Εθνικο Μετσοβιο Πολυτεχνειο
Σχολη Ηλεκτρολογων Μηχανικων Και Μηχανικων Υπολογιστων

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΥΠΟΛΟΓΙΣΤΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

Αποτελεσματική συνδρομολόγηση εφαρμογών σε συστοιχίες που ελέγχονται από τον Κυβερνήτη

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

Γεώργιου Λ. Κανελλόπουλου

Επιβλέπων: Γεώργιος Ι. Γκούμας
Επίκουρος Καθηγητής

Αθήνα, Ιανουάριος 2021

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΥΠΟΛΟΓΙΣΤΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ (CSLAB)

Αποτελεσματική συνδρομολόγηση εφαρμογών σε συστοιχίες που ελέγχονται από τον Κυβερνήτη

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

Γεώργιου Λ. Κανελλόπουλου

Επιβλέπων: Γεώργιος Ι. Γκούμας
Επίκουρος Καθηγητής

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 14η Ιανουαρίου 2021.

....	....	....
Γεώργιος Γκούμας	Νεκτάριος Κοζύρης	Διονύσιος Πνευματικάτος
Επίκουρος Καθηγητής	Καθηγητής	Καθηγητής

Αθήνα, Ιανουάριος 2021

....
Γεώργιος Λ. Κανελλόπουλος
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών

National Technical University of Athens
School of Electrical And Computer Engineering

DEPARTMENT OF COMPUTER SCIENCE
COMPUTING SYSTEMS LABORATORY

Efficient workload co-location on a Kubernetes cluster

DIPLOMA THESIS

by

Georgios Kanellopoulos

Supervisor: Georgios Goumas
Assistant Professor

Athens, January 2021

Σύνοψη

Τα συστήματα υπολογιστικού νέφους γίνονται όλο και πιο δημοφιλή για την φιλοξενία ποικίλων εφαρμογών, όπως διαδραστικές δικτυακές εφαρμογές, μηχανική μάθηση και υπολογιστική υψηλών επιδόσεων. Πολλοί οργανισμοί επιλέγουν να τρέχουν τις εφαρμογές τους στο νέφους, καθώς είναι πιο ευέλικτο, οικονομικό και ασφαλές. Υπάρχει πληθώρα παρόχων υπηρεσίων νέφους, από δημόσια νέφη όπως το Google Compute Engine, το Microsoft Azure και τα Amazon Web Services μέχρι ιδιωτικά κέντρα δεδομένων που ενορχηστρώνονται από συστήματα όπως το Mesos και ο Κυβερνήτης. Οι πάροχοι υπηρεσίων νέφους επωφελούνται από την ταυτόχρονη χρήση των πόρων τους από πολλαπλούς ενοίκους και την δημιουργία οικονομιών κλίμακος κατά την δημιουργία μεγάλων κέντρων δεδομένων.

Αν και οι πάροχοι υπηρεσιών νέφους πρέπει προβούν σε πολύ μεγάλες επενδύσεις για να κατασκευάσουν, να εξοπλίσουν και να λειτουργήσουν ένα κέντρο δεδομένων, οι διαθέσιμοι πόροι τους παρουσιάζουν χαμηλή χρησιμοποίηση. Ευαίσθητες εφαρμογές θέτουν αυστηρούς στόχους ως προς τον χρόνο απόκρισής τους και ο φόβος παραβίασης των συμφωνεθέντων στόχων οδηγεί τους διαχειριστές των κέντρων δεδομένων στην υπεράριθμη ανάθεση πόρων στις εφαρμογές αυτές, ειδικά όταν δρομολογούνται και μη ευαίσθητες εφαρμογές στα ίδια μηχανήματα. Στα τέλη της δεκαετία του 2000, η μέση χρήση των μηχανημάτων σε ιδιωτικά κέντρα δεδομένων κυμαινόταν μεταξύ 6% και 12%, με τα καλύτερα εξ αυτών να φτάνουν το 30%. Πιο πρόσφατα δεδομένα δείχνουν ότι οι χρησιμοποίηση των μηχανημάτων σε κέντρα δεδομένων της Google έχουν φτάσει το 60%. Πολλοί ερευνητές έχουν προτείνει ιδέες για την αύξηση χρησιμοποίησης των πόρων σε περιβάλλοντα νέφους. Κοινός τόπος των προτάσεων αυτών είναι η αντικατάσταση των στατικών δεσμεύσεων πόρων με δυναμική προσαρμογή ανάλογα με τις ανάγκες των εφαρμογών.

Η διπλωματική αυτή εργασία προτείνει ένα σύστημα διαχείρισης πόρων για συστάδες μηχανημάτων που ελέγχονται από τον Κυβερνήτη. Οι χρήστες των μηχανημάτων καθορίζουν στόχους επίδοσης, αντί να δεσμεύουν πόρους. Το προτεινόμενο σύστημα αναλαμβάνει να βρει τους ελάχιστους πόρους που απαιτεί μια ευαίσθητη εφαρμογή προκειμένου να ικανοποιήσει τους στόχους που έχουν τεθεί. Οι υπόλοιποι πόροι ανατίθενται σε μία μη ευαίσθητη εφαρμογή. Η προτεινόμενη υλοποίηση αξιοποιεί δεδομένα από το σύστημα παρακολούθησης Προμηθέας και δεν απαιτεί πρόσβαση σε μετρητές υλικού. Συγκρινόμενο με μια συντηρητική σταθερή δέσμευση πόρων, το προτεινόμενο σύστημα αυξάνει την χρησιμοποίηση των μηχανημάτων κατά 1,6 φορές ενώ οι παραβιάσεις των πόρων αυξάνονται από 4% σε 11%.

Λέξεις κλειδιά: υπολογιστική νέφους, διαχείριση πόρων, Κυβερνήτης, containers, κέντρο δεδομένων, χρησιμοποίηση

Abstract

Cloud computing has become increasingly popular with a diverse set of applications spanning the areas of web service applications, machine learning and high-performance computing. Many organizations choose to deploy their applications in the cloud, as it offers flexibility, high availability and cost efficiency. A wide variety of cloud providers is available, from public clouds like Google Compute Engine, Microsoft Azure and Amazon Web Services to private clouds orchestrated by frameworks like Openstack, Apache Mesos and Kubernetes. Cloud providers benefit from multi-tenancy, since their resources are shared by multiple users; moreover building large-scale datacenters across the globe could create economies of scale.

Although cloud providers invest very large amounts of money in building and operating datacenters, their resources are significantly underutilized. Latency-critical applications operate under strict latency constraints and the fear of violation leads cloud administrators to be conservative when scheduling batch jobs on the same servers. In the late 2000s, it was estimated that the average server utilization ranged between 6% and 12%, with the high-end reaching 30%. More recent data from 2019 show that server utilization in Google datacenter servers has reached 60%, thanks to increased resource utilization by batch jobs. A plethora of researchers from industry and academia have proposed ideas, so as to increase the resource utilization in cloud datacenters. Their efforts are mostly concentrated in moving from a reservation-centric to a performance-centric resource allocation approach, because reservations tend to be severely overestimated.

This thesis proposes a resource manager for container-based clusters managed by Kubernetes. The cluster users define performance targets in terms of latency instead of reserving resources. The resource manager is responsible to find the least amount of vCPUs the latency-critical workload requires to avoid latency target violations; the remaining vCPUs are allocated to a batch job. The resource manager leverages measurements from Prometheus monitoring system and does not require access to hardware counters. Compared to a conservative static resource allocation, the proposed resource manager increases the server utilization by 1,6x, while increasing the latency target violations from 4% to 11%.

Keywords: cloud computing, resource management, Kubernetes, containers, datacenter, utilization

Ευχαριστίες

Η παρούσα διπλωματική εργασία εκπονήθηκε στο Εργαστήριο Υπολογιστικών Συστημάτων. Αρχικά, θα ήθελα να ευχαριστήσω τον επίκουρο καθηγητή κ. Γεώργιο Γκούμα για την ανάθεση αυτού του πολύ ενδιαφέροντος θέματος, καθώς και για τις γνώσεις που μού μετέδωσε μέσα από τα μαθήματά του σχετικά με τα υπολογιστικά συστήματα και την παράλληλη επεξεργασία. Θα ήθελα επίσης να ευχαριστήσω τον υποψήφιο διδάκτορα Ιωάννη Παπαδάκη για την διαρκή και πολύτιμη βοήθεια του, χωρίς την οποία δεν θα ήταν δυνατή η ολοκλήρωση αυτής της εργασίας. Τέλος, θα ήθελα να εκφράσω την ευγνωμοσύνη μου προς την οικογένεια μου για την υποστήριξη που μου προσέφερε κατά τα φοιτητικά μου χρόνια.

Περιεχόμενα

Μέρος I

Αποτελεσματική συνδρομολόγηση εφαρμογών σε συστοιχίες που ελέγχονται από τον Κυβερνήτη

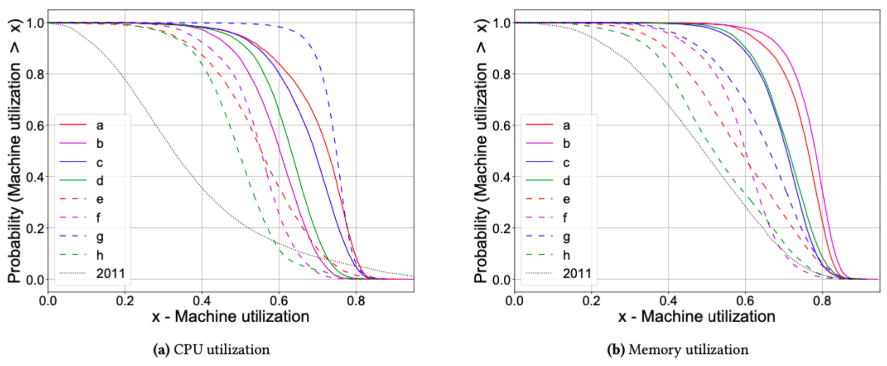
Κατάλογος σχημάτων

Κεφάλαιο 1

Εισαγωγή

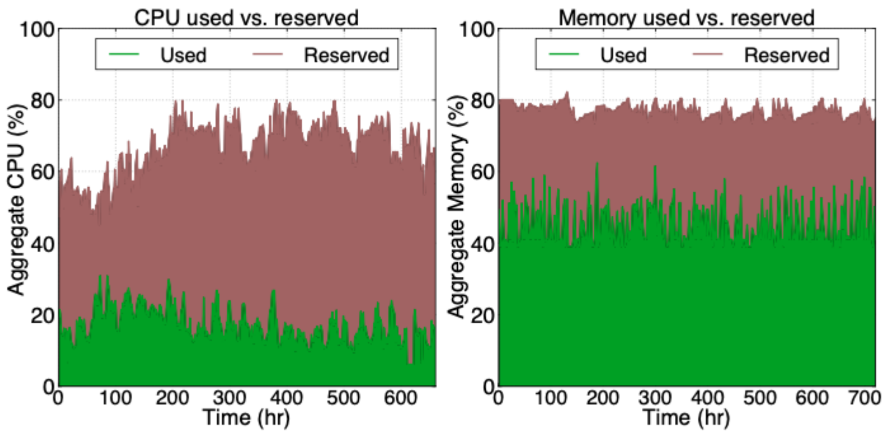
Το θέμα της διπλωματικής εργασίας είναι η αποτελεσματική συνδρομολόγηση φορτίων σε ένα cluster που ελέγχεται από τον Κυβερνήτη. Στο πλαίσιο της εργασίας αυτής, τα υπολογιστικά φορτία χωρίζονται σε δύο κατηγορίες: τα ευαίσθητα (latency-critical / LC) που έχουν αυστηρούς στόχους σχετικά με την καθυστέρηση και τα μη ευαίσθητα (best-effort / BE) που έχουν πολύ χαλαρούς ή και καθόλου στόχους για τον χρόνο εκτέλεσης τους. Τα ευαίσθητα φορτία είναι συνήθως διαδραστικές δικτυακές εφαρμογές (π.χ. ιστοσελίδα κρατήσεων για ξενοδοχεία), ενώ τα μη ευαίσθητα αφορούν ανάλυση μεγάλων δεδομένων και συστήματα συστάσεων.

Αφορμή για την ενασχόληση με το θέμα αποτελούν επιστημονικές δημοσιεύσεις που δείχνουν ότι η χρησιμοποίηση των πόρων στα κέντρα δεδομένων είναι χαμηλή, παρότι η κατασκευή, ο εξοπλισμός και η λειτουργία ενός κέντρου δεδομένων απαιτεί επενδύσεις εκατομμυρίων [3]. Στα τέλη της δεκαετίας του 2000, η χρησιμοποίηση πόρων σε ιδιωτικά κέντρα δεδομένων βρισκόταν μεταξύ 6% [7] και 12% [29]. Την ίδια περίοδο, η χρησιμοποίηση των επεξεργαστών σε εξυπηρετητές της Google που ελέγχοται από το σύστημα Borg ανερχόταν σε 25-35%. Δεδομένα του 2019 δείχνουν ότι το νούμερο αυτό έχει αυξηθεί στο 60% χάρη στην αυξημένη δρομολόγηση μη ευαίσθητων εφαρμογών [28]. Τα δεδομένα για τους εξυπηρετητές της Google αποτυπώνονται στο διάγραμμα 1.1. Το λογισμικό Borg πρόκειται για ένα ώριμο και καλά σχεδιασμένο σύστημα διαχείρισης πόρων και ως εκ τούτου οι εξυπηρετητές της Google εμφανίζουν υψηλότερη χρησιμοποίηση από τα περισσότερα κέντρα δεδομένων.



Σχήμα 1.1: Χρησιμοποίηση πόρων σε εξυπηρετητές της Google που ελέγχονται από το σύστημα Borg το 2019 και το 2011 [28]

Η χαμηλή χρησιμοποίηση πόρων οφείλεται πιθανώς στην δέσμευση πόρων για τις ευαίσθητες εφαρμογές. Οι εφαρμογές αυτές τρέχουν με αυστηρούς στόχους καθυστέρησης και ο φόβος πιθανής παραβίασης των στόχων αυτών σε περίπτωση μιας ξαφνικής αύξησης της εισερχόμενης κίνησης κάνει τους διαχειριστές να δεσμεύουν περισσότερους πόρους απ΄ όσους απαιτούνται πράγματι. Το σχήμα 1.2 δείχνει τους δεσμευμένους και τελικά χρησιμοποιηθέντες πόρους σε εξυπηρετητές της Twitter που ελέγχονται από το σύστημα Mesos και για διάστημα 30 ημερών. Η δέσμευση επεξεργαστικής ισχύος είναι 3 με 5 φορές υψηλότερη και η δέσμευση μνήμης 1,5 με 2 φορές μεγαλύτερη. Η δημοσίευση Quasar [11] αποδεικνύει ότι μόνο το 10% των δεσμεύσεων έχουν το σωστό μέγεθος, ενώ το 70% αυτών είναι



Σχήμα 1.2: Δεσμεύσεις και χρήση πόρων σε εξυπηρετητές της Twitter που ελέγχονται από το σύστημα Mesos [11]

μεγαλύτερες από όσο χρειάζεται.

Αν και υπάρχουν πολλές εργασίες που ασχολούνται με την βέλτιστη διαχείριση πόρων σε συστήματα υπολογιστικού νέφους μεγάλης κλίμακας, η πλειονότητα αυτών θεωρεί ότι οι εφαρμογές τρέχουν σε εικονικές μηχανές ή χωρίς κάποια μορφή εικονοποίησης. Τα τελευταία χρόνια έχουν εμφανιστεί δυναμικά τα containers, τα οποία είναι μια ελαφριά μορφή εικονοποίησης. Αντί να εικονοποιούν ολόκληρο το λειτουργικό σύστημα μαζί με τον πυρήνα του, εικονοποιούν μόνο τον χώρο χρήστη και τις άκρως απαραίτητες βιβλιοθήκες. Συνεπώς τα containers είναι πιο ελαφριά από τις εικονικές μηχανές, ξεκινούν πιο σύντομα και ο αριθμός αντιγράφων τους κλιμακώνει ευκολότερα ανάλογα με τις ανάγκες της εφαρμογής. Ο Κυβερνήτης [8] είναι ένα από τα πιο δημοφιλή συστήματα για την ενορχήστρωση containers, αναλαμβάνοντας την δρομολόγηση, διαχείριση και κλιμάκωσή τους. Λαμβάνοντας υπόψη τα παράπανω, η παρούσα εργασία διερευνά τρόπους αύξησης της χρήσης πόρων σε συστάδες υπολογιστών που ελέγχονται από τον Κυβερνήτη μέσω της αποτελεσματικής συνδρομολόγησης ευαίσθητων και μη ευαίσθητων φορτίων.

