
            {Also worst configuration significantly better than vanilla}

            return IsolationFriendly
        else
            {Best configuration significantly outperforms vanilla}

            return tag
        end if
    else
        {Best configuration similar to vanilla}
        if $|s_v - s_w| < e$ **then**

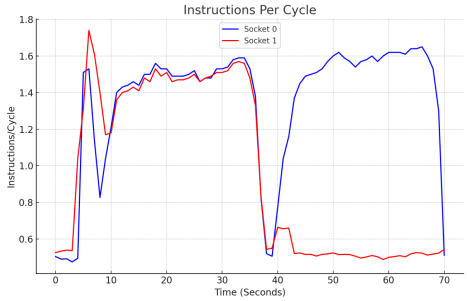
            return Agnostic
        else

            return Vanilla {Fallback, should not occur}
        end if
    end if
end if

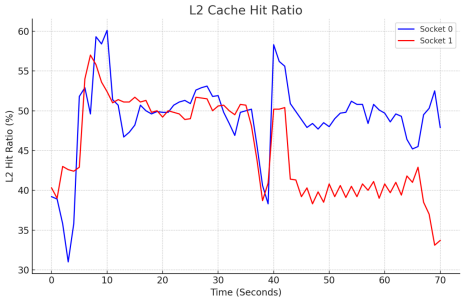
In the evaluation chapter, we will analyze the performance characteristics of the workloads outlined in Table 7.2, following the classification with our decision tree for the collected execution times. Following the profiling of our applications, we will employ our custom scheduling and resource allocation mechanism to assign resources based on the inferred tags. We will demonstrate how our mechanism achieves better slowdown values by utilizing the offline classification schema we propose.

Table 7.2: Benchmark Table Grouped by Suite

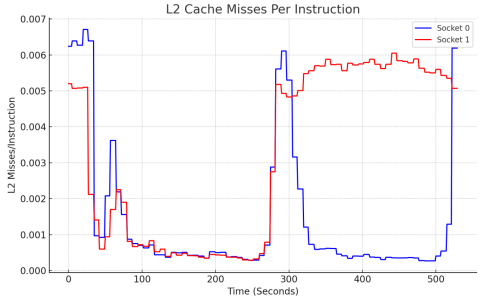
Suite	Benchmark	Internal Name	Summary	Algorithm	Dataset	Comments
CloudSuite	Graph Analytics	`ga`	Performs graph analytics on large-scale datasets using the Spark framework and its GraphX library.	PageRank, Connected Components, Triangle Count	A graph dataset generated from Twitter.	Requires significant memory, especially for Triangle Count, and supports multi-node deployment.
	In-Memory Analytics	`in-mem`	Evaluates the performance of an in-memory collaborative filtering algorithm using Apache Spark and MLlib.	Alternating Least Squares (ALS)	User-movie ratings datasets provided by Movielens.	Measures the time to compute movie recommendations, requires sufficient memory allocation for in-memory execution.
STREAM	STREAM	`stream`	Measures sustainable memory bandwidth.	Copy, Scale, Sum, Triad	Array of `double` with its size configured on compile time. Set to 256M elements.	Uses long vector operations to eliminate data reuse.
PARSEC	canneal	`canneal`	Minimizes routing cost in chip design.	Cache-aware simulated annealing	Netlist representing the chip design.	Employs a lock-free synchronization strategy based on data race recovery.
	fluidanimate	`fluidanimate`	Simulates incompressible fluid dynamics.	Smoothed Particle Hydrodynamics (SPH)	Particle data representing the fluid.	Emphasizes stability, accuracy, and speed, uses Verlet integration for particle updates.
	freqmine	`freqmine`	Mines frequent itemsets from a transaction database.	Array-based FP-growth	Transaction database, such as click streams or web documents.	Parallelized with OpenMP, susceptible to false sharing.
	streamcluster	`streamcluster`	Performs online clustering on streaming data points.	K-means clustering	Stream of data points with varying dimensionality.	Working set size is user-definable, transitions from memory-bound to computationally intensive.
	ferret	`ferret`	Performs content-based similarity search of feature-rich data, configured for image similarity search.	Multi-probe Locality Sensitive Hashing (LSH), Earth Mover's Distance (EMD)	Image database with varying number of images and queries.	Uses a pipeline model, working set size is dominated by the image database size, considered unbounded.



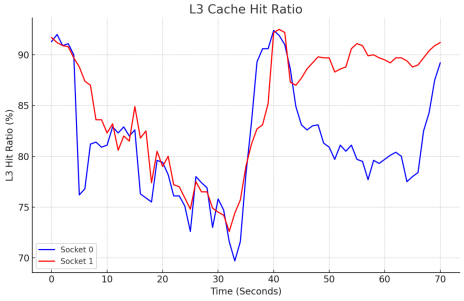
(a) Instructions Per Cycle



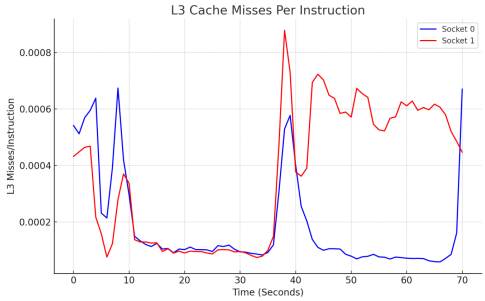
(b) L2 Cache Hit Ratio



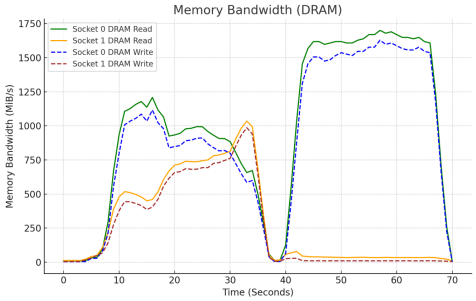
(c) L2 Cache Misses Per Instruction



(d) L3 Cache Hit Ratio



(e) L3 Cache Misses Per Instruction



(f) Memory Bandwidth (DRAM)

Figure 7.4.1: Performance metrics of two consecutive executions of `in-memory-analytics` with two threads. On the first run, the threads are distributed to the sockets, reducing the memory bandwidth needed and doubling the instructions executed per cycle.

Chapter 8

Evaluation

8.1 Classification of SpreadFriendly - PackedFriendly Workloads

In this section, we will attempt to classify our test benchmarks into two of the main categories of our allocator mechanism, PackedFriendly and SpreadFriendly. Table 8.1 presents the slowdown values of all the tested benchmarks, and the classification tag that our algorithm decided.

Benchmark	**Packed-Friendly**	**Spread-Friendly**	**Classification**
canneal-2	0.78	1.59	Packed Friendly
canneal-4	0.83	1.04	Packed Friendly
canneal-8	0.90	0.96	Packed Friendly
ferret-2	0.87	0.96	Packed Friendly
ferret-4	0.87	0.92	Packed Friendly
ferret-8	1.09	0.84	Spread Friendly
fluidanimate-2	0.80	0.98	Packed Friendly
fluidanimate-4	0.98	1.00	Agnostic
fluidanimate-8	1.15	0.88	Spread Friendly
freqmine-2	0.88	0.94	Packed Friendly
freqmine-4	0.96	0.95	Agnostic
freqmine-8	1.35	1.00	Spread Friendly
ga-2	0.98	0.98	Agnostic
ga-4	1.01	0.98	Agnostic
ga-8	1.68	0.91	Spread Friendly
in-mem-2	0.98	1.02	Agnostic
in-mem-4	0.98	1.01	Agnostic
in-mem-8	1.31	0.96	Spread Friendly
stream-2	0.83	0.94	Packed Friendly
stream-4	1.44	0.94	Spread Friendly
stream-8	1.16	0.97	Spread Friendly
streamcluster-2	0.79	0.98	Packed Friendly
streamcluster-4	0.86	0.98	Packed Friendly
streamcluster-8	0.91	0.85	Spread Friendly

Table 8.1: Benchmark Slowdown and Classification

From the classification, we can conclude that most PARSEC benchmarks turned out to be PackedFriendly, except the eight-threaded ones. This might occur because our worker consists of two sockets with five physical cores each. When executed under PackedFriendly configuration, the benchmark threads are placed on already allocated physical cores, within the same socket, introducing self-interference. This resulted in worse execution times than the SpreadFriendly configuration. This behaviour is proof that the profiling of

a workload is legitimate on a specific hardware configuration and it might differ when being executed on other hardware. The execution duration might depend on hardware-specific attributes such as cache line size, available NUMA nodes and physical cores, as well as other factors.

8.2 Single Node Experiment

8.2.1 Workload Collocation of Heterogeneously Labeled Applications

We conducted co-execution scenarios combining benchmarks of different suites and classification tags. While the Maestro scheduler would be able to make even more performant and cost-effective solutions on a multi-node environment, we gave emphasis on a single-node experimental setup. This approach was chosen because it represents the scenario which is most likely to exhibit the greatest performance degradation. All co-executions were deployed on node-4 of our cluster to ensure that even benchmarks with high CPU requirements could fit within a single host.