Κεφάλαιο 2

Σχετικές τεχνολογίες

2.1 Docker

Το Docker [12] είναι μία από τις πιο δημοφιλείς τεχνολογίες containers. Ένα container είναι μία μονάδα λογισμικού που περιέχει όλο τον κώδικα μιας εφαρμογής μαζί όλες τις εξαρτήσεις (π.χ. βιβλιοθήκες) και τις ρυθμίσεις παραμέτρων [9]. Τα containers είναι ελαφριά και απομονωμένα από το περιβάλλον στο οποίο τρέχουν· έτσι, το ίδιο container μπορεί να τρέξει σε ένα κέντρο δεδομένων, μία πλατφόρμα υπολογιστικού νέφους ή ακόμα και σε έναν προσωπικό υπολογιστή. Συγκρινόμενο με μία εικονική μηχανή (που εικονοποιεί όλο το λειτουργικό σύστημα), ένα container περιέχει μόνο τον χώρο χρήστη του λειτουργικού συστήματος, τις βιβλιοθήκες και τις υπηρεσίες που είναι απαραίτητες για την ερφαρμογή. Ως εκ τούτου, τα containers έχουν μικρότερες απαιτήσεις σε μνήμη και μικρότερο χρόνο εκκίνησης [10].

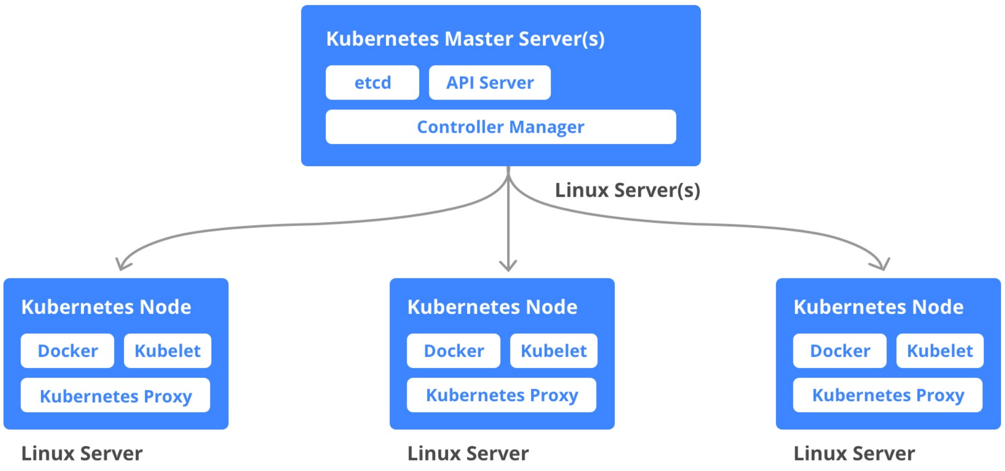
2.2 Kubernetes

Ο Κυβερνήτης (Kubernetes ή K8s) [8] είναι ένα ολοκληρωμένο λογισμικό διαχείρισης για containers, που αρχικά σχεδιάστηκε από την Google. Ο Κυβερνήτης αυτοματοποιεί την δημιουργία, την κλιμάκωση και την διαχείριση εφαρμογών που τρέχουν εντός containers σε διαφορετικά περιβάλλοντα, όπως εξυπηρετητές, εικονικές μηχανές, υπολογιστικά νέφη ή και υβριδικά συστήματα. Μπορεί να διαχειριστεί μεγάλες συστάδες μηχανημάτων που περιλαμβάνουν μέχρι 5000 μηχανήματα και 100 container ανά μηχάνημα.

Ο Κυβερνήτης αναπαριστά την κατάσταση της συστάδας μηχανημάτων μέσω οντοτήτων λογισμικού που ονομάζονται αντικείμενα. Κάθε αντικείμενο αποτελείται από το πεδίο `spec` όπου καταγράφεται η επιθυμητή κατάσταση του αντικειμένου, και το πεδίο `status` όπου η φαίνεται η τρέχουσα κατάσταση. Τα βασικότερα αντικείμενα είναι τα εξής:

- **Pod**: Το pod είναι η μικρότερη μονάδα που μπορεί να χειριστεί ο Κυβερνήτης και αποτελεί μία γενίκευση του container. Μέσω αυτής της γενίκευσης, ο Κυβερνήτης μπορεί να λειτουργεί ανεξάρτητα από τις τεχνολογίες των containers.
- **Service**: Πρόκειται για έναν γενικευμένο τρόπο ανάθεσης στατικής IP διεύθυνσης σε μία εφαρμογή που τρέχει σε ένα ή περισσότερα containers.
- **ReplicaSet**: Ένα ReplicaSet διαχειρίζεται πολλαπλά αντίγραφα του ίδιου pod που τρέχουν σε μία συστάδα μηχανημάτων. Στόχος του αντικειμένου αυτού είναι η διατήρηση ενός σταθερού αριθμού υγειών containers.
- **Deployment**: Το Deployment είναι ένα αντικείμενο που διαχειρίζεται ReplicaSets, προκειμένου να ελέγξει την δημιουργία, ενημέρωση και καταστροφή pods.

Τα μηχανήματα της συστάδας που διαχειρίζεται ο Κυβερνήτης χωρίζονται σε δύο κατηγορίες: τους εργάτες και τους αφέντες. Τα μηχανήματα - εργάτες φιλοξενούν τα containers μέσα στα οποία τρέχουν οι εφαρμογές και πρέπει να έχουν εγκατεστημένο ένα περιβάλλον εκτέλεσης container (π.χ. Docker ή containerd), το kubelet που δημιουργεί τα νέα containers και φροντίζει να τρέχουν σωστά, και το kube-proxy που προωθεί τα δικτυακά αιτήματα (προερχόμενα εντός ή εκτός της συστάδας) στις εφαρμογές. Τα μηχανήματα - αφέντες φιλοξενούν το επίπεδο ελέγχου (control plane) και προσφέρουν την διεπαφή για την δημιουργία και την διαχείριση του κύκλου ζωής των pods. Κάθε συστάδα έχει τουλάχιστον ένα αφέντη, και μεγαλύτερο πλήθος αφεντών εγγυάται ανοχή σε σφάλματα.



Σχήμα 2.1: Αρχιτεκτονική και συστατικά μιας συστάδας που ελέγχεται από τον Κυβερνήτη [21]

2.3 Prometheus

Ο Προμηθέας (Prometheus [26]) είναι ένα σύστημα ελέγχου και ειδοποίησης που χρησιμοποιείται συχνά σε περιβάλλοντα με containers και μικροϋπηρεσίες. Η δημοφιλία του οφείλεται στην ικανότητα ανάκτησης μετρήσεων τόσο σε επίπεδο υλικού όσο και σε επίπεδο εφαρμογής. Ο Προμηθέας παρακολουθεί τα σφάλματα κατά την εκτέλεση του λογισμικού, την καθυστέρηση των εφαρμογών και την χρησιμοποίηση των διαθέσιμων πόρων, καθιστώντας ευκολότερη την διαχείριση συστάδων μεγάλης κλίμακας που εκτελούν μεγάλο αριθμό εφαρμογών ταυτόχρονα.

Οι μετρήσεις που συλλέγει ο Προμηθέας αποθηκεύονται σε μία βάση δεδομένων με την μορφή χρονοσειρών. Μια χρονοσειρά είναι μια ακολουθία χρονοσημασμένων δεδομένων που διαχωρίζονται με βάση το όνομα της μετρικής και τις ετικέτες που φέρουν. Οι χρονοσειρές απαρτίζονται από δομές δεδομένων που περιέχουν μία τιμή ακρίβειας 64 bit και μια χρονοσφραγίδα ακρίβειας χιλιοστού του δευτερολέπτου. Τα δεδομένα μιας χρονοσειράς έχουν υψηλό αριθμό διαστάσεων και αυτό το χαρακτηριστικό οφείλεται στις ετικέτες. Για παράδειγμα μια χρονοσειρά περιέχει όλα τα HTTP αιτήματα· μία διάσταση δημιουργείται από την ετικέτα της μεθόδου (π.χ. POST, GET) και μία δεύτερη διάσταση από την ετικέτα του κωδικού απάντησης (π.χ. 200 - OK, 404 - Not Found).

Κεφάλαιο 3

Σχετικές εργασίες

Στο κεφάλαιο αυτό παρουσιάζονται σύντομα επιλεγμένες εργασίες που πραγματεύονται την διαχείριση πόρων σε συστήματα νέφους μεγάλης κλίμακας. Η πρώτη εργασία που θα παρουσιαστεί είναι το σύστημα Heracles [22], ένα δυναμικό σύστημα πραγματικού χρόνου που αποσκοπεί στην καλύτερη διαχείριση πόρων επιτρέποντας την συνδρομολόγηση ενός ευαίσθητου φορτίου με οποιοδήποτε άλλο μη ευαίσθητο. Στην διάρκεια του χρόνου εκτέλεσης, το σύστημα Heracles διαχειρίζεται δυναμικά πολλαπλούς μηχανισμούς απομόνωσης στο υλικό και στο λογισμικό, όπως οι επεξεργαστικοί πυρήνες, η κύρια μνήμη, το τελευταίο επίπεδο κρυφής μνήμης και το εύρος ζώνης του δικτύου. Με αυτόν τον τρόπο διασφαλίζει ότι αποδίδονται οι αναγκαίοι πόροι στην ευαίσθητη εφαρμογή και ότι αυτή δεν παραβιάζει τον στόχο καθυστέρησης. Ταυτόχρονα μεγιστοποιούνται οι πόροι που μπορεί να χρησιμοποιήσει η μη ευαίσθητη εφορμογή. Το σύστημα αξιολογήθηκε πειραματικά με εφαρμογές από την Google και διαπιστώθηκε ότι μπορεί να επιτύχει χρησιμοποίηση των εξυπηρετητών σε επίπεδο 90%, αποτρέποντας ταυτόχρονα την παραβίαση των στόχων καθυστέρησης.

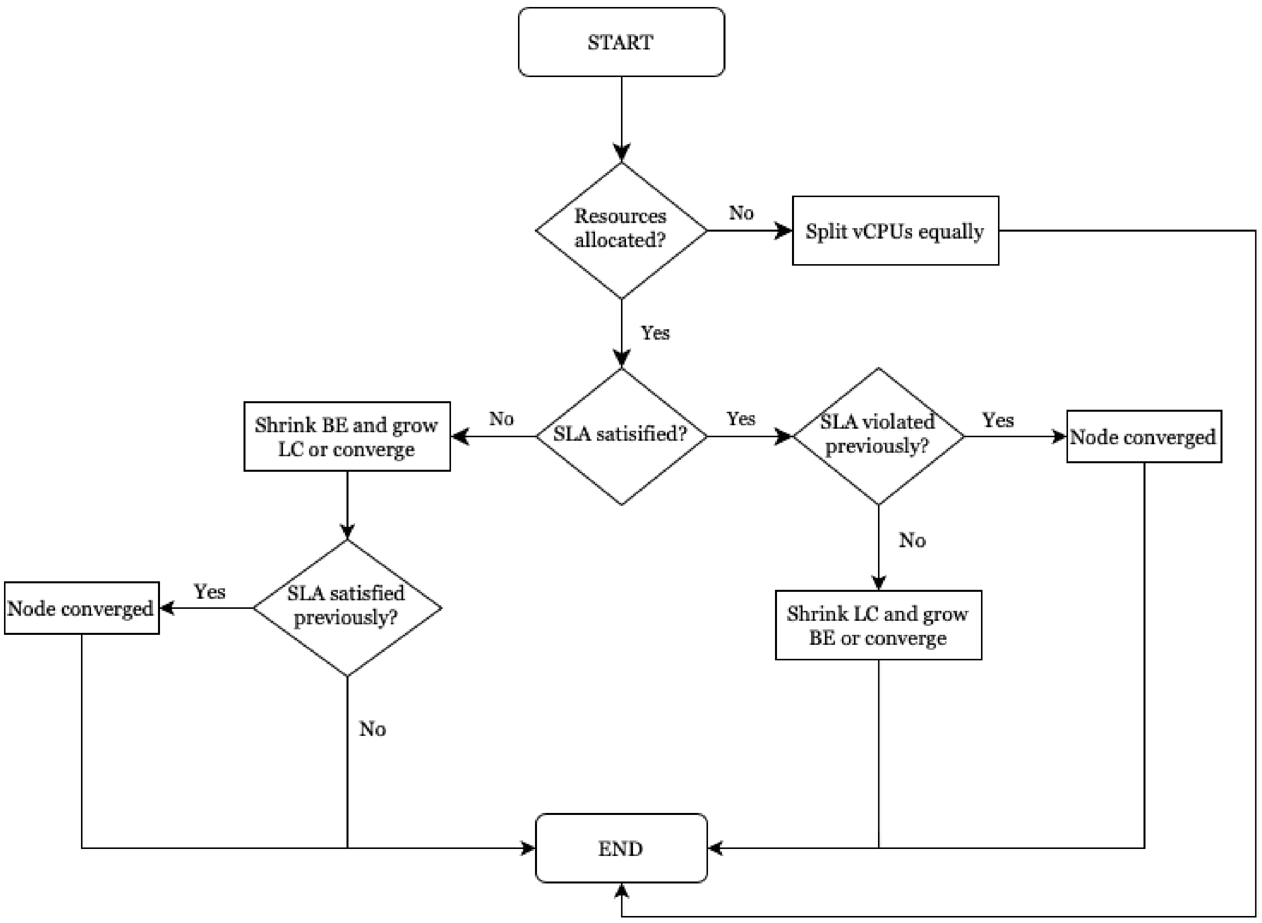
Μια παρόμοια προσέγγιση ακολουθεί και το σύστημα Proctor. Το σύστημα αυτό συλλέγει χρονοσειρές δεδομένων από τους μετρητές υλικού κάθε εξυπηρετητή σχετικά με 4 διαφορετικούς πόρους: δίκτυο, είσοδος/έξοδος, κρυφή μνήμη και επεξεργαστής. Κατά τον χρόνο εκτέλεσης παρακολουθείται η καθυστέρηση των εφαρμογών και χρησιμοποιούνται ένας αλγόριθμος μείωσης θορύβου και ένας αλγόριθμος εντοπισμού απότομων μεταβολών προκειμένου να εντοπιστεί η μειωμένη επίδοση κάποιας εφορμογής. Αφού εντοπιστεί κάποια μειωμένη επίδοση, το σύστημα Proctor συλλέγει δεδομένα από τους μετρητές υλικού του εξυπηρετητή και τα αναλύει στατιστικά προκειμένου να εντοπίσει ποιος πόρος έχει κορεστεί και ποια εφαρμογή ευθύνεται για τον κορεσμό αυτό. Κατόπιν μετακινεί ή απομονώνει την "ενοχλητική" εφαρμογή. Έτσι βελτιώνει την επίδοση των εφαρμογής κατά 2 φορές μεσοσταθμικά.

Το σύστημα Quasar [11] ακολουθεί μία διαφορετική προσέγγιση, καθώς αναλύει το προφίλ των εφαρμογών προτού τις δρομολογήσει. Πιο αναλυτικά, το σύστημα αυτό συλλέγει δεδομένα για τις νεοεισερχόμενες εφαρμογές ενώ αυτές τρέχουν απομονωνομένες. Αξιοποιεί κατόπιν τεχνικές μηχανικής μάθησης προκειμένου να εκτιμήσει την επίδραση της πόσοτητας και του είδους των πόρων, της παρενόχλησης από παρακείμενα φορτία και του αριθμού αντιγράφων της εφαρμογής στην επίδοσή της. Αξιοποιώντας τα αποτελέσματα της πρόβλεψης, αποφασίζει πόσοι και ποιοι ακριβώς πόροι θα δοθούν στο νεοεισερχόμενο φορτίο. Σε περίπτωση εσφαλμένης απόφασης, το σύστημα Quasar τρέχει πάλι τον αλγόριθμο μηχανικής μάθησης και αλλάζει τους πόρους που έχουν ανατεθεί στην εφαρμογή. Με αυτήν την μέθοδο πετυχαίνει να αυξήσει την χρησιμοποίση των μηχανημάτων κατά 47%, ενώ σέβεται τους στόχους καθυστέρησης.

Το σύστημα Seer ασχολείται με την πρόβλεψη καθυστέρησης σε γράφους μικροϋπηρεσίων, αξιοποιώντας μεγάλους όγκους δεδομένων από καταγραφές και εκπαιδεύοντας βαθιά νευρωνικά δίκτυα. Το σύστημα καταγράφει τις ουρές που δημιουργούνται στην είσοδο κάθε υπηρεσίας παρακολουθώντας τα πακέτα που φτάνουν στην κάρτα δικτύου. Τα δεδομένα για τα μεγέθη των ουρών τροφοδοτούν το νευρωνικό δίκτυο, το οποίο εκτιμά για κάθε υπηρεσία την πιθανότητα να παραβιαστεί ο στόχος καθυστέρησης της εφαρμογής εξαιτίας της υπηρεσίας αυτής. Όταν προβλεφθεί αυξημένη καθυστέρηση, το σύστημα Seer λαμβάνει μέτρα, όπως η εξασφάλιση αποκλειστικής πρόσβασης σε κομμάτι της κρυφής μνήμης και του εύρους ζώνης του δικτύου για την πιεσμένη υπηρεσία. Η πειραματική αποτίμηση δείχνει ότι το σύστημα Seer προβλέπει το 91% των περιπτώσεων μειωμένης επίδοσης και αποτρέπει το 84% αυτών.