For each co-execution scenario, we evaluated four configurations: the default Kubernetes behavior (Vanilla), both benchmarks labeled as PackedFriendly, both benchmarks labeled as SpreadFriendly, and scheduling with Maestro. To assess performance, we measured the slowdown introduced relative to the isolated vanilla execution time of each application, calculated as:

$$S = \frac{T_{\text{co-executed}}}{T_{\text{isolated, vanilla}}}$$

We conducted multiple runs for each test and computed the average slowdown for every co-running benchmark across all four configurations. The results demonstrate that Maestro consistently outperforms the vanilla configuration, effectively reducing slowdown in most cases. The bar plots below illustrate co-executions of two benchmarks with different number of program threads (e.g., `stream-4` represents the STREAM benchmark with four threads) and the average slowdown they impose running alongside eachother, when running concurrently under different allocation strategies.

PackedFriendly vs. PackedFriendly

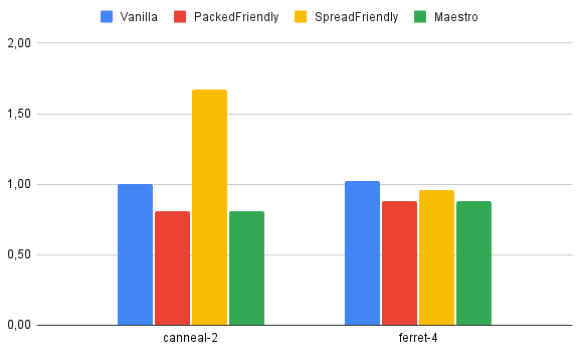


Figure 8.2.1: PackedFriendly vs. PackedFriendly (1)

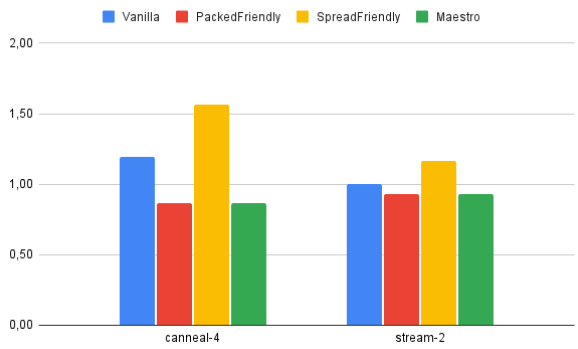


Figure 8.2.2: PackedFriendly vs. PackedFriendly (2)

Maestro outperforms vanilla Kubernetes, reducing co-execution slowdown by 16% and 10%. When PackedFriendly workloads are considered SpreadFriendly, their performance is degraded even more where the co-execution slowdown increases by 32% and 36% respectively. This highlights that misclassifying PackedFriendly workloads into a different category can result in resource contention, causing longer execution times and reduced performance.

SpreadFriendly vs. SpreadFriendly

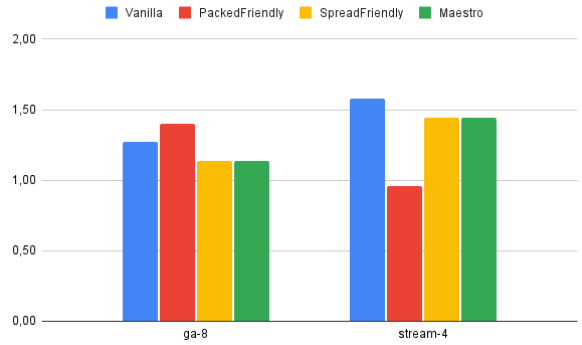


Figure 8.2.3: SpreadFriendly vs. SpreadFriendly (1)

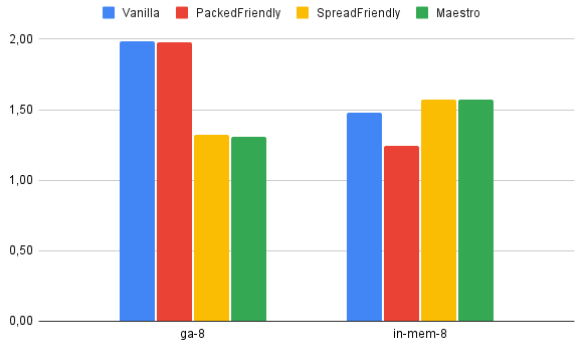


Figure 8.2.4: SpreadFriendly vs. SpreadFriendly (2)

Maestro outperforms vanilla Kubernetes, reducing average slowdown by 14% and 28%. Additionally, scenario 8.2.4 involves 16 program threads running on the same worker. In a multi-node cluster, the scheduler would distribute workloads across different nodes to enhance isolation, leading to ideal execution times that closely align with those observed in vanilla Kubernetes.

PackedFriendly vs. SpreadFriendly

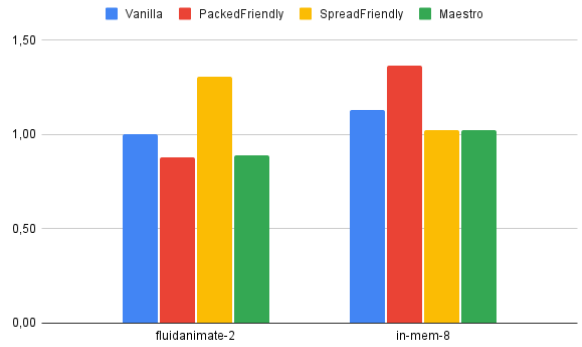


Figure 8.2.5: PackedFriendly vs. SpreadFriendly (1)

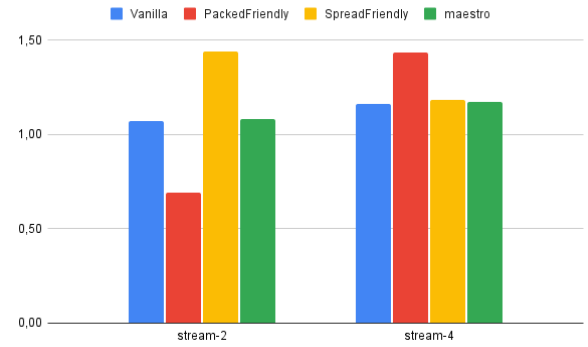


Figure 8.2.6: PackedFriendly vs. SpreadFriendly (2)

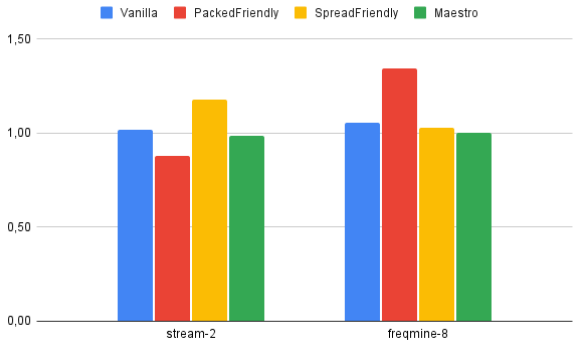


Figure 8.2.7: PackedFriendly vs. SpreadFriendly (3)

Figures 8.2.5 and 8.2.7 illustrate two scenarios comparing execution times between Maestro and the vanilla configuration that outperform vanilla by 11% and 5% respectively. While Figure 8.2.7 shows that stream-2 performs slightly better when both benchmarks are classified as PackedFriendly, freqmine-8 experiences a

significant performance hit, leading to increased overall slowdown. Despite this, Maestro consistently delivers the best execution times for both applications. Figure 8.2.6 shows a scenario where Maestro achieves the same execution times as the vanilla configuration, while PackedFriendly configuration achieves better results, because of stream-2 unexpectedly good performance under this configuration. This highlights the need for more sophisticated application profiling that is able to detect these anomalies.

PackedFriendly vs. Agnostic

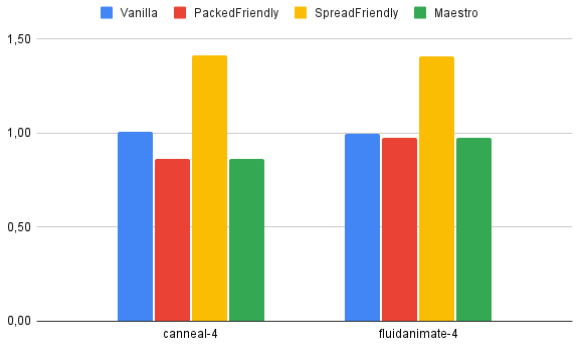


Figure 8.2.8: PackedFriendly vs. Agnostic (1)

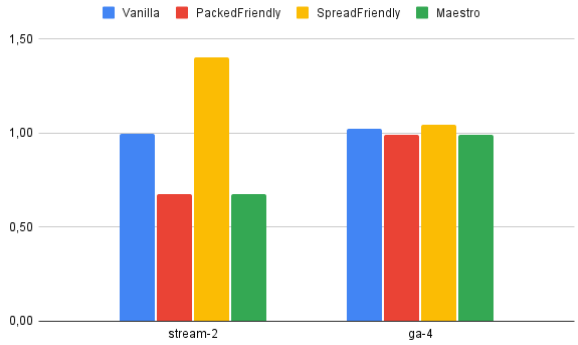


Figure 8.2.9: PackedFriendly vs. Agnostic (2)

After evaluating Agnostic applications in both PackedFriendly and SpreadFriendly configurations, we observed that they perform better when treated as PackedFriendly. Based on this observation, we configured Maestro to handle them as PackedFriendly, yielding promising results that outperformed vanilla Kubernetes by 8% and 18% in Figures 8.2.8 and 8.2.9, respectively.