Κεφάλαιο 4

Προτεινόμενο σύστημα διαχείρισης πόρων



Σχήμα 4.1: Διάγραμμα ροής του προτεινόμενου συστήματος διαχείρισης πόρων

Η εργασία αυτή προτείνει ένα δυναμικό σύστημα για την ανάθεση vCPU σε μία ευαίσθητη και μία μη ευαίσθητη εφαρμογή που τρέχουν ταυτόχρονα στον ίδιο κόμβο ενός cluster ελεγχόμενο από τον Κυβερνήτη. Η προτεινόμενη υλοποίηση χρησιμοποιεί το API του Κυβερνήτη και δεν απαιτεί μετρήσεις από το υλικό. Απαιτεί όμως οι ευαίσθητες εφαρμογές να δηλώνουν την καθυστέρηση απόκρισης στο σύστημα Prometheus, ώστε να γίνεται γνωστό αν ικανοποιούνται οι στόχοι που έχουν τεθεί για κάθε εφαρμογή. Επίσης απαιτεί οι εφαρμογές να ικανοποιούν τους παρακάτω περιορισμούς:

- Τα ευαίσθητα Deployment πρέπει να έχουν την ετικέτα “class = latency-critical” και τα μη ευαίσθητα την ετικέτα "class = best-effort", καθώς Deployments χωρίς ετικέτα δεν επηρεάζονται από το σύστημα διαχείρισης πόρων.

- Πρέπει να τρέχει το πολύ ένα ευαίσθητο και το πολύ ένα μη ευαίσθητο Deployment σε κάθε μηχάνημα.
- Δεν πρέπει να υπάρχουν δύο container με το ίδιο όνομα.
- Κάθε Deployment πρέπει να έχει το πολύ ένα αντίγραφο. Για να δημιουργηθούν δύο αντίγραφα του ίδιου Pod, πρέπει να οριστούν δύο διαφορετικά Deployments που συνδέονται με ένα Service.

Αρχικά, το σύστημα συνδέεται με το API του Κυβερνήτη και συλλέγει δεδομένα σχετικά με το cluster. Πιο αναλυτικά, τα δεδομένα που συλλέγει και αποθηκεύει περιλαμβάνουν το όνομα και την IP διεύθυνση των μηχανημάτων, τα Pods μαζί με την κατάσταση, τις ετικέτες και τους πόρους που δεσμεύουν, και μία λίστα με τα Deployments και τα Pods που ελέγχει κάθε Deployment.

Στην συνέχεια, το σύστημα διαχείρισης πόρων εξετάζει κάθε μηχάνημα ξεχωριστά και υλοποιεί τον αλγόριθμο που φαίνεται στο διάγραμμα ροής του σχήματος 4.1. Αν σε ένα μηχάνημα δεν τρέχει καμία εφαρμογή ή αν τρέχει μόνο μία, το σύστημα δεν κάνει κάποια αλλαγή. Στην περίπτωση που τρέχουν ακριβώς δύο φορτία που είναι καινούρια και δεν έχουν περιορισμούς σχετικά με την χρήση πόρων, ο αλγόριθμος μοιράζει εξίσου τις διαθέσιμες vCPUs.

Διαφορετικά, το σύστημα συνδέεται με το API του Prometheus και συλλέγει δεδομένα για την καθυστέρηση της ευαίσθητης εφαρμογής. Το API του Prometheus δίνει δεδομένα σχετικά με το 99% ποσοστημόριο καθυστέρησης της ευαίσθητης εφαρμογής τα τελευταία 30 δευτερόλεπτα. Το παράθυρο των 30 δευτερολέπτων είναι επαρκές, ώστε οι μετρήσεις να έχουν στατιστικά νόημα. Τα επιθυμητά δεδομένα συλλέγονται από το Prometheus μέσω του παρακάτω ερωτήματος:

```
histogram_quantile(0.99, rate(flask_http_request_duration_seconds_bucket{method="POST",status
    ="200"}[30s])
```

Αν ο στόχος καθυστέρησης της ευαίσθητης εφαρμογής ικανοποιείται, το σύστημα παίρνει μία vCPU από την ευαίσθητη εφαρμογή και την αποδίδει στην μη ευαίσθητη. Αν αντιθέτως δεν ικανοποιείται ο στόχος καθυστέρησης, το σύστημα αποδίδει μία επιπλεόν vCPU στην ευαίσθητη εφαρμογή. Προκειμένου να μην ταλαντώνεται το σύστημα μεταξύ γειτονικών καταστάσεων, καταγράφονται οι κόμβοι μόλις φτάσουν στο σημείο ισορροπίας. Το σημείο ισορροπίας εντοπίζεται όταν ο στόχος καθυστέρησης αλλάξει φάση, δηλαδή ικανοποιηθεί ενώ στην προηγούμενη κατάσταση παραβιαζόταν ή παραβιαστεί ενώ στην προηγούμενη κατάσταση ικανοποιούταν (στην περίπτωση αυτή επιστρέφει πρώτα ο κόμβος στην προηγούμενη κατάσταση δίνοντας μία vCPU στην ευαίσθητη εφαρμογή). Ο αλγόριθμος φτάνει επίσης σε σημείο ισορροπίας όταν ικανοποιείται ο στόχος και έχει μείνει μόνο μία vCPU στο ευαίσθητο φορτίο ή όταν ο στόχος δεν ικανοποιείται αλλά έχει μείνει μόνο μία vCPU στο μη ευαίσθητο φορτίο.

Κεφάλαιο 5

Πειραματική αξιολόγηση

Στο πλαίσιο της εργασίας αυτής γίνεται πειραματική αξιολόγηση διαφόρων σεναρίων συνδρομολόγησης ευαίσθητων και μη ευαίσθητων εφαρμογών σε cluster που ελέγχονται από τον Κυβερνήτη. Για τον σκοπό αυτό δημιουργείται ένα cluster αποτελούμενο από 1 κόμβο-αφέντη και 3 κόμβους-εργάτες. Κάθε εργάτης έχει 6 vCPUs διαθέσιμους και 8 GB μνήμης. Επίσης δημιουργείται ένα μικρότερο μηχάνημα με 3 vCPUs που δεν ανήκει στο cluster και δημιουργεί την εισερχόμενη κίνηση. Όλοι οι κόμβοι είναι εικονικά μηχανήματα που τρέχουν στον ίδιο φυσικό server.

5.1 Μετροπρογράμματα

Ως ευαίσθητο φορτίο σχεδιάστηκε και υλοποιήθηκε μία εφαρμογή που αποθηκεύει και ανακτά δεδομένα από μία βάση δεδομένων. Σκοπός της εφαρμογής είναι η στατιστική ανάλυση και η παρακολούθηση της καταναλωτικής συμπεριφοράς των πελατών μίας αλυσίδας supermarket. Η βάση δεδομένων αποθηκεύει τους πελάτες, τις συναλλαγές και τα προϊόντα του supermarket και έχει υλοποιηθεί με την MySQL [23]. Η βάση έχει 4 πίνακες (Product, Store, Customer, Transaction) μαζί με τα χαρακτηριστικά τους, όπως φαίνεται στο διάγραμμα σχέσεων-οντοτήτων (σχήμα 5.1). Τα δεδομένα της βάσης παρουσιάζονται μέσω μιας διαδικτυακής εφαρμογής που σχεδιάστηκε με το εργαλείο Flask.

Για την αξιολόγηση του προτεινόμενου συστήματος διαχείρισης πόρων, επιλέχθηκαν 3 διαφορετικά ερωτήματα προς την βάση δεδομένων:

- Το ”γρήγορο” ερώτημα εκτελεί ένα απλό select:

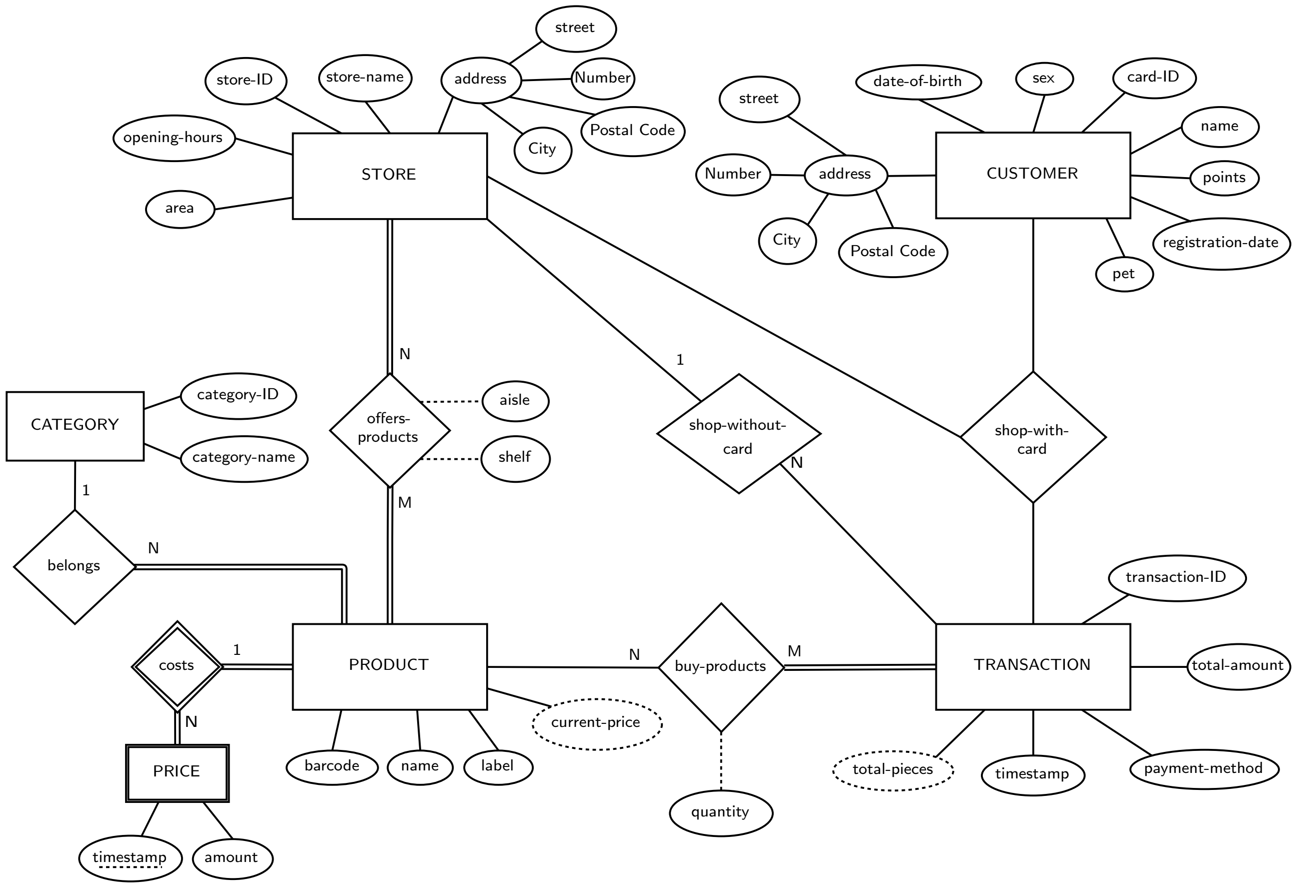
```sql
SELECT DATE(start_date), DATE(end_date), amount
FROM Price
WHERE barcode = <selected barcode>
```

- Το ”μεσαίο” ερώτημα είναι πιο περίπλοκο και περιλαμβάνει ένα καρτεσιανό γινόμενο, μία εμφωλευμένη επιλογή και μία ταξινόμηση:

```sql
SELECT P.barcode, P.product_name, sum(B.quantity) AS total_quantity
FROM buy_products AS B
INNER JOIN Product AS P ON B.barcode = P.barcode
AND B.transaction_id IN
(SELECT transaction_id FROM Transaction WHERE card_id = <selected_card>)
GROUP BY P.barcode
ORDER BY total_quantity DESC
LIMIT 10
```

- Το ”αργό” ερώτημα εκτελεί το καρτεσιανό γινόμενο δύο πινάκων, καθένας από τους οποίους προκύπτει από μία φυσική ένωση:

```sql
WITH buy_products_names(barcode, name, transaction_id) AS
( SELECT P.barcode, P.product_name, B.transaction_id
  FROM buy_products AS B
  NATURAL JOIN Product AS P )
SELECT B1.barcode, B1.name, B2.barcode, B2.name, COUNT(*) AS pair_freq
FROM buy_products_names AS B1, buy_products_names AS B2
```



Σχήμα 5.1: Διάγραμμα σχέσεων - οντοτήτων της βάσης δεδομένων

```
WHERE B1.transaction_id = B2.transaction_id and B1.barcode < B2.barcode
GROUP BY B1.barcode, B2.barcode
ORDER BY pair_freq DESC
LIMIT 10
```

Ως μη ευαίσθητα φορτία επιλέχθηκαν οι εφαρμογές fluidanimate, streamcluster, swaptions από την βιβλιοθήκη PARSEC [27]. Οι εφαρμογές της βιβλιοθήκης PARSEC χρησιμοποιούν πολλαπλά νήματα λογισμικού και ως εκ τούτου μπορούν να τρέχουν σε πολλαπλούς επεξεργαστικούς πυρήνες ταυτόχρονα. Τα προγράμματα προέρχονται από διάφορους επιστημονικούς τομείς και όχι μόνο από την υπολογιστική υψηλών επιδόσεων. Πιο συγκεκριμένα, η εφαρμογή fluidanimate απεικονίζει την δυναμική ενός ρευστού βάσει της υπολογιστικής τεχνικής λείου σωματιδίου. Το πρόγραμμα streamcluster είναι ένας αλγόριθμος κατηγοριοποίησης για ακολουθίες δεδομένων, και το swaptions εκτελεί μία προσομοίωση Monte Carlo για την αποτίμηση δικαιωμάτων προαίρεσης επί συμφωνιών ανταλλαγής (swaptions).

5.2 Πειράματα

Προς αξιολόγηση του συστήματος διαχείρισης πόρων, εκτελέστηκε ένας αριθμός πειραμάτων στο cluster που ελέγχεται από τον Κυβερνήτη χρησιμοποιώντας τα μετροπρογράμματα που αναλύθηκαν προηγουμενως. Πραγματοποιήθηκαν τα εξής πειράματα:

- **Πείραμα βάσης (alone)**: Το πρώτο πείραμα που πραγματοποιήθηκε ήταν η εκτέλεση της ευαίσθητης και των μη ευαίσθητων εφαρμογών μόνων τους σε έναν κόμβο ώστε να διαπιστωθεί ο ιδανικός χρόνος εκτελεσής τους.

- **Ελεύθερο πείραμα (colocated)**: Στο επόμενο πείραμα η ευαίσθητη εφαρμογή δρομολογήθηκε μαζί με καθεμία από τις εφαρμογές PARSEC στον ίδιο κόμβο και αφέθηκαν να τρέχουν χωρίς περιορισμό στην χρήση πόρων.

- **Στατικό πείραμα (static)**: Στο πείραμα αυτό εφαρμόστηκε μία στατική και συντηρητική πολιτική ανάθεσης πόρων. Πιο αναλυτικά, περιορίσθηκαν τα μη ευαίσθητα φορτία σε μία μόνο vCPU και δόθηκαν οι υπόλοιπες 5 στην ευαίσθητη εφαρμογή.
- **Δυναμικό πείραμα (dynamic/RM)**: Στο δυναμικό πείραμα αποτιμήθηκε στην πράξη η επίδοση του προτεινόμενου συστήματος διαχείρισης πόρων. Αρχικά οι vCPU μοιράζονται εξίσου μεταξύ των εφαρμογών. Σε κάθε επανάληψη δίνεται στην ευαίσθητη εφαρμογή μία επιπλέον vCPU, μέχρις ότου να ικανοποιηθεί ο στόχος καθυστέρησης ή να μην υπάρχουν άλλες διαθέσιμες vCPU. Αν ο στόχος καθυστέρησης ικανοποιείται από την αρχή, δίνονται παραπάνω vCPU στην μη ευαίσθητη εφαρμογή.

Η ευαίσθητη εφαρμογή εξυπηρετεί αιτήματα που προέρχονται από εξωτερικούς χρήστες. Στο πλαίσιο της πειραματικής αξιολόγησης, τα αιτήματα δημιουργούνται από ένα εικονικό μηχάνημα που δεν ανήκει στο cluster. Αξιολογούνται τρεις διαφορετικές υποθέσεις για την κίνηση προς την βάση δεδομένων: χαμηλή κίνηση (100 ερωτήματα το λεπτό), μεσαία κίνηση (250 ερωτήματα το λεπτό) και υψηλή κίνηση (400 ερωτήματα το λεπτό). Συνολικά, λοιπόν, προκύπτουν 9 σενάρια για κάθε πείραμα, που προκύπτουν από τους δυνατούς συνδυασμούς των 3 PARSEC προγραμμάτων με τις 3 υποθέσεις για την κίνηση.