SpreadFriendly vs. Agnostic

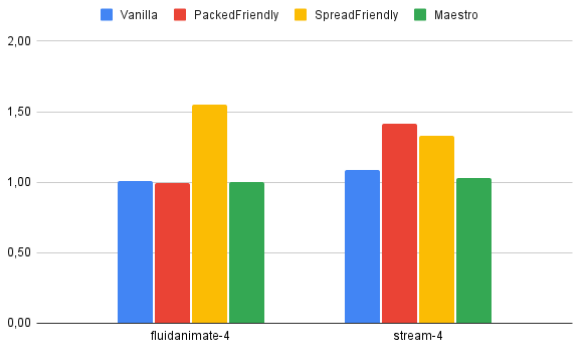


Figure 8.2.10: SpreadFriendly vs. Agnostic (1)

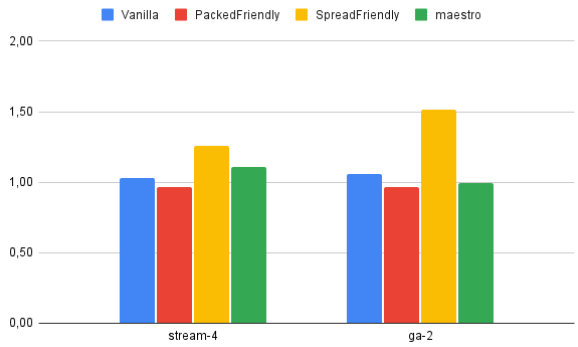


Figure 8.2.11: SpreadFriendly vs. Agnostic (2)

Figure 8.2.10 shows a scenario where Maestro slightly outperforms vanilla while PackedFriendly and SpreadFriendly configurations have a performance hit by 16% and 39% respectively. Scenario in Figure 8.2.11 shows that Maestro has slightly worse performance compared to PackedFriendly.

Agnostic vs. Agnostic

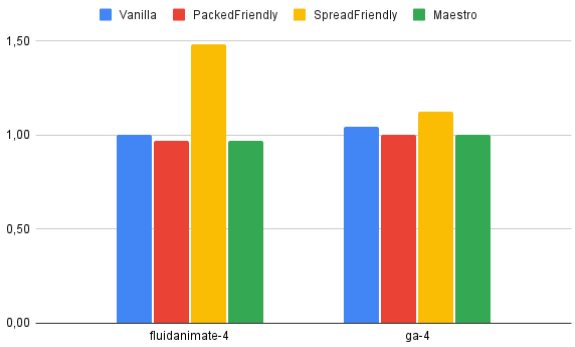


Figure 8.2.12: Agnostic vs. Agnostic

In this scenario Maestro would have the same behaviour as scheduling two PackedFriendly workloads. Both benchmarks seem to take a performance hit when considered SpreadFriendly, and Maestro barely manages to beat vanilla Kubernetes by 3%. This is an expected behavior of Agnostic workloads as they seem to perform almost the same as the vanilla configuration.

Aggregated Results

Figure 8.2.13 presents the aggregated results from all the scenarios discussed earlier, summarizing the average slowdown of both applications across the four configurations. The plot demonstrates that Maestro consistently achieves a lower average slowdown (calculated as the mean slowdown of both workloads) compared to the other three allocation strategies tested. Significant scenarios where Maestro greatly outperforms vanilla Kubernetes include slowdown reduction by 29%, 20%, 18% and 17%. We can conclude with confidence that Maestro can generally make correct CPU allocation decisions and can almost always reduce execution time by applying CPU pinning and isolation, as well as program thread distribution across sockets.

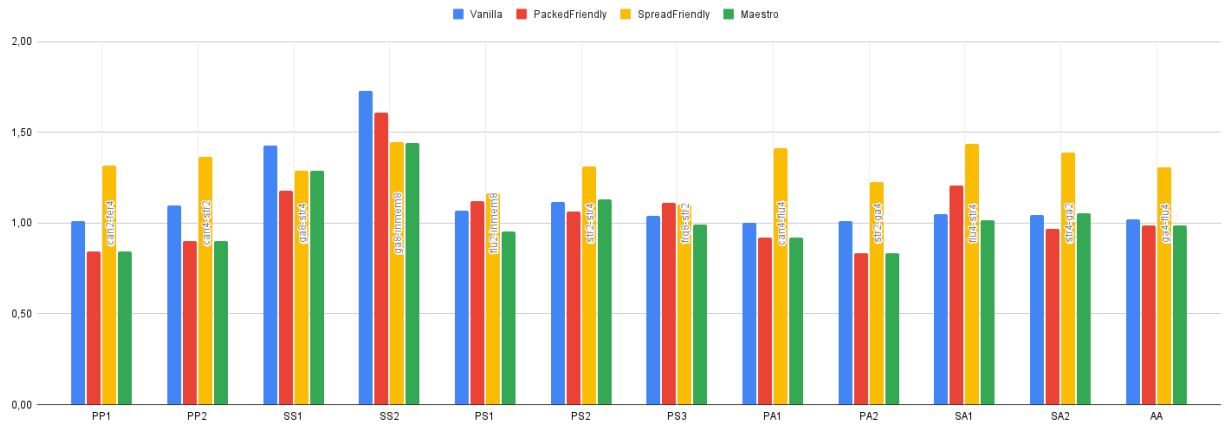


Figure 8.2.13: Aggregated Results

8.3 Multi Node Experiment

8.3.1 Node CPU Utilization Percentage

The Node CPU Utilization percentage is calculated using the formula:

$$\text{Node CPU Utilization} = \frac{\sum \text{Running Program Threads}}{\text{Node Logical CPUs}} \times 100\% \tag{8.3.1}$$

8.3.2 Additional WorkloadAware Scheduler Plugin Features

PhysicalCores

The `PhysicalCores` feature allows Maestro to schedule workloads onto physical cores rather than hyper-threaded logical cores. This capability reduces interference caused by hyper-threading, leading to improved performance for compute-bound workloads. Benchmarks categorized as PackedFriendly particularly benefit from this feature, as it minimizes contention and ensures stable execution times.

MemoryBoundExclusiveSockets

`MemoryBoundExclusiveSockets` feature optimizes scheduling for memory-intensive workloads by ensuring they are assigned exclusive access to sockets with sufficient memory bandwidth. This feature is critical for SpreadFriendly workloads, which perform better when distributed across nodes or sockets to minimize memory contention. Maestro leverages this feature to enhance performance for workloads that require high memory access rates.

BestEffortSharedCPUs

When scheduling applications unprioritized Quality of Service class, their threads could be packed onto a single logical CPU, leaving more isolated space for latency-critical workloads. When enabled, this feature packs BestEffort workloads when needed, increasing the overall CPU utilization percentage of the node.

Chapter 9

Conclusion and Future Work

9.1 Discussion

In this thesis, we describe in detail the phenomena of performance degradation and underutilization of resources on cloud systems. Every workload that can be deployed on multi-tenant clustered systems is susceptible to challenges related to hardware utilization and resource isolation, resulting in sub-optimal performance. Furthermore, in an effort to meet Service-Level Agreements, Cloud Providers often resort to hardware underutilization, which can result in significant environmental costs over time. Continuous research is being conducted in an attempt to come up with more efficient scheduling and resource allocation policies that consider advanced metrics beyond simple total utilization. Kubernetes, the most widely-adopted container orchestrator, lacks awareness of the specific characteristics of the executed workloads, preventing it from making allocation decisions tailored to each application. In a multi-tenant environment, some workloads might benefit from resource isolation and exclusive computational resources, whereas others might achieve better performance when distributed across multiple NUMA nodes. Additionally, certain workloads may face bottlenecks due to factors like I/O load, making the allocation of isolated resources inefficient and potentially wasteful. Achieving maximum performance for latency-critical workloads while minimizing node utilization percentage is key to optimizing performance and cost-efficiency in cloud environments. Numerous alternatives to Kubernetes' naive scheduling and resource allocation mechanisms have been proposed in the literature. These include offline profiling of workloads using machine learning techniques and real-time low-level system monitoring to enable more informed and effective allocation decisions.

In this work, we explore the developer tools the Kubernetes community provides, in an attempt to design a lightweight and extensible framework that enables future researchers and developers to experiment with fine-grained allocation policies. Leveraging Custom Resource Definitions and the Operator Pattern, along with other tools and languages, we introduce a declarative approach for specifying resource requirements for Pods running within a Kubernetes cluster. We collected representative benchmarks from different suites and applied our offline profiling algorithm to classify them in one of four categories. After that, we employed our custom scheduler and resource allocator to run them in a single node in order to evaluate their co-execution performance, by utilizing execution durations, as well as low-level system metrics such as memory bandwidth, instructions per cycle, LLC cache misses, and other relevant indicators. With a very simplistic profiling and allocation strategy, Maestro, our custom mechanism, consistently outperformed the vanilla Kubernetes scheduler and kubelet's default allocation strategy in nearly all tested scenarios, achieving speedups of up to 29%. The performance efficiency offered by our scheduler, combined with the flexibility of different scheduling policies and configurable options—such as disabling SMT and allocating full sockets for memory-intensive workloads—can lead to lower resource utilization while maintaining optimal performance.

9.2 Future Work

While several promising benchmark results were observed, the mechanism we implemented was an initial step towards the development of a comprehensive workload-aware scheduling and resource allocation mechanism.