Για την ευαίσθητη εφαρμογή τίθενται στόχοι καθυστέρησης, οι οποίοι αφορούν τον χρόνο εκτέλεσης των ερωτημάτων στην βάση δεδομένων. Οι στόχοι τίθενται βάσει του 99% ποσοστημορίου της καθυστέρησης για κάθε ερώτημα στο πείραμα βάσης (όπου η ευαίσθητη εφαρμογή τρέχει μόνη της στο cluster). Η καθυστέρηση αυτή πολλαπλασιάζεται με έναν συντελεστή, ως εξής:

- 1,50 αν η κίνηση είναι χαμηλή (100 ερωτήματα το λεπτό)
- 1,75 αν η κίνηση είναι μέτρια (250 ερωτήματα το λεπτό)
- 2,50 αν η κίνηση είναι υψηλή (400 ερωτήματα το λεπτό)

	100 RPM	250 RPM	400 RPM
γρήγορο	0,075	0,090	0,100
μεσαίο	0,40	0,45	1,00
αργό	3,00	3,50	6,00

Πίνακας 5.1: Στόχοι καθυστέρησης σε δευτερόλεπτα για κάθε συνδυασμό κίνησης και ερωτήματος στην βάση δεδομένων

Σε κάθε σενάριο συλλέγονται μετρήσεις για την αξιολόγηση της επίδοσης των εφαρμογών. Για κάθε ερώτημα που γίνεται στην βάση δεδομένων, υπολογίζεται το ιστόγραμμα του χρόνου εκτέλεσης. Επίσης συλλέγονται μετρήσεις για τον κυλιόμενο μέσο όρο χρήσης κάθε vCPU σε παράθυρα του ενός λεπτού.

5.3 Αποτελέσματα

Ο πίνακας 5.2 δείχνει τον αριθμό vCPU που το σύστημα διαχείρισης πόρων επέλεξε να αποδώσει στην ευαίσθητη εφαρμογή στα διάφορα σενάρια κίνησης και ταυτόχρονης εκτέλεσης με μη ευαίσθητες εφαρμογές. Ο αριθμός διαθέσιμων vCPU είναι 6 και ο αριθμός αυτών που αποδίδονται στην ευαίσθητη εφαρμογή κυμαίνεται από 2 έως 5. Στην περίπτωση εκτέλεσης μαζί με το fluidanimate ή το swaptions, το δυναμικό σύστημα αποδίδει λιγότερους πόρους όταν η κίνηση είνα χαμηλή και περισσότερους όταν η κίνηση είναι υψηλή. Στην περίπτωση του swaptions επιλέγει να δώσει τον ίδιο αριθμό vCPU ανεξάρτητα από την κίνηση.

	Χαμηλή κίνηση (100)	Μεσαία κίνηση (250)	Υψηλή κίνηση (400)
swaptions	4	4	4
fluidanimete	2	3	5
streamcluster	4	5	5

Πίνακας 5.2: Αριθμός vCPU που ανατέθηκαν στην ευαίσθητη εφαρμογή από το προτεινόμενο δυναμικό σύστημα

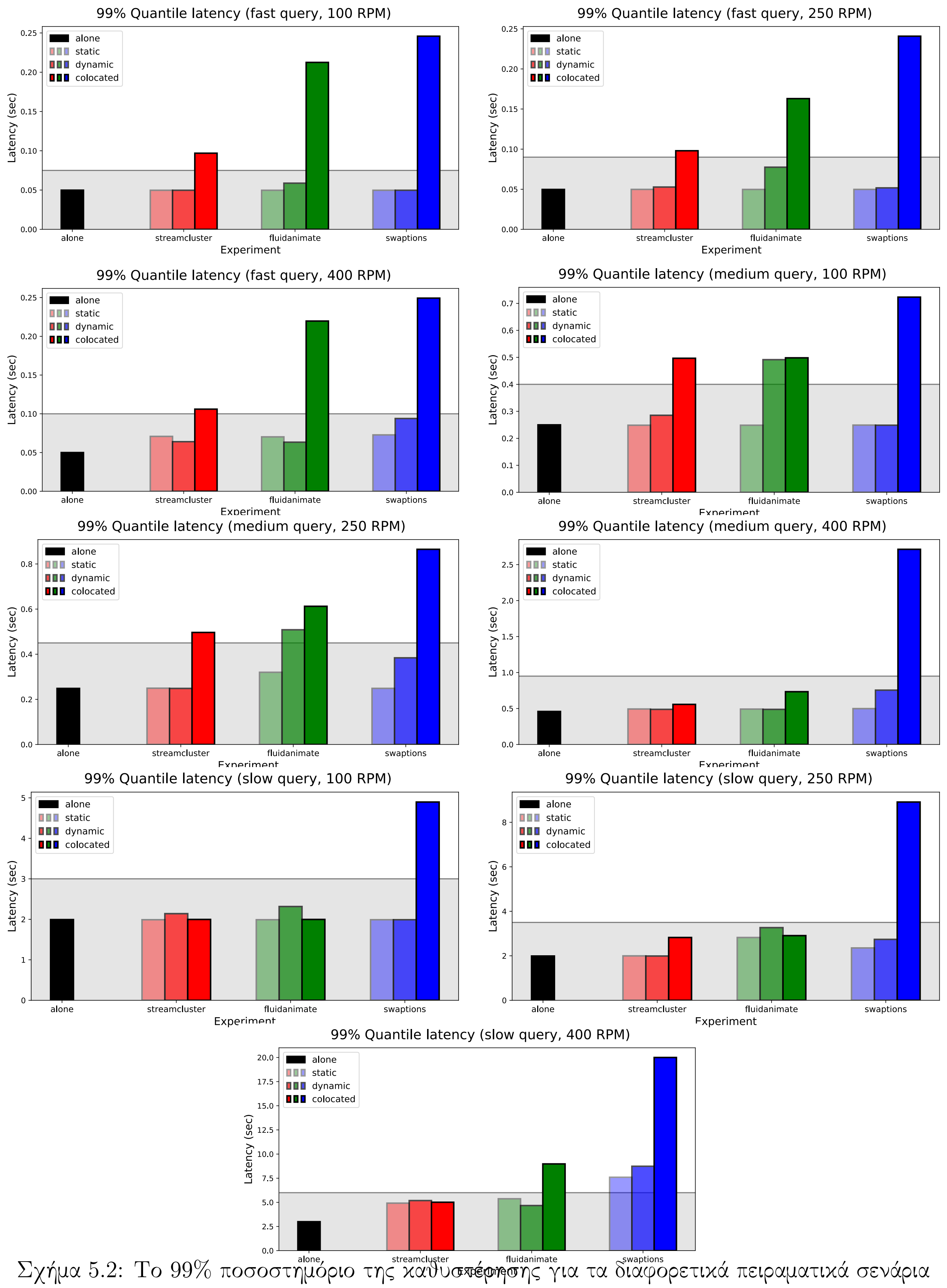
5.3.1 Αποτελέσματα μέσης καθυστέρησης

Τα διαγράμματα του σχήματος 5.2 δείχνουν το 99% ποσοστημόριο της καθυστέρησης εκτέλεσης των ερωτημάτων στην βάση δεδομένων για τα διάφορα σενάρια των πειραμάτων. Οι μπάρες με την ανοιχτή σκίαση αντιστοιχούν στο στατικό πείραμα, με την έντονη σκίαση στο ελεύθερο και με την ενδιάμεση σκίαση στο δυναμικό. Η γκρι γραμμή στο παρασκήνιο δείχνει τον στόχο καθυστέρησης για κάθε συνδυασμό ερωτήματος και φόρτου κίνησης.

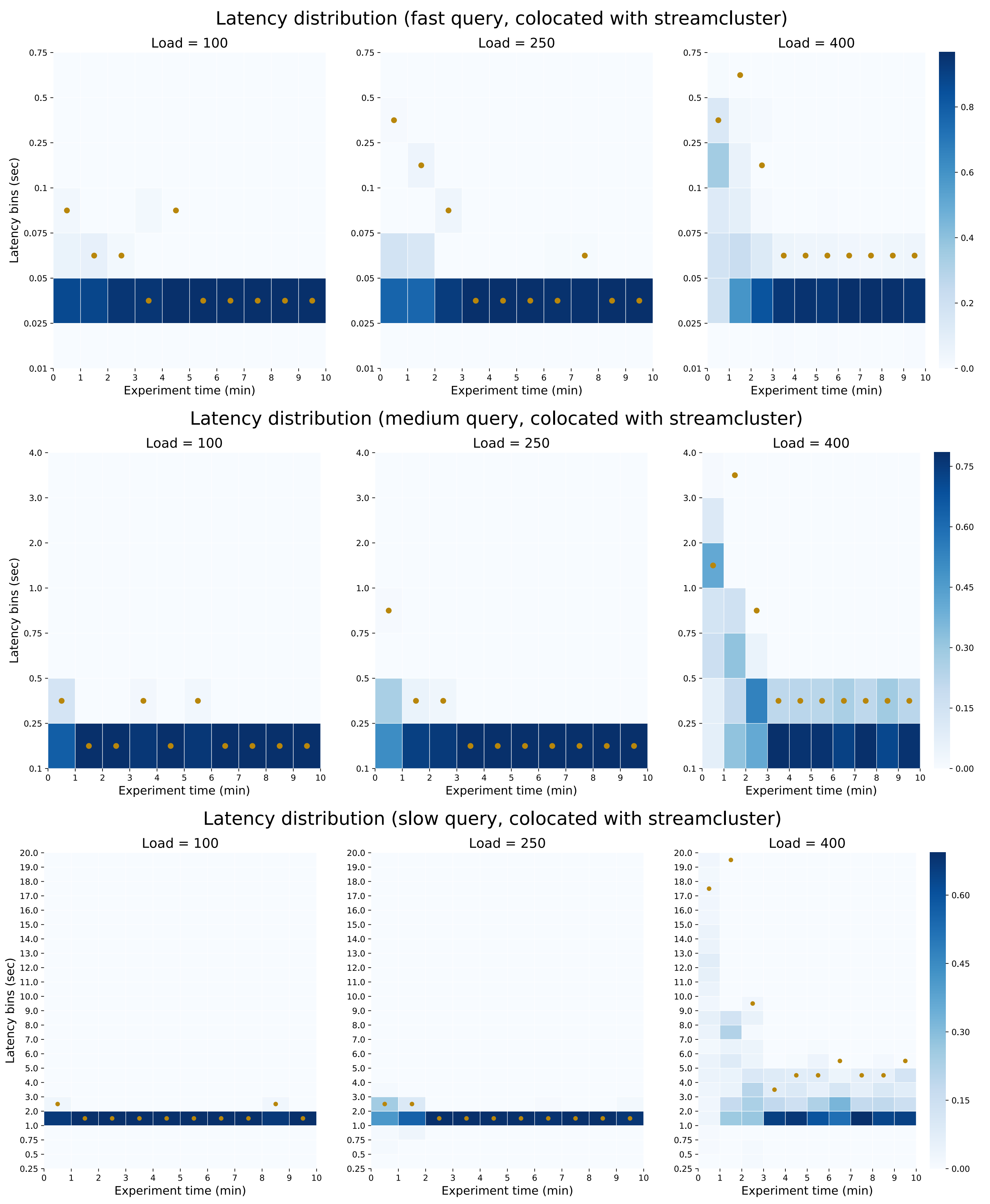
Η ελεύθερη χρήση πόρων είχε τα χειρότερα αποτελέσματα, καθώς παραβίαζε τον στόχο στο 70% των πειραμάτων. Αντιθέτως, η συντηρητική ανάθεση πόρων είχε τα καλύτερα αποτελέσματα παραβιάζοντας τον στόχο μόνο στο 4% των περιπτώσεων. Η προτεινόμενη δυναμική υλοποίηση αυξάνει ελαφρώς τις παραβιάσεις στόχων στο 11%.

Μεταβατικό φαινόμενο

Στην δυναμική υλοποίηση δημιουργείται ένα μεταβατικό φαινόμενο μέχρις ότου να συγκλίνει ο αλγόριθμος. Κατά το μεταβατικό φαινόμενο αυξάνεται προσωρινά η καθυστέρηση της ευαίσθητης εφαρμογής, όπως φαίνεται στο σχήμα 5.3. Αυτό οφείλεται στο γεγονός ότι όταν αλλάζουν οι πόροι που μπορεί να χρησιμοποιήσει μία εφαρμογή στον Κυβερνήτη, το container της πρέπει να καταστραφεί και να δημιουργηθεί ένα καινούριο με τις νέες ρυθμίσεις. Αυτή η διαδικασία διαρκεί στην περίπτωση της MySQL 15 με 20 δευτερόλεπτα, τα οποία είναι αρκετά για να συσσωρευτούν ερωτήματα. Αφού όμως συγκλίνει ο αλγόριθμος, η καθυστέρηση επανέρχεται γρήγορα στα επιθυμητά επίπεδα.



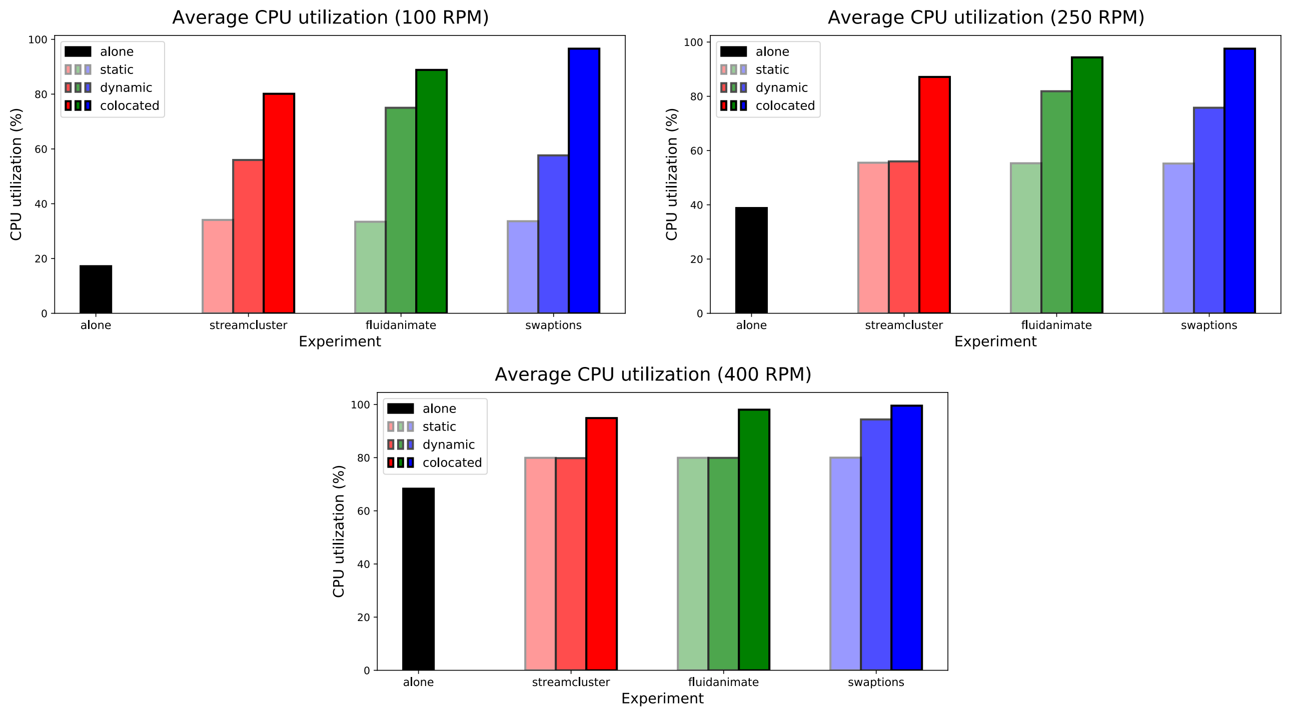
Σχήμα 5.2: Το 99% ποσοστημόριο της καθυστέρησης για τα διαφορετικά πειραματικά σενάρια



Σχήμα 5.3: Κατανομή της καθυστέρησης στην διάρκεια του πειράματος όταν υπάρχει συνδρομολόγηση με το streamcluster

Μέση χρησιμοποίηση επεξεργαστών

Το σχήμα 5.4 δείχνει την μέση χρησιμοποίηση των vCPU κατά την εκτέλεση των πειραμάτων. Η χρησιμοποίηση των vCPU είναι χαμηλότερη στο στατικό πείραμα και υψηλότερη στο ελεύθερο πείραμα. Η συντηρητική στατική ανάθεση πόρων αφήνει υπολογιστικούς ανεκμετάλλευτους και έτσι πετυχαίνει τους στόχους καθυστέρησης. Αντιθέτως, η ελεύθερη χρήση πόρων από τις εφαρμογής πιέζει τους επεξεργαστές και οδηγεί σε πολλές παραβιάσεις των στόχων καθυστέρησης. Το προτεινόμενο σύστημα διαχείρισης πόρων πετυχαίνει έναν συμβιβασμό μεταξύ της χρησιμοποίησης των διαθέσιμων πόρων και της καθυστέρησης. Συγκρινόμενο με το στατικό πείραμα, αυξάνει την χρησιμοποίηση των vCPU μέχρι 2,2 φορές και κατά 1,6 φορές κατά μέσο όρο.