Such software should be able to conduct both online and offline application profiling, enabling static and dynamic resource allocation strategies, effectively mitigating unpredictable interference phenomena. Workload characterization, a topic that is heavily being researched, can be achieved with various methods, and can become a lot more sophisticated by incorporating low-level system metrics and hardware performance counters. Profiling requires an additional pre-deployment step, during which workload characteristics are gathered by executing the application in a controlled lab environment and analyzing them using a well-trained neural network. Multiple reinforcement learning (RL) and deep learning (DL) algorithms have been proposed that might be suitable for optimizing cloud-native container orchestration in Kubernetes clusters.

We developed API extensions that enable our resource allocator to have a cluster-wide supervisory view of the nodes' CPU and memory resources, and the workloads that are bound to them, a feature that was not implemented in Kubernetes' core until now. These APIs could be the cornerstone of a more sophisticated resource allocator that do not only apply static resource allocation but can also modify the resources given to a specific Pod when performance degradation is detected. This could be achieved with sophisticated healthy-state machine learning models that detect interference on runtime and are able to make resource re-binding decisions. Online metrics extraction could be implemented using PCM metrics, as we have already described. Furthermore, research has to be made on how such software, that require low-level hardware access could be implemented in a cloud-native manner. The goal is to enable seamless deployment on any Kubernetes cloud provider without requiring the setup of external daemons outside the Kubernetes cluster.

Cloud Service Providers should prioritize the adoption of state-of-the-art resource management systems, not only to maximize financial efficiency but also to minimize the environmental impact caused by resource overutilization. Further research has to be conducted on how to improve node utilization in an effort to minimize aggressive installation of new hardware within datacenters, as well as the energy consumption they impose. Balancing the requested Quality of Service of the deployed workloads with efficient utilization is a complex topic in multi-tenant systems and requires great knowledge of low-level hardware concepts, such as cache locality and NUMA-aware computing. Fine-grained, workload-aware orchestration strategies must be employed to address the challenges of resource sharing during the co-existence of workloads on cloud systems, while promoting environmental sustainability.

Chapter 10

Bibliography

[1] Blagodurov, S. and Fedorova, A. "In search for contention-descriptive metrics in HPC cluster environment". en. In: *Proceedings of the 2nd ACM/SPEC International Conference on Performance engineering*. Karlsruhe Germany: ACM, Mar. 2011, pp. 457–462. ISBN: 978-1-4503-0519-8. DOI: 10.1145/1958746.1958815. URL: https://dl.acm.org/doi/10.1145/1958746.1958815 (visited on 06/29/2024).

[2] Cusack, G. et al. "Escra: Event-driven, Sub-second Container Resource Allocation". In: *2022 IEEE 42nd International Conference on Distributed Computing Systems (ICDCS)*. Bologna, Italy: IEEE, July 2022, pp. 313–324. ISBN: 978-1-66547-177-0. DOI: 10.1109/ICDCS54860.2022.00038. URL: https://ieeexplore.ieee.org/document/9912180/ (visited on 06/29/2024).

[3] *Docker*. URL: https://www.docker.com.

[4] *Electricity 2024*. Tech. rep. International Energy Agency, 2024. URL: https://www.iea.org/reports/electricity-2024.

[5] Fu, Y. et al. "Progress-based Container Scheduling for Short-lived Applications in a Kubernetes Cluster". In: *2019 IEEE International Conference on Big Data (Big Data)*. Los Angeles, CA, USA: IEEE, Dec. 2019, pp. 278–287. ISBN: 978-1-72810-858-2. DOI: 10.1109/BigData47090.2019.9006427. URL: https://ieeexplore.ieee.org/document/9006427/ (visited on 09/30/2024).

[6] Hutcheson, A. and Natoli, V. "Memory Bound vs . Compute Bound : A Quantitative Study of Cache and Memory Bandwidth in High Performance Applications". In: 2011. URL: https://api.semanticscholar.org/CorpusID:14832325.

[7] Kaur, K. et al. "KEIDS: Kubernetes-Based Energy and Interference Driven Scheduler for Industrial IoT in Edge-Cloud Ecosystem". In: *IEEE Internet of Things Journal* 7.5 (May 2020), pp. 4228–4237. ISSN: 2327-4662, 2372-2541. DOI: 10.1109/JIOT.2019.2939534. URL: https://ieeexplore.ieee.org/document/8825476/ (visited on 09/30/2024).

[8] *Koordinator*. URL: https://koordinator.sh.

[9] *Kubernetes*. URL: https://kubernetes.io.

[10] Li, D., Wei, Y., and Zeng, B. "A Dynamic I/O Sensing Scheduling Scheme in Kubernetes". en. In: *Proceedings of the 2020 4th International Conference on High Performance Compilation, Computing and Communications*. Guangzhou China: ACM, June 2020, pp. 14–19. ISBN: 978-1-4503-7691-4. DOI: 10.1145/3407947.3407950. URL: https://dl.acm.org/doi/10.1145/3407947.3407950 (visited on 06/29/2024).

[11] Liu, P. and Guitart, J. *Fine-Grained Scheduling for Containerized HPC Workloads in Kubernetes Clusters*. arXiv:2211.11487 [cs]. Nov. 2022. URL: http://arxiv.org/abs/2211.11487 (visited on 06/29/2024).

[12] Liu, P. et al. "Scanflow-K8s: Agent-based Framework for Autonomic Management and Supervision of ML Workflows in Kubernetes Clusters". In: *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. Taormina, Italy: IEEE, May 2022, pp. 376–385. ISBN: 978-1-66549-956-9. DOI: 10.1109/CCGrid54584.2022.00047. URL: https://ieeexplore.ieee.org/document/9826110/ (visited on 12/29/2024).

[13] Liu, Q. et al. *Jiagu: Optimizing Serverless Computing Resource Utilization with Harmonized Efficiency and Practicability*. arXiv:2403.00433 [cs]. Mar. 2024. DOI: 10.48550/arXiv.2403.00433. URL: http://arxiv.org/abs/2403.00433 (visited on 12/29/2024).

[14] Lo, D. et al. "Heracles: improving resource efficiency at scale". en. In: *ACM SIGARCH Computer Architecture News* 43.3S (Jan. 2016), pp. 450–462. ISSN: 0163-5964. DOI: 10.1145/2872887.2749475. URL: https://dl.acm.org/doi/10.1145/2872887.2749475 (visited on 02/01/2025).

[15] Mell, P. M. and Grance, T. *The NIST definition of cloud computing*. en. Tech. rep. NIST SP 800-145. Edition: 0. Gaithersburg, MD: National Institute of Standards and Technology, 2011, NIST SP 800–145. DOI: 10.6028/NIST.SP.800-145. URL: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-145.pdf (visited on 09/25/2024).

[16] Psomadakis, S. et al. "ACTiManager: An end-to-end interference-aware cloud resource manager". en. In: *Proceedings of the 20th International Middleware Conference Demos and Posters*. Davis CA USA: ACM, Dec. 2019, pp. 27–28. ISBN: 978-1-4503-7042-4. DOI: 10.1145/3366627.3368114. URL: https://dl.acm.org/doi/10.1145/3366627.3368114 (visited on 06/29/2024).

[17] Rodriguez, M. A. and Buyya, R. *Containers Orchestration with Cost-Efficient Autoscaling in Cloud Computing Environments*. arXiv:1812.00300 [cs]. Dec. 2018. DOI: 10.48550/arXiv.1812.00300. URL: http://arxiv.org/abs/1812.00300 (visited on 12/29/2024).

[18] Song, C. et al. *ChainsFormer: A Chain Latency-aware Resource Provisioning Approach for Microservices Cluster*. arXiv:2309.12592 [cs]. Oct. 2023. DOI: 10.48550/arXiv.2309.12592. URL: http://arxiv.org/abs/2309.12592 (visited on 12/29/2024).

[19] Tang, L., Mars, J., and Soffa, M. L. "Contentiousness vs. sensitivity: improving contention aware runtime systems on multicore architectures". en. In: *Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era*. San Jose California USA: ACM, June 2011, pp. 12–21. ISBN: 978-1-4503-0708-6. DOI: 10.1145/2000417.2000419. URL: https://dl.acm.org/doi/10.1145/2000417.2000419 (visited on 06/29/2024).

[20] *The State of Cloud Optimization 2024*. Tech. rep. Intel Granulate, 2024. URL: https://granulate.io/lp/state-of-cloud-optimization-2024/.

[21] Ungureanu, O.-M., Vlădeanu, C., and Kooij, R. "Kubernetes cluster optimization using hybrid shared-state scheduling framework". en. In: *Proceedings of the 3rd International Conference on Future Networks and Distributed Systems*. Paris France: ACM, July 2019, pp. 1–12. ISBN: 978-1-4503-7163-6. DOI: 10.1145/3341325.3341992. URL: https://dl.acm.org/doi/10.1145/3341325.3341992 (visited on 06/29/2024).

[22] *Volcano*. URL: https://volcano.sh/en/.

[23] Wang, Q. and Kan, J. *Practice of Fine-grained Cgroups Resources Scheduling in Kubernetes*. Nov. 2020.

[24] Yeung, G. et al. "Horus: Interference-Aware and Prediction-Based Scheduling in Deep Learning Systems". In: *IEEE Transactions on Parallel and Distributed Systems* 33.1 (Jan. 2022), pp. 88–100. ISSN: 1045-9219, 1558-2183, 2161-9883. DOI: 10.1109/TPDS.2021.3079202. URL: https://ieeexplore.ieee.org/document/9428512/ (visited on 09/30/2024).