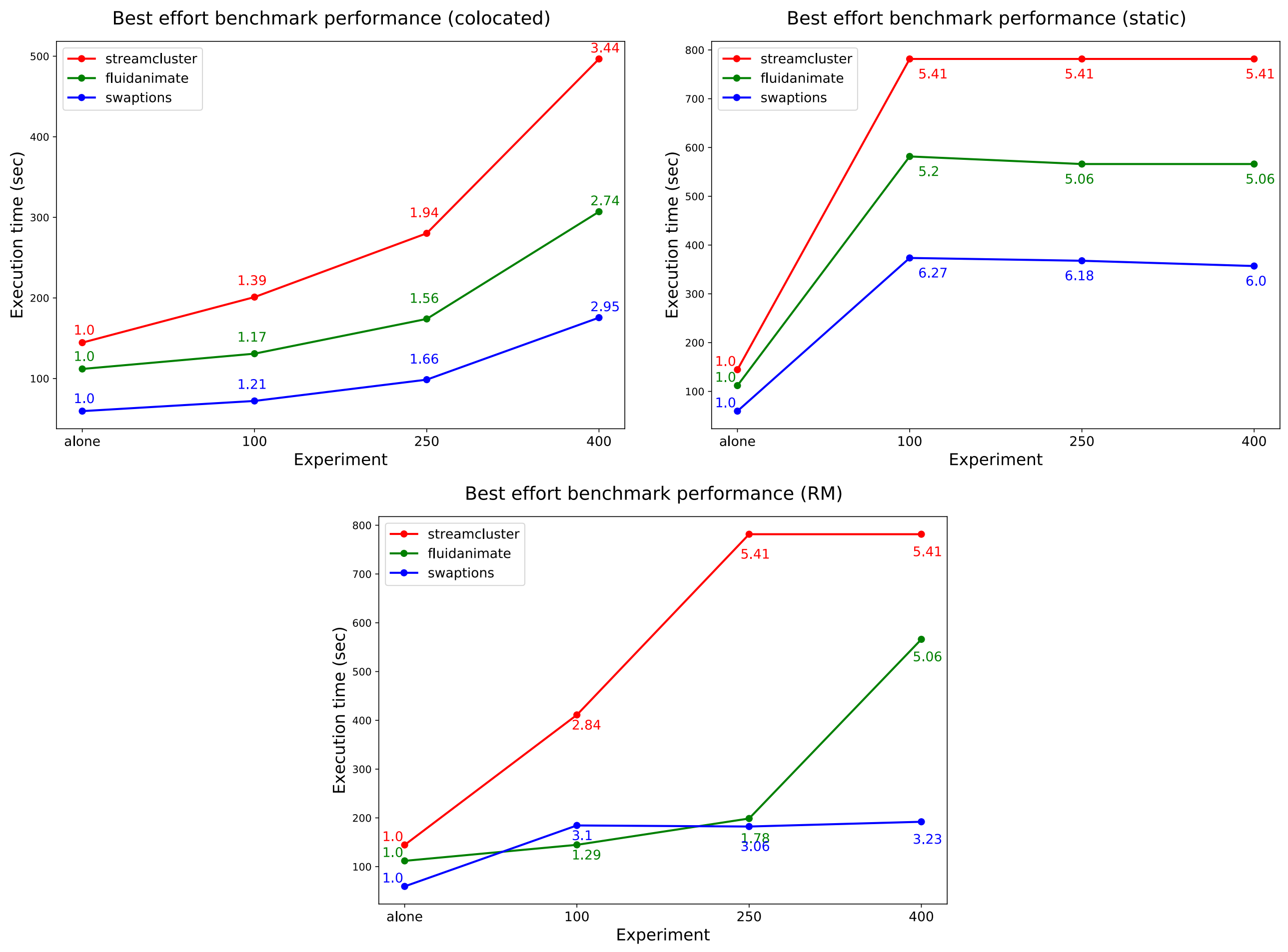


Σχήμα 5.4: Μέση χρησιμοποίηση των vCPU

5.3.2 Επίδοση μη ευαίσθητης εφαρμογής

Η επίδοση της μη ευαίσθητης εφαρμογής στα διάφορα πειράματα φαίνεται στο σχήμα 5.5. Για την αξιολόγηση της μη ευαίσθητης εφαρμογής υπολογίζεται η επιβράδυνση της σε σχέση με τον ιδανικό χρόνο εκτέλεσης (όταν τρέχει μόνη της σε ένα μηχάνημα). Στα παρακάτω διαγράμματα ο κατακόρυφος άξονας παρουσιάζει τον χρόνο εκτέλεσης και ο αριθμός πάνω από τις κουκκίδες υποδεικύνει την επιβράδυνση. Δεν παρουσιάζεται η διασπορά των μετρήσεων, διότι είναι αμελητέα.

Η επίδοση των μη ευαίσθητων PARSEC εφαρμογών είναι καλύτερη στο ελεύθερο πείραμα (πάνω αριστερά)· όμως το πείραμα αυτό εμφανίζει συχνές παραβιάσεις του στόχου καθυστέρησης της ευαίσθητης εφαρμογής, όπως αναλύθηκε προηγουμένως. Το στατικό πείραμα (πάνω δεξιά) περιορίζει υπερβολικά τις μη ευαίσθητες εφαρμογές και συνεπώς η επιβράδυνση είναι πολύ υψηλή. Το δυναμικό πείραμα (κάτω) πετυχαίνει μικρότερη επιβράδυνση από το στατικό, με λίγες μόνο παραβιάσεις των στόχων καθυστέρησης.



Σχήμα 5.5: Επίδοση των μη ευαίσθητων εφαρμογών. Ο κατακόρυφος άξονας δείχνει τον χρόνο εκτέλεσης και οι αριθμοί πάνω από τις κουκκίδες αντιπροσωπεύουν την επιβράδυνση σε σχέση με την ιδανική εκτέλεση.

Κεφάλαιο 6

Συμπεράσματα και μελλοντικές εργασίες

Στο πλαίσιο της παρούσας εργασίας σχεδιάστηκε και αξιολογήθηκε πειραματικά ένα σύστημα δυναμικής ανάθεσης πόρων για clusters που ελέγχονται από τον Κυβερνήτη. Το σύστημα χρησιμοποιεί το API του Κυβερνήτη και αξιοποιεί μετρήσεις καθυστέρησης που προέρχονται από το σύστημα Προμηθέας, προκειμένου να λάβει αποφάσεις. Η πειραματική αποτίμηση έγινε σε αντιδιαστολή με μία συντηρητική στατική ανάθεση πόρων στα containers και έδειξε ότι η δυναμική υλοποίηση καταφέρνει να αυξήσει την χρησιμοποίηση των κόμβων μέχρι και 2,2 φορές. Ταυτόχρονα, οι παραβιάσεις του στόχου καθυστέρησης για την ευαίσθητη εφαρμογή αυξάνονται πολύ λίγο, από 4% σε 11%. Η δυναμική υλοποίση παρουσίαζει ένα σύντομο μεταβατικό φαινόμενο, κατά το οποίο αυξάνεται προσωρινά η καθυστέρηση της ευαίσθητης εφαρμογής τα πρώτα λεπτά εκτέλεσής της. Γι' αυτό τον λόγο η προτεινόμενη δυναμική υλοποίηση είναι χρήσιμη για φορτία που θα τρέχουν για μεγάλα χρονικά διαστήματα και η κίνηση τους δεν αλλάζει απότομα.

Το προτεινόμενο σύστημα μπορεί να επεκταθεί μελλοντικά, ώστε να αξιοποιηθεί σε φάση παραγωγής. Για τον λόγο αυτό, είναι χρήσιμο να δοκιμαστεί σε ένα πραγματικό cluster με μεγάλο αριθμό κόμβων, ώστε να διαπιστωθεί αν το σύστημα κλιμακώνει. Επιπλέον, θα μπορούσε να χρησιμοποιηθεί μία βάση δεδομένων, στην οποία θα αποθηκεύονται όλα τα δεδομένα που αφορούν στην κατάσταση του cluster και των εφαρμογών που τρέχουν σε αυτό. Τα δεδομένα αυτά αποθηκεύονται στην παρούσα υλοποίση σε προσωρινές μεταβλητές.

Τέλος προτείνονται μερικές ιδέες για μελλοντική σχετική έρευνα. Αξίζει να δοκιμάστει σε clusters που ελέγχονται από τον Κυβερνήτη μία προσέγγιση παρόμοια με αυτή που παρουσιάζονται στις εργασίες Quasar [11] και Pythia [31]. Πιο αναλυτικά, οι νεοεισερχόμενες εφαρμογές θα τρέχουν απομονωμένες ενώ συλλέγονται δεδομένα για την επίδοσή τους, και η απόφαση δρομολόγησής τους σε κάποιο κόμβο θα λαμβάνεται αξιοποιώντας τα δεδομένα αυτά. Η προσέγγιση αυτή αντιμετωπίζει το πρόβλημα έλλειψης μηχανισμού ζωντανής μετακίνησης container στον Κυβερνήτη. Μία άλλη πρόταση είναι η ανάπτυξη ενός τέτοιου μηχανισμού μετακίνησης, ώστε τα container να μπορούν να μετακινηθούν σε άλλο κόμβο χωρίς να καταστρέφονται και να υποχρεώνονται να αρχίσουν την εκτέλεση τους από την αρχή.

Βιβλιογραφία

[1] Sameer Ajmani. *Go Concurrency Patterns: Context.* 2014. URL: `blog.golang.org/context`.

[2] Michael Armbrust et al. “Above the Clouds: A Berkeley View of Cloud Computing”. In: *University of California at Berkeley UCB/EECS-2009-28, February* 28 (Jan. 2009).

[3] Richard Brown et al. “Report to Congress on Server and Data Center Energy Efficiency: Public Law 109-431”. In: (Jan. 2007). DOI: `10.2172/929723`.

[4] Eric Chung et al. “Serving DNNs in Real Time at Datacenter Scale with Project Brainwave”. In: *IEEE Micro* 38 (Mar. 2018), pp. 8–20. DOI: `10.1109/MM.2018.022071131`.

[5] *Client-go GitHub repository.* URL: `github.com/kubernetes/client-go`.

[6] *Cloud Native Computing Foundation.* URL: `cncf.io`.

[7] McKinsey Company. “Revolutionizing data center efficiency”. In: 2008.

[8] *Considerations for large clusters.* URL: `kubernetes.io/docs/`.

[9] *Containers at Google.* URL: `https://cloud.google.com/containers`.

[10] *Containers vs. virtual machines.* 2019. URL: `https://docs.microsoft.com/en-us/virtualization/windowscontainers/about/containers-vs-vm`.

[11] Christina Delimitrou and Christos Kozyrakis. “Quasar: Resource-Efficient and QoS-Aware Cluster Management”. In: vol. 49. Feb. 2014, pp. 127–144. DOI: `10.1145/2541940.2541941`.

[12] *Docker: Empowering App Development for Developers.* URL: `https://www.docker.com`.

[13] *Flask documentation.* URL: `flask.palletsprojects.com/en/1.1.x/`.

[14] Yu Gan et al. “Leveraging Deep Learning to Improve Performance Predictability in Cloud Microservices with Seer”. In: *ACM SIGOPS Operating Systems Review* 53 (July 2019), pp. 34–39. DOI: `10.1145/3352020.3352026`.

[15] *Golang Microservices Example.* URL: `https://github.com/harlow/go-micro-services`.

[16] Henry Hoffmann et al. “Application heartbeats: A generic interface for specifying program performance and goals in autonomous computing environments”. In: Jan. 2010, pp. 79–88. DOI: `10.1145/1809049.1809065`.

[17] *How Kubernetes Deployments work.* URL: `https://thenewstack.io/kubernetes-deployments-work/`.

[18] Eric Jonas et al. “Cloud Programming Simplified: A Berkeley View on Serverless Computing”. In: (Feb. 2019).

[19] Norman Jouppi et al. “In-Datacenter Performance Analysis of a Tensor Processing Unit”. In: June 2017, pp. 1–12. DOI: `10.1145/3079856.3080246`.

[20] Ram Kannan et al. “Proctor: Detecting and Investigating Interference in Shared Datacenters”. In: Apr. 2018, pp. 76–86. DOI: 10.1109/ISPASS.2018.00016.

[21] *Kubernetes Components*. URL: https://www.leverege.com/iot-ebook/kubernetes-components.

[22] David Lo et al. “Improving Resource Efficiency at Scale with Heracles”. In: *ACM Transactions on Computer Systems* 34 (May 2016), pp. 1–33. DOI: 10.1145/2882783.

[23] *MySQL*. URL: mysql.com.

[24] *PARSEC Wiki*. 2010. URL: wiki.cs.princeton.edu/index.php/PARSEC.

[25] *Persistent Volumes on Kubernetes*. URL: https://docs.ovh.com/gb/en/kubernetes/ovh-kubernetes-persistent-volumes/.

[26] *Prometheus: Monitoring system time series database*. URL: https://www.prometheus.io.

[27] *The PARSEC Benchmark Suite*. 2007. URL: parsec.cs.princeton.edu.

[28] Muhammad Tirmazi et al. “Borg: the next generation”. In: Apr. 2020, pp. 1–14. DOI: 10.1145/3342195.3387517.

[29] A. Vasan et al. “Worth their watts? - an empirical study of datacenter servers”. In: *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*. 2010, pp. 1–10. DOI: 10.1109/HPCA.2010.5463056.

[30] *What is Platform-as-a-Service (PaaS)*. URL: https://www.cloudflare.com/learning/serverless/glossary/platform-as-a-service-paas.

[31] Ran Xu et al. “Pythia: Improving Datacenter Utilization via Precise Contention Prediction for Multiple Co-located Workloads”. In: Nov. 2018. DOI: 10.1145/3274808.3274820.

[32] Hailong Yang et al. “Bubble-flux: Precise Online QoS Management for Increased Utilization in Warehouse Scale Computers”. In: vol. 41. July 2013, pp. 607–618. ISBN: 9781450320795. DOI: 10.1145/2485922.2485974.

Part II

Efficient workload co-location on a Kubernetes cluster

List of Figures

Chapter 7

Introduction

7.1 Overview of cloud computing

Cloud computing has already changed significantly the IT industry and the way new software products are being developed and deployed. It has also transformed the way computer hardware is engineered, since the emphasis is now put on large-scale datacenter servers rather than personal computers and alone-standing servers. The cloud offers computing as a service, and as a result corporations and organizations in general do not need to invest effort and capital in provisioning state-of-the-art computing resources for deploying their own software; they can instead rent resources from public cloud providers.

As a more formal definition [2], cloud computing refers to both the applications delivered as services over the Internet and the hardware and systems software in datacenters that provide those services. The services provided are called **Software as a Service (SaaS)**, while the datacenter hardware and software is called a **Cloud**. Clouds that are available to the public and users can pay for the computing resources as they utilize them are called public clouds. Examples of public clouds include Amazon Web Services, Microsoft Azure and Google Cloud Engine. On the contrary, there are also private datacenters, the access to which is restricted to the organizations that own them and their clients.

Advantages of cloud computing

The emergence of cloud computing offers an abundance of privileges to datacenter owners and users. The advantages include the following:

- **Infinite resources on demand**
  Cloud forms the illusion that there are infinite resources which can be utilized on demand; thus organizations and users do not need to forecast future needs well in advance and prepare themselves. Decisions based on possible future demands can be risky and may lead to underutilized investments, if the demand growth is lower than expected.

- **Pay-as-you-go policies**
  Public cloud providers allow users to pay for the use of computing resources on a short-term basis. Most cloud providers charge users per-minute or even per-second of resource utilization.

- **Economies of scale**
  Building and operating large datacenters in multiple locations across the globe gives cloud providers the opportunity to form economies of scale. As a result, they pay computing utilities, such as hardware, electricity and power to a fraction of the cost offered to medium-sized datacenters.

- **Higher resource utilization**
  Cloud providers can also benefit from multi-tenancy to increase the utilization of their servers.

As multiple workloads from different users can be colocated to the same machines, the datacenter utilization can increase. For example, a datacenter deploying web applications may observe that the incoming load is low during nighttime and at that time batch jobs could be scheduled on the servers dedicated to the web apps.

Applications deployed on the cloud

Cloud platforms are very popular with a variety of application types [2]. In this section, the applications usually deployed on the cloud are split into categories and are briefly presented.

- **Mobile interactive applications**
  A widespread category of applications include web services that respond in real time to information provided by users. Examples of such services are hotel reservation web sites and e-banking platforms. Cloud is ideal for these applications, because it ensures high-availability and can easily handle the large datasets they utilize (often coming from different sources).

- **Parallel batch jobs**
  Except for interactive SaaS, cloud is also a popular place to host massively parallel batch jobs. These jobs include analytics applications based on frameworks like MapReduce and Handoop, which leverage very large amounts of data (in the scale of Gigabytes or even Terabytes) to analyze customer behaviour (recommendation systems) or forecast supply chain needs. They can benefit from the plethora of computational cores and accelerators like GPUs and FPGAs to process huge amount of data in parallel and dramatically reduce execution time.

- **Extension of desktop applications**
  Another class of cloud applications includes intensive desktop applications that are now being mitigated to the cloud. Examples range from text processing applications to scientific applications and 3D animation software. The traditional approach of running the applications locally required very large initial investments from end users or organizations, if the desktop applications they wished to run were heavy. However, the expensive infrastructure may be underutilized, if the heavy workloads are executed infrequently. Moving the application to a pay-as-you-go public cloud can be cost-efficient and guarantee higher performance.

7.2 Current trends in cloud computing

7.2.1 Multiple models of cloud computing

As the popularity of cloud computing has increased, different models and deployment strategies have emerged to meet specific needs of various users. In brief the three models of cloud computing (fig. 7.1) are:

- **IaaS** Infrastructure-as-a-Service is infrastructure on the cloud. The cloud provider manages the servers, storage and networking; the users have access over the Internet to cloud storage and virtual servers, where they can deploy their applications.

- **PaaS** Platform-as-a-Service goes one step further and provides users with a platform to build their applications. PaaS vendors offer development tools, middleware, operating systems, storage and database management, and infrastructure.

- **SaaS** Software-as-a-Service refers to fully built applications. These applications are offered to end users over the Internet and are managed and maintained by the service providers. An example of SaaS is a web-based email where users connect through a web browser.

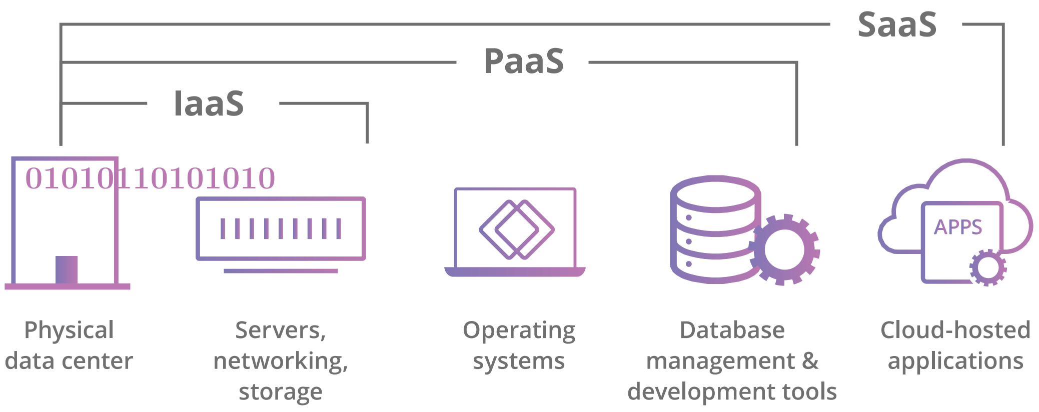

SaaS
PaaS
IaaS
010101110101010
APPS
Physical data center
Servers, networking, storage
Operating systems
Database management & development tools
Cloud-hosted applications

Figure 7.1: The three cloud service models [30]

7.2.2 Serverless computing

A recent trend in cloud computing is the adoption of a "serverless" approach [18]. This section aims at presenting the current condition and the needs driving this swift.

Cloud providers tend to provide their users with low-level virtual machines, which the latter can customize according to their needs and deploy their applications on. This approach is very flexible, for it allows the recreation of the local working environment on the cloud to simplify porting already functional software. However, it also comes with a large administrative overhead. Managing many instances of virtual machines can be a challenging task, as the system administrator has to tackle with the following responsibilities:

- Redundancy for high availability of the deployed services, so that a single point of failure does not result in down time.
- Backing up storage, to prevent a physical disaster from causing irreparable data loss.
- Load balancing to ensure that the incoming traffic is evenly distributed among the running instances.
- Autoscaling (scale-up or scale-down) to meet the fluctuations in incoming traffic.
- Incident logging for debugging purposes or performance analysis.
- System upgrades, including security patches.

Taking into account the difficulty of completing these administrative tasks, cloud providers have started to offer serverless computing products. The word "serverless" does not precisely describe this new trend, since the deployed applications still run on servers. The key difference is that users just write the code and cloud providers take the responsibility of server provisioning and task administration. The core of serverless computing is **Function as a Service (FaaS)**; the user provides the software functions and the cloud providers make sure that the application is successfully deployed and running. Cloud platforms also provide **Backend as a Service (BaaS)**, which is pre-written software for all tasks taking place on servers (e.g. user authentication, database management, cloud storage).

The first public cloud to offer serverless computing was Amazon in 2015 with AWS Lambda. Nowadays there is a large number of serverless platforms, like Google Cloud Functions, IBM Cloud Functions and Azure Functions. In all these platforms the deployed software functions scale automatically according to the incoming traffic, without any explicit provisioning, and the users are billed based on usage.

7.2.3 Microservices

Cloud applications are transitioning from monolithic implementations to a collection of smaller parts. Each of these parts is called a microservice, as it performs only one service, runs independently

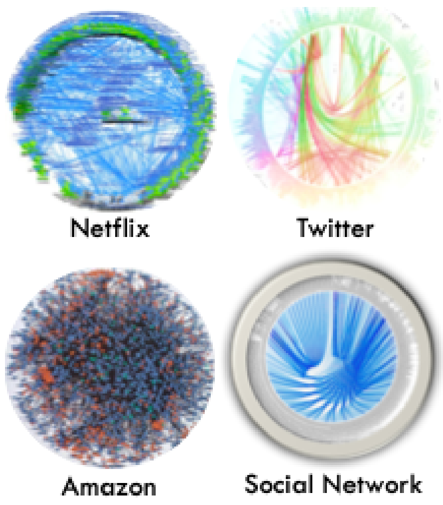


Figure 7.2: Examples of microservice graphs [14]

of other microservices, operates in its own environment and stores its own data. The individual microservices are connected in a chain to form a full application, and because a microservice can be part of different chains, a graph of microservice is generated. Examples of microservice graphs are presented in figure 7.2. From the end user's point of view, an application built with microservices has a single interface and works the same as a monolithic application.

Microservices are gaining popularity due to their advantages. Every microservice can be written in a different programming language and an already existing microservice can be used in many applications; these characteristics make the development process more flexible. This modular design approach makes it also easier to add or modify features of an application. In addition, there is the opportunity to selectively scale up or down specific parts of an application and thus avoid bottlenecks.

Although not necessary, microservices are usually deployed on cloud platforms. They can be deployed on PaaS, using the development platform made available by the cloud vendor. Another option is to deploy the microservices in containers or serverless. Serverless microservices run only when they are needed by the application, scale automatically and may be split in smaller cloud functions if a microservice has multiple functionalities.

7.3 Motivation

The topic of this thesis is efficient workload colocation on clusters orchestrated by Kubernetes. In the scope of this thesis, the workloads are split into two categories: latency-critical (LC) and best-effort (BE). The LC workloads have strict tail latency targets and are usually interactive web services. On the contrary, the BE workloads are usually batch jobs created by big data analytics frameworks or recommendation systems, and have loose or no latency targets. The latency targets for applications deployed on cloud platforms are specified on formal agreements between cloud providers and organizations called Service Level Agreements (SLA).

Previous work has shown that resource utilization in datacenters is low, although large amounts of money should be invested in the construction, equipment and operation of a datacenter [3]. Research data from the late 2000s show that the resource utilization in private datacenters are between 6% [7] and 12% [29]. During the same period, the CPU utilization in Google servers managed by the Borg software was 25 - 35%. More recent data from 2019 show that CPU utilization in Google servers has increased to 60% on average, as shown in figure 7.3. This figure shows the cumulative distribution function of the CPU and memory utilization for 8 different Google servers (labeled a-h) and compares it with equivalent data from 2011. The Borg resource manager is a sophisticated software and thus Google servers are on the high end, as far as resource utilization is concerned.

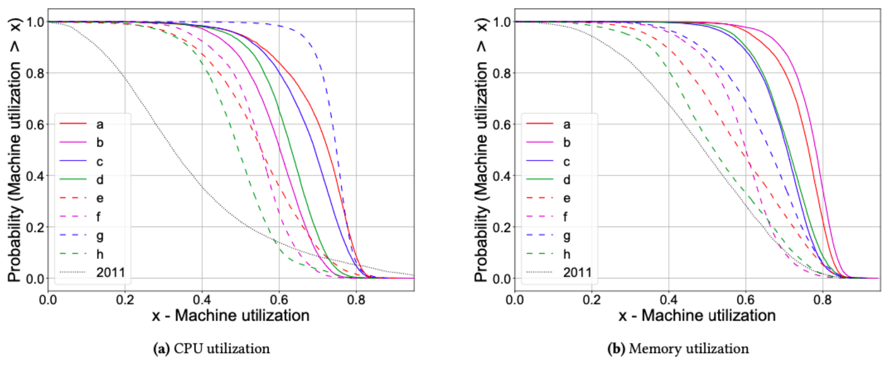


Figure 7.3: Cumulative distribution function for CPU and memory utilization in Google servers managed by Borg in 2009 and 2019 [28]

The cause of low utilization may lie in the reservation-centric approach many datacenter administrators have adopted. The administrators decide themselves the amount of resources every workload is allowed to use. Due to the strict agreements signed with customers, they are afraid that LC application might violate their SLA during an unexpected traffic spike. As a consequence, they allocate more resources than necessary to LC application. Quasar [11] proves than only 10% of the reservations are right-sized; 70% of them overestimate the reservations by up to 10x. Figure 7.4 compares resource utilization and reservation over 30 days for a large-scale cluster owned by Twitter and managed by Mesos, and shows that CPU overestimation is 3-5x and memory overestimation is 1,5-2x.

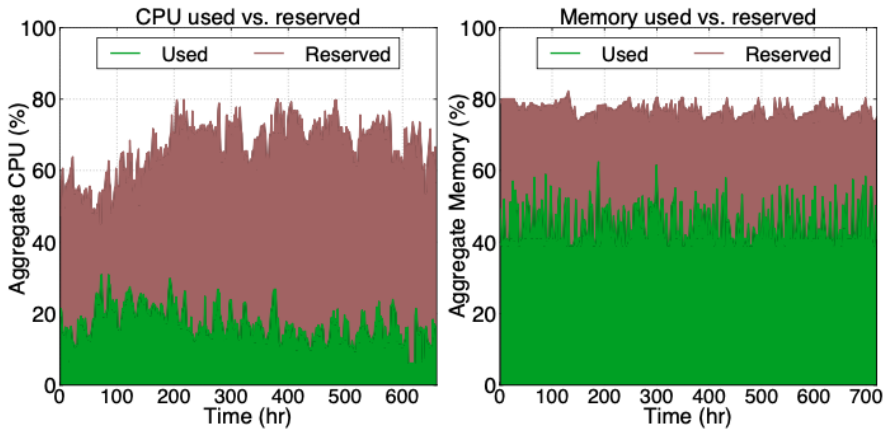


Figure 7.4: Reservations and actual usage in Twitter servers [11]

Although there are numerous research publications on resource management in datacenters [22] [11] [20] [31] [14], the majority of them assumes that workload run in virtual machines or bare-metal. There is a recent trend to run application in containers, which are lighter than VMs, boot very quickly and can be scaled up and down. Kubernetes [8] is a very popular tool for orchestrating containers; it is responsible for scheduling, managing and scaling containerized applications. Taking into account the low resource utilization, this thesis aims at investigating ways to optimize resource usage in a Kubernetes cluster through the efficient colocation of LC and BE workloads.

7.4 Contributions

The main contributions of this thesis are the following:

- The different approaches in resource management for large-scale cloud systems are summarized, compared and contrasted. A selected collection of relevant research articles is presented; each of them proposes a complete system for managing shared resources in multi-tenant datacenters.
- A resource management system for Kubernetes clusters is designed, implemented and evaluated. As opposed to previous work, the proposed system works with containers orchestrated by Kubernetes rather than virtual machines or bare-metal applications. Furthermore, it leverages the Kubernetes API functionalities for resource management and requires no measurements from hardware counters.

7.5 Structure of the thesis

This thesis starts with presenting the technologies that will be used in chapter 8. These technologies include the container runtime Docker, the container orchestrator Kubernetes and a tool for cluster monitoring called Prometheus. In chapter 9 selected research articles that deal with the topic of resource manager in large-scale cloud systems are briefly presented. Chapter 10 presents the dynamic resource manager this thesis proposes for Kubernetes clusters. The algorithm is thoroughly explained and the implementation code is presented in detail. The proposed resource manager has been tested on a small local cluster managed by Kubernetes using a LC application designed for the thesis needs. Chapter 11 includes the presentation of the LC application and the experiments results. The thesis concludes with a brief summary and suggestions for future work in chapter 12.

Chapter 8

Background

8.1 Docker

8.1.1 Containers

A container is a standard unit of software that packs all the code of an application, along with all necessary dependencies and configuration. [9]. Containers are lightweight and portable, and can be easily shared. They are isolated from the environment they live in, so the same container can be easily and consistently deployed in a public cloud, a private datacenter or even a personal computer. This abstraction allows development teams to focus more on the software itself, rather than bothering about software version incompatibilities and server configuration; therefore, the development and deployment of an application is more efficient.

A container has many similarities with a virtual machine, but these two are substantially different [10]. Both provide isolation from the host machine and other virtualized entities running on the same machine. Yet, VMs offer a more strict isolation; containerized applications appear in the user space as isolated processes. Both are fully operational and alone-standing virtualized environments allowing access to the underlying hardware. A VM virtualizes the whole hardware stack and runs a complete operating system including the kernel. On the contrary, a container runs only the user portion of the operating system and includes services and libraries that are necessary for the specific application. As a result, containers are much more lightweight that VMs, take up less memory and need less time to boot.

8.1.2 Docker container technology

Docker [12] is one of the most popular container technologies, initially launched in 2013. Docker makes use of Linux concepts like cgroups and namespaces to separate application dependencies from infrastructure. Today Docker is used by all major cloud providers and serverless frameworks. It includes the open-source libcontainer library, which was originally written especially for itself.

Docker containers need a Docker runtime to run successfully. This runtime is called Docker Engine and it allows containerized applications to run on any infrastructure. It has a wide support for public cloud and this includes hybrid solutions.

8.2 Kubernetes

8.2.1 Outline

Kubernetes [8], also known as K8s, is an open-source container orchestration framework originally developed by Google. Kubernetes automates deployment, scaling and management of containerized applications in different infrastructure like physical servers, virtual machines, public clouds, or even hybrid environments. Kubernetes is able to manage large clusters with up to 5000 nodes and 100 containers per node.

It implements the following functionalities:

- **Load balancing**
  Kubernetes can expose a containerized application running on one or multiple containers using a DNS name or IP address; thus the application can be accessible from outside the cluster. Kubernetes can guarantee that the load assigned to each container is evenly distributed.

- **Storage orchestration**
  Kubernetes is able to deal with storage systems and supports mounting of volumes from local storage, public cloud providers or network file systems.

- **Update of deployment status**
  The desired state for deployed containerized applications can be easily described with YAML or JSON files. The actual state can be changed at a controlled rate; for example the number of deployed containers can be changed.

- **Resource management**
  Kubernetes allows cluster administrators to determine the CPU and RAM usage for deployed applications and places the containers to the nodes so as to use best the available resources.

- **High availability**
  Kubernetes restarts failed containers, replaces them if the desired state is updated, kills non-responding ones and starts advertising them only when they are ready to serve.

- **Security management**
  Kubernetes puts a strong emphasis on security. For this reason, it includes tools for storing and managing sensitive information like passwords, OAuth tokens and SSH keys without exposing them in the application code.

- **Disaster recovery**
  In case the infrastructure fails for any reason, Kubernetes will try to recover the cluster and run the containerized applications from the latest state.

8.2.2 Kubernetes objects

Kubernetes objects are persistent entities used to represent the state of the cluster. They describe which containerized applications run in the cluster, what resources are available to them and the policies regarding them (e.g. restart policy). The objects can be created through the Kubernetes API, which is discussed in paragraph 8.2.3. After they are created, they represent the cluster's desired state and Kubernetes will constantly work to move the cluster current state towards the desired state.

Kubernetes objects include fields `spec` and `status`, which define the object configuration. The `spec` field is completed during the creation of the object and represents its desired state. On the contrary, the `status` represents the current state of the object and is updated by Kubernetes and its components.

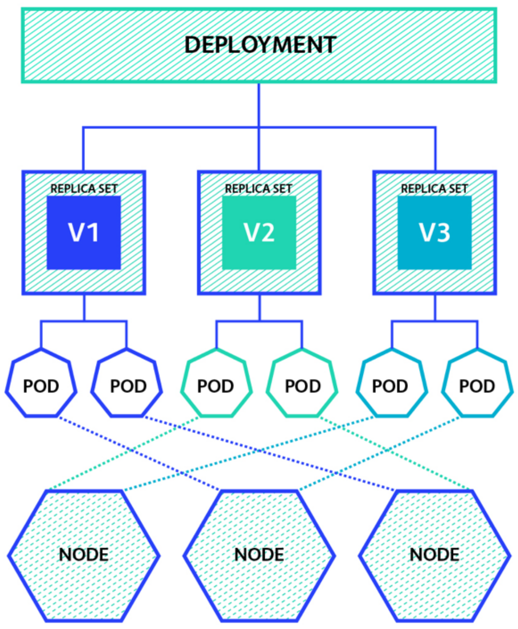


Figure 8.1: ReplicaSets manage pod replicas and are managed by deployments [17]

A fundamental Kubernetes object is the **pod**. A pod is the smallest unit Kubernetes can handle and is basically an abstraction over containers. With the introduction of pods, Kubernetes is independent of the container technologies. A pod can include more than one containers, but usually there is one container in each pod. Kubernetes creates a virtual network over the cluster and assigns an unique internal IP address to each pod; if a pod fails and gets recreated, a new IP is assigned to it. If a pod has more than one containers, then all containers share the same IP address and can communicate with each other via `localhost` or other inter-process communication mechanisms like shared memory and pipes.

The lack of a permanent IP address can be a problem when different pods try to communicate with each other. This problem is solved by Kubernetes **services**, which are an abstract way to expose an application running on a set of pods as a network service with a permanent IP address. For example, a complex application runs the frontend and backend in different pods and there are multiple replicas of the backend pod due to high traffic. The frontend and backend must communicate and this can be done by creating two services. A user request will be caught by the frontend service and be forwarded to the frontend pod; a frontend request will be caught by the backend service and be forwarded to one of the backend pods, since a service also functions as a load balancer. The frontend does not need to know that there are multiple backend pods and or if one of them crashed and acquired a new IP address.

Multiple replicas of the same pod (like the ones mentioned in the previous example) are managed by a **ReplicaSet**. The purpose of a replicaSet is to maintain a stable number of identical pods. ReplicaSets are in turn managed by the higher-level concept of **Deployment**. Deployments use replicaSets as a mechanism to orchestrate Pod creation, deletion and updates. The Deployment creates automatically a replicaSet and can update it with server-side rolling updates. In case that one pod replica should be placed on each node, because it provides a node-level function like node monitoring or logging, a **daemonSet** object can be used. The pods created by a daemonSet have a lifecycle tied to the lifecycle of the node they live in. Finally, pods that are supposed to terminate on their own after execution (also called batch jobs) can be created with a **Job**.

For stateful applications, the most suitable object type is **StetefulSet**. Like deployments, statefulSets manage a set of identical pods; however pods managed by a statefulSet have a unique persistent identifier, which remains even if a pod fails and is recreated. These identifiers make it easier to match storage volumes to new pods replacing failed ones. Kubernetes manages storage by offering an API that splits storage provision and consumption. **PersistentVolume** is a piece of storage in the cluster provided statically by the administrator or dynamically by the cloud provider (e.g. AWS, Azure etc) or by a cloud infrastructure platform (e.g. Openstack). A persistentVolume has a lifecycle independent of any pod that utilizes it. A persistentVolume is consumed by a **persistentVolumeClaim**.

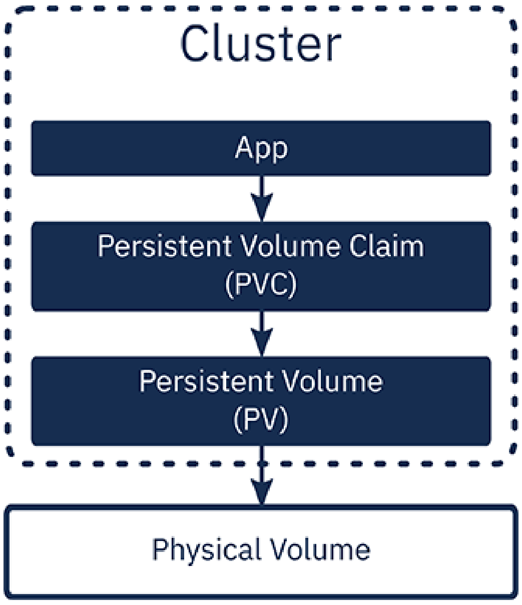


Figure 8.2: Persistent storage on Kubernetes [25]

8.2.3 Kubernetes components

Kubernetes manages clusters made up of worker machines called nodes, which run the containerized applications. Every cluster has at least one worker node. Worker nodes host the pods, which are the components of application workload. Every functional worker node must run the following components:

- **Container runtime**
  The container runtime is the software responsible for running containers. Kubernetes supports a number of container runtimes, including Docker and containerd.

- **Kubelet**
  Kubelet interacts with both the container and the node. It is responsible for starting containers in pods and ensures that the pods follow their specification (PodSpec), are running and remain healthy.

- **Kube-proxy**
  Kube-proxy forwards the requests coming from network sessions inside or outside the cluster to the pods. Kube-proxy has intelligent forwarding logic and makes sure that communication is efficient and with low overhead.

The worker nodes and the pods in the cluster are managed by the **control plane**. The control plane is the container orchestration layer that exposes the API and interfaces to define, deploy and manage the lifecycle of containers. It takes decisions about the whole cluster (e.g. which node a new pod will be scheduled on), detects and responds to cluster events (e.g. scaling up an application deployment). The control plane components run on specific machines called masters, which do not run application containers. Every cluster has at least one master (multiple masters guarantee high availability and fault tolerance).

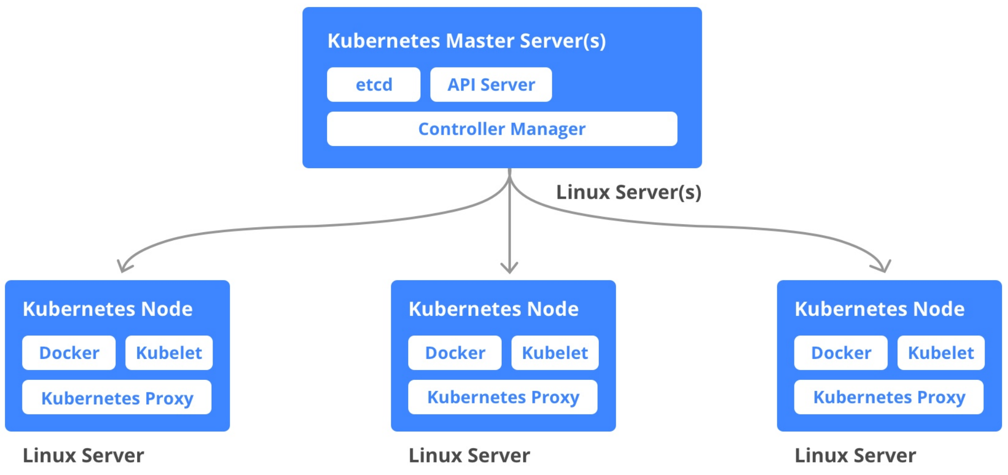


Figure 8.3: Kubernetes architecture and components [21]

The control plane is made up of the following components:

- **kube-apiserver**
  The API server exposes the Kubernetes API and can be considered as the frontend of the control plane. kube-apiserver is the main implementation of the API server. In large clusters, multiple instances of the API servers can be spawned for load balancing. A Kubernetes user can access the API server through a user interface (UI) like Kubernetes dashboard, a command line tool like kubectl or REST requests. The requests coming from the users are validated, so that only authenticated ones are forwarded to the cluster.

- **kube-scheduler**
  The scheduler watches for newly created pods that are not assigned to any node and decides what node they should by placed on. The scheduling decisions are complex, since a variety of factors should be taken into account (resource requirements, policy constraints, affinity and anti-affinity, data locality). The scheduler only decides on which node the pod will be placed; the actual pod is started by the kubelet instance running on the selected node.

- **kube-controller-manager**
  This component runs controller processes. A controller implements a control loop that watches the shared state of the cluster through the API server and makes the changes needed to move from the current state towards the desired state. The controller manager runs the following controllers:

  – Node controller: Watches the state of nodes and responds when a node goes down.

  – Replication controller: Monitors the desired number of pods and makes sure it matches the actual number of running pods.

  – Endpoints controller: An endpoint is the pod IP address and a port, through which it can communicate with the cluster and outer world. The endpoints controller watches for new endpoints (i.e. in new pods or restarted pods which have a new IP).

- **etcd**
  etcd is a consistent and highly available key-value store of the cluster state. All cluster changes (e.g. creation of a new pod) are stored in etcd. It can be described as the "cluster brain", because all components retrieve data from etcd and their decisions are based on it. For backup purposes, a cluster can have multiple master nodes and then etcd forms a distributed storage across all master nodes.

- **cloud-controller-manager**
  Cloud controller manager links the cluster to the cloud provider API, when the cluster is deployed on a public cloud like Google Cloud Engine (GCE), Amazon Web Services (AWS) or Microsoft Azure. It runs the controllers that are dependent on the cloud platform; thus a self-hosted cluster has no cloud controller manager. It can be scaled to more than one masters, in order to improve performance or avoid outages. It may run the following controllers:

  – Node controller: Checks if a failed node is deleted from the cloud platform

  – Route controller: Sets up routes in the underlying cloud infrastructure

  – Service controller: Creates, updates and deletes cloud provider load balancers

8.3 Prometheus

Prometheus [26] is an open-source monitoring and alerting system, based on a custom time series database implementation. It was originally developed in SoundCloud, but later joined the Cloud Native Computing Foundation [6] as a hosted project. It has become a popular monitoring system for container and microservice environments, because it allows administrators to have an insight on what is happening on hardware and application level. Prometheus reports software errors and exceptions, service latency, hardware outages and resource utilization, thus making the monitoring of large-scale, geographically distributed clusters running hundreds of applications more efficient.

The main component of Prometheus is Prometheus server. It is comprised of a time series database, which stores all metrics data (e.g. CPU usage, number of exceptions in an application). The data gets stored in the database by a data retrieval worker, which pulls the metrics data from targets (i.e. applications, services, servers). Finally, there is an HTTP server that accepts queries and forwards them to the database. The queries are written in a tailor-made query language called PromQL.

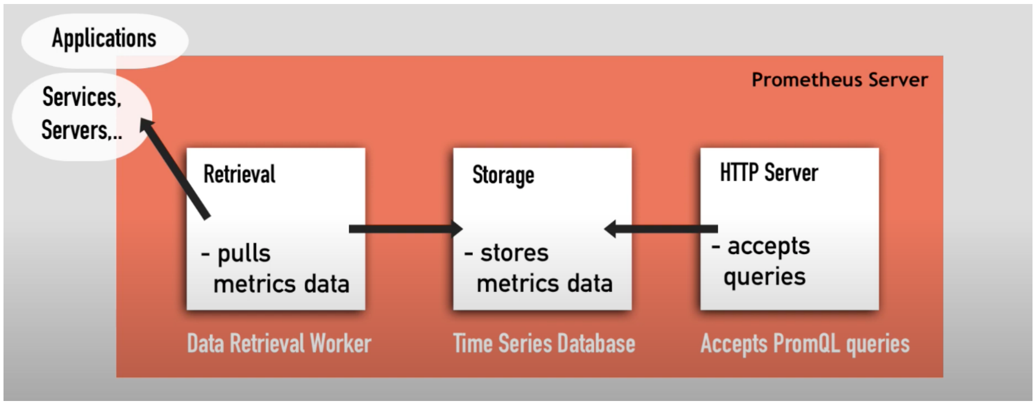


Figure 8.4: Prometheus architecture [26]

8.3.1 Time series database

Prometheus stores all the retrieved data as time series. A time series is a stream of timestamped values identified by a metric name and key-value pairs called labels. The time series is comprised of data structs called samples with a float64 value and a millisecond-precision timestamp. The data stored by Prometheus is multi-dimensional and this functionality is enabled by the labels. To be more specific, any given combination of labels for the same metric name define a particular (possibly multidimensional) time series; for example all HTTP requests (metric name) that use POST method and the return status is 200 (labels). Changing, adding or removing a label will result in a different time series. Given a metric name and a label key-value, a time series is identified using the following notation:

```
<metric name>{<label name>=<label value>, ...}
```

Targets must report metrics in specific formats, else Prometheus server will not be able to retrieve the data and push them to the time series database. Targets can use Prometheus client libraries which are available in multiple programming languages like Go, Python, Java and Ruby. The supported metric types are the following:

- **Counter**
  A counter is a cumulative metric that represents a single monotonically increasing counter whose value can only increase or be reset to zero on restart. For example, a counter could represent the number of requests served, tasks completed, or errors.

- **Gauge**
  A gauge is a metric that represents a single numerical value that can arbitrarily go up and down. Examples of metrics that can be represented with is a gauge are CPU and memory utilization.

- **Histogram**
  A histogram samples observations (usually request durations or response sizes) and counts them in configurable buckets. It also provides a sum of all observed values and exposes it as `<metricName>_sum`. To access a specific cumulative bucket, the notion `<metricName>{le="<upper inclusive bound>"}` is used. There is also the `histogram_quantile` function that allows the calculation of quantiles from the histogram data.

8.3.2 PromQL

Prometheus provides a tailor-made query language for the time series database. It is called Prometheus Query Language (PromQL) and allows users to select and aggregate time series data. The queries return four different data types:

- **Instant vector**
  An instant vector is a set of time series containing one sample per time series, all sharing the same timestamp. An instant vector is returned for example by the PromQL query `http_requests_total`. This query returns the latest sample for all time series (all possible label combinations) with metric name `http_requests_total`.

- **Range vector**
  A range vector is a set of time series containing a range of data points over time for each time series. Compared to an instant vector, it returns the same time series but not only the latest sample. For example, a range vector is returned by the query `http_requests_total[1m]`; it returns the samples from the last one minutes for all time series with metric name `http_requests_total`.

- **Scalar**
  A scalar is a simple numerical floating-point value. A scalar is returned when the PromQL query returns just one sample from one time series.

Chapter 9

Related Work

The problem of resource management in large-scale cloud systems has been a popular subject for numerous computer science researchers and as a result many relevant scientific works have been published. In this chapter a number of selected articles is briefly presented.

9.1 Heracles

Heracles [22] is a dynamic, mostly online resource manager which allows the colocation of a latency-critical (LC) application with any best-effort (BE) application. The LC applications operate under strict service level objectives (SLO) and Heracles aims at eliminating SLO violations while optimizing the throughput of the BE applications. An instance runs on each server and leverages only local information. Colocation is a challenging task, because applications running concurrently on a server compete for shared resources and OS-level scheduling mechanisms - such as Linux's completely fair scheduler (CFS) - can lead to SLO violations. Furthermore, static allocation of resources is not a solution either, because one the one hand the LC might need to utilize all the available during an unexpected load spike and on the other hand may lead to very low resource utilization.

The authors of Heracles characterized and analyzed the interference on shared resources for LC benchmarks running on one server and recorded their tail latency under various load scenarios. The LC application ran along with a tailor-made BE applications that stress a specific resource in isolation. To be more specific, competition for CPU resources has been observed with the LC and BE application running on different HyperThreads. For the last-level cache (LLC) and the memory bandwidth, the applications were pinned to different cores on the same socket. The same configuration was used for the power experiments, except that the BE application was power-intensive and forced the CPU to work on the lowest possible clock frequency. For the network utilization, the TCP congestion control mechanism was used to throttle excessive BE network usage. These experiments lead to the conclusion that static allocation for shared resources result in either underutilization or SLO violations.

Heracles' design allows the dynamic adjustment of hardware and software isolation mechanisms, namely `cpuset cgroups` for CPU core isolation, Intel's cache allocation technology (CAT) for the LLC, a custom-made software monitor for memory bandwidth (since there are no commercially available chips with a memory bandwidth partitioning mechanism) which requires offline information, Intel's Running Average Power Limit (RAPL) for power consumption and `qdisc` scheduler for network traffic isolation. Heracles tries to solve the high dimension problem of optimal resource allocation for all isolation mechanisms and for each possible combination of LC and BE application and load as an optimization problem where the objective is maximizing utilization with the constraint of the SLO. This problem can be decomposed to easier one or two-dimensional subproblems, as the interference lead to performance degradation when the shared resource becomes saturated. These subproblems can be solved with the gradient descent method.

The evaluation of Heracles relies on measurements about the LC application latency and the server utilization. Experiments are carried out both on a single server and a small cluster. Heracles achieves no SLO violation in either setup, but leaves a latency slack in many cases. An increase in resource utilization and throughput is also achieved, since an average of 90% server utilization is reached. Furthermore, Heracles puts an emphasis on energy efficiency realizing that increased energy costs have a negative impact on datacenter operation, and records a gain in energy efficiency from 2.3 to 3.4 times.

9.2 Quasar

Quasar [11] is a centralized cluster management system that aims at increasing server utilization while maintaining a high quality of service (QoS). The authors of this scientific article investigate data from reservation-based cloud systems and reach the conclusion that only a small fraction of the resource reservations are right-sized. Most workloads tend to overestimate the resource they need to satisfy the QoS, but this strategy leads to low resource utilization in datacenters. For this reason, Quasar adopts a performance-centric approach and asks users about the performance constraints of their applications rather than the low-level resources they need. These performance constraints may be expressed in terms of latency or throughput (i.e. queries per second), depending on the application type. It is up to Quasar to determine the optimal amount of resources to be allocated to each workload.

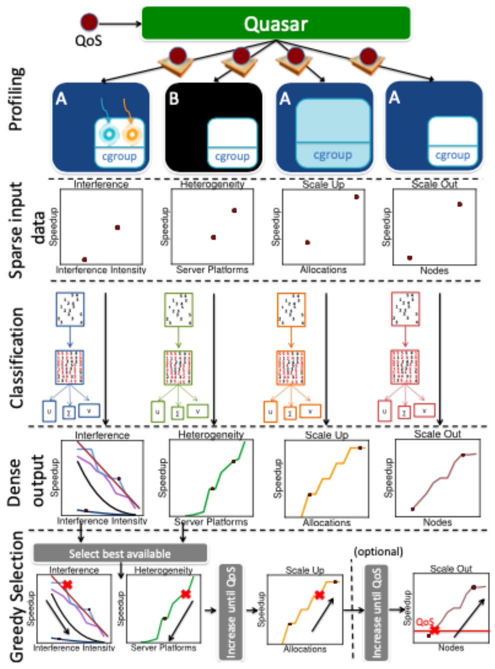


Figure 9.1: Quasar architecture [11]

Quasar uses collaborative filtering to quickly estimate how the performance of a application is affected by the amount and type of resources as well as colocation with other applications. Collaborative filtering is a classification technique often used in recommendation systems and is used as an alternative to exhaustive space exploration. Every incoming workload is profiled for a short time period on predefined servers. The data collected is combined with data gathered from applications characterized online and the data collected from previous applications scheduled on the cluster. Quasar make four independent classifications in parallel: amount of resources (scale-up), number of nodes (scale-out), server configuration (heterogeneity) and interference. These classifications are adjusted to the different workload types (i.e. no scale-out for single-node applications). The decomposition of the original classification problem in four subproblems might slightly degrade the classification accuracy, but decreases the complexity significantly.

The classification results are given as input to a greedy scheduler responsible for the resource allocation and assignment (i.e. the scheduler decides on which servers the workload is scheduled and how many resources it can take up). The target of the scheduler is to allocate the minimum amount of resources needed to satisfy the workload's QoS constraints. Except for placing new applications, Quasar also keeps the per-workload and per-node state. If SLO violations or idling resources (due to the workload changing phase, varying traffic or classification failure) are detected, Quasar reclassifies the workload and adjusts the allocation and assignment.

The proposed cluster management system is evaluated with a wide variety of workloads and traffic scenarios. The experiments are carried out in two large-scale clusters with 40 and 200 server respectively. The supported workloads include distributed analytics frameworks (e.g. Handoop and Spark), web-serving applications, NoSQL databases (e.g. memory-based memcached, or disk-based Cassandra) and single-node batch workloads (e.g. PARSEC, SPEC2006). The evaluated traffic scenarios are flat traffic, fluctuated and traffic with big spikes. Quasar is compared to the reservation-based schedulers of the analytics frameworks and to auto-scaling systems. The experiments show that Quasar manages to speedup workload execution while improving overall cluster utilization.

9.3 Proctor

Metric	Source
CPU utilization	`perf`
Page faults per second	`perf`
Context switches per second	`perf`
Network throughput (bytes transfered)	`netstat`
Cache misses (L1, L2, LLC)	`perf`
I/O requests	`iostat`
Branch misses	`perf`

Table 9.1: Metrics gathered by Proctor

Proctor [20] is a real-time, centralized monitoring system for datacenters, that aims at detecting intrusive virtual machines as well as their victims and the saturated shared resources (network, I/O, cache, CPU). The authors of this scientific work understand that private datacenters and public clouds run many new applications with no prior performance data, which makes the detection of performance degradation a difficult task. Moreover, reduced performance results from one or more stressed shared resources; therefore effective colocation requires the investigation of the contention source. It is also clear to the authors that the above mentioned detection and investigation tasks must have a low performance overhead.

Proctor includes a Performance Degradation Detector (PDD) to detect performance anomalies leveraging time series data from a QoS metric specific for each application. Applications report their QoS constraints and their current performance through the Application Heartbeats framework [16]. These time series often contain a significant amount of noise and sharp changes. Although popular noise reduction techniques like Kalman filtering are not suitable for such time series, median filtering reduces noise while keeping drastic changes. Afterwards, a signal processing algorithm called step detection is used to identify sudden changes in the QoS metric time series. These changes represent the beginning of degraded performance periods.

Having detected the sudden drops in performance, Proctor invokes the Performance Degradation Identification (PDI) to examine which virtual machines and shared resources are to blame for the reduced performance. For this reason, Proctor collects time series data from low-level counters for each VM and correlates them with the metrics from the VM's primary QoS metric. The metrics utilized and the software tools they are obtained from appear in table 9.1. Leveraging the whole amount of data makes the correlation very computationally expensive and time consuming; thus Proctor samples the available data. The authors put a great emphasis on selecting a representative data population; they observe that the data are far from normally distributed and choose the χ^2-test to check if a sample is representative of the population. They also suggest that a sampling rate of 6.5% is a good trade-off between accuracy and performance overhead.

When the disruptive VM and the stressed resource have been identified, Proctor applies a simple mitigation technique - the contentious VM is moved to another core, disk or network channel depending on the saturated resource. By applying this mitigation method, the workload performance is improved by a factor of 2 on average, with the performance improvement being higher for CPU and I/O contention and lower for LLC contention. Proctor has been tested on a large datacenter setup with 2560 servers and 12800 VMs. With the proposed sampling rate of 6.5%, Proctor occupies less than 0.5% of the available servers.

9.4 Pythia

Pythia [31] is a datacenter scheduling manager that estimates the combined contention created when a LC workload runs concurrently with multiple BE applications. The authors of this scientific work observe that summing the expected contention caused by each BE workload is a naive approach and leads to contention overestimations, especially as the number of colocated applications increases. The naive approach fails, because the colocated BE workloads interfere with and throttle each other, thus reducing the combined contention on shared resources.

Pythia proposes a simple linear regression model to predict the combined contention from multiple colocate workloads. This model measures initially the contention caused by each BE workload when it runs concurrently with the LC one. Then it estimates a weight for each LC workload that shows the amount of prevailing contention when other LC workloads are allowed to run concurrently. This weight depends on the number (but not the combination) of colocated BE workloads and larger values show resistance to inference. Due to the very large number of possible combinations, only a small fraction of them (around 5%) are actually profiled; the rest are predicted using least square optimization to minimize the prediction error. During the profiling the only metric collected is IPC of the LC application.

The steps described before are completed offline when possible, because for every LC workload 195 seconds are needed on average. For this reason, Pythia is effective if the workloads are repetitively submitted for scheduling. When a known workload is submitted, the predictive model is used to determine which cluster nodes will not suffer from performance degradation if the incoming workload is scheduled there. The workload if finally placed on a node using the best fit algorithm (i.e. the node with the least available shared resources left). Assuming that the predictive model is created offline, the online scheduling overhead can be considered negligible (some milliseconds). During runtime, Pythia adopts PiPo (Phase-in/Phase-out) dynamic mechanism proposed in Bubble-Flux [32] to protect the LC workload from colocated BE applications running temporarily on high load.

The PiPo mechanism is part of the online flux engine. The flux engine controls the execution of scheduled workloads using decision intervals. Each interval consists of a phase-in interval, where the BE workload operates normally and the phase-out interval, where the BE workload is paused. The IPC of the LC application is continuously measured during the decision interval and at its end the average IPC is calculated. This average IPC is compared to the QoS metric and then the phase-in and phase-out intervals are adjusted, so that the average IPC is above the QoS threshold.

A key parameter in evaluating Pythia is the QoS degradation (with reference to the scenario of LC workload running alone on a server) that the datacenter administrator allows. With a 95% QoS policy, Pythia achieves an impressive 99% utilization. If the QoS is relaxed further to 90%, the utilization achieved is 71%. It is worth noting here that the authors deliberately excluded some BE benchmarks that cause very high contention and that the QoS policy violation rate is not reported.

9.5 Seer

Seer [14] is an online, centralized performance debugging system for cloud systems accommodating latency-critical microservices. The authors of this article observe that cloud applications are transitioning from the traditional monolithic design to graphs of loosely-coupled microservices. Maintaining a strict QoS (in terms of throughput or tail latency) is more challenging for microservices than for single-binary applications, because performance degradation in one microservice is going to propagate and amplify across the graph. Furthermore, the QoS constraints for each individual microservice are far more strict than for traditional cloud applications. Except for the changes in software design patterns, datacenter hardware is becoming increasingly heterogeneous due to the introduction of accelerators like GPUs and FPGAs, thus making predictable performance even more challenging.

Seer actively monitors the application running on the cloud using two-level tracing. First, a distributed RPC-level tracing system is designed with a view to recording per-microservice latencies. This system records every incoming request on the network interface controller (NIC), the time it

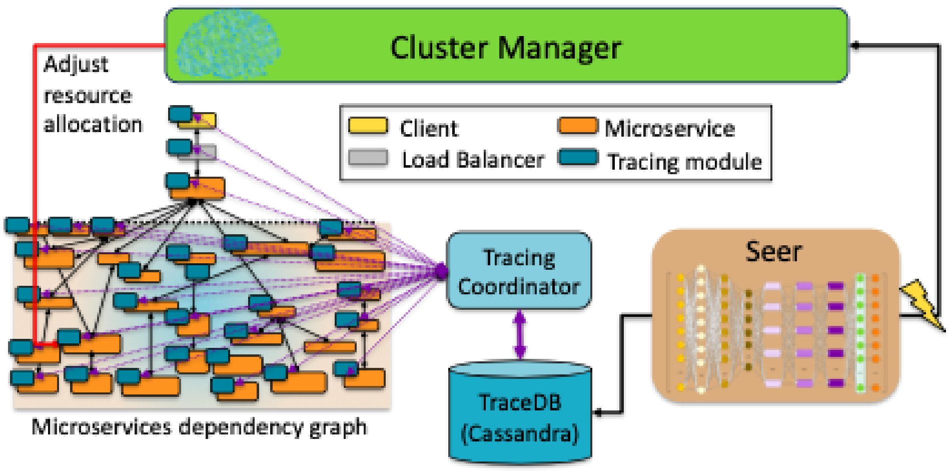


Figure 9.2: Seer architecture [14]

spent waiting on the controller and the processing time (i.e. time until the response arrived on the NIC). Whenever possible, Seer also records any queues created within the microservice; this leads to more detailed tracing, but requires access to the microservice software code which is only seldom available. Second, Seer collects on-demand low-level node diagnostics from hardware counters. This data is used to identify which microservice causes QoS violations and which shared resource has been saturated.

The data collected from the first level of tracing is used to train a deep neural network (DNN). The number of input and output neurons is equal to the number of microservices traced before. The DNN receives as input the NIC queue depths and gives the probability of a given microservice to cause a QoS violation as output. The DNN consists of a set of convolutional layers (CNN) followed by a set of long-short term memory layers (LSTM). The CNNs are effective at reducing dimensionality of large datasets and finding spatial patterns, while LSTMs are good at finding patterns in time. Seer leverages both types of neural networks, because it aims at finding both spatial (i.e. identifying problematic clusters of neighboring microservices) and temporal (i.e. using past QoS violations to forecast future ones) patterns. The training of the DNN is time consuming, happens once and gives best accuracy when 100 - 200 GB of tracing data is consumed. In case that the microservice is slightly altered, a short retraining takes place, and when the microservice or datacenter architecture changes significantly, the DNN needs to be trained from the scratch.

During runtime, Seer signals potential QoS violations and pinpoints the responsible microservice, using data from the trained DNN. At that point, the second level of tracing is activated and Seer starts monitoring the utilization of shared resources on the node that host the problematic microservice. The monitored resources include CPU, memory capacity and bandwidth, network bandwidth, LLC and disk I/O. In public clouds hardware counters are not available; instead Seer injects benchmarks stressing a specific shared resource and monitors the behaviour of the microservice. Once the saturated resources have been detected, Seer notifies the cluster manager who in turn takes actions like resizing the container of the problematic microservice, activating LLC partitioning and using `qdisc` to partition network bandwidth.

Seer has been evaluated on a small private cluster and large-scale public clouds. For the evaluation process, the authors designed four end-to-end microservice applications implementing a social network, a media service, an e-commerce website and a banking system, as well as a hotel reservation system based on Go-microservices [15] architecture. The experiments show that Seer manages to foresee 91% of QoS violations and avoid 84% of them. The mispredicted QoS violations correlate with application updates. The authors also try to optimize the DNN training utilizing the hardware accelerators available in public clouds and reach that conclusion that both Tensor Processing Units (TPUs) on Google Cloud Engine [19] and FPGA-based Brainwave on Microsoft Azure [4] manage to boost up the DNN performance significantly.

Chapter 10

Resource Manager Architecture

10.1 Problem definition

This thesis proposes an online resource manager for Kubernetes. The problem this resource manager is going to tackle is the colocation of one latency-critical (LC) deployment with one best-effort (BE) deployment on the same cluster node. A latency-critical deployment has to meet a strict service level agreement (SLA) on tail latency. For the purpose of this thesis, the SLA is based on the 99%-quantile latency of HTTP requests; this definition of the SLA can however be easily adjusted to LC application's needs. On the other hand, a best-effort deployment has loose quality of service restrictions.

10.2 Outline

The proposed resource manager checks periodically the performance of the applications running in the cluster utilizing metrics from Prometheus monitoring tool API. It tries to effectively colocate a BE and a LC application by allocating a variable number of vCPUs to them. Initially the available vCPUs on a cluster node are split equally between the two applications. In case that a LC deployment violates the SLA, one more vCPU is allocated to it. On the contrary, if the SLA is easily met, then one vCPU is taken from the LC deployment and is allocated to the BE one. This procedure stops when the SLA status changes (i.e. from violated to satisfied or from satisfied to violated after rolling back to the previous state). A pseudo-code implementation of this algorithm is presented as Algorithm 1.

The proposed resource manager works with the following restrictions:

- BE deployments must carry the label "class = best-effort" and the LC deployments the label "class = latency-critical". Deployments without this label will not be changed by the resource manager.
- There must be at most one BE and at most one LC deployment on each cluster node.
- The containers running in the cluster must have distinct names.
- Each deployment must create only one pod replica. If two replicas are desired, then two distinct deployments can be created. The deployments could get connected with a service, with also acts as a load balancer.

10.3 Implementation

The proposed resource manager has been implemented using the Go programming language. The specific programming language has been chosen, because Kubernetes itself is written in Go and exposes a programming interface in Go. This API is called client-go and can be found on GitHub [5].

Algorithm 1: Resource manager algorithm

```
for each node in cluster do
    Collect node, deployment and pod data;
    if resources not allocated then
        Split vCPUs equally between LC and BE deployment;
        Mark the node as not converged;
    else if node not marked as converged then
        Collect SLA data;
        if SLA is not satisfied then
            if BE can be shrunk then
                Shrink BE and grow LC;
            else
                Mark node as converged;
            end
            if SLA was satisfied in the previous state then
                Mark node as converged;
            end
        else
            if SLA was violated in the previous state then
                Mark node as converged;
            else
                if LC can be shrunk then
                    Shrink LC and grow BE;
                else
                    Mark node as converged;
                end
            end
        end
    end
end
```

First of all the resource manager connects to the Kubernetes API and retrieves data about the nodes in the cluster and the running deployments and pods. This connection is possible using the code in listing 10.1. The code uses the `clientcmd` package in order to read and parse information about the cluster (e.g. node name, credentials). Next, the `kubernetes.NewForConfig` command builds a client set, which contains clients for all native Kubernetes resources. From this collection of clients, the core group (which allows access to the cluster nodes and pods) and the apps group (which contains the deployments) are selected.

```go
func connectToAPI() (coreV1API corev1.CoreV1Interface, appsV1API appsv1.AppsV1Interface) {
    kubeconfig := flag.String("kubeconfig", "~/.kube/config", "path to kubeconfig")
    flag.Parse()
    config, err := clientcmd.BuildConfigFromFlags("", *kubeconfig)
    if err != nil {
        panic(err.Error())
    }
    clientset, err := kubernetes.NewForConfig(config)
    if err != nil {
        panic(err.Error())
    }
    coreV1API = clientset.CoreV1()
    appsV1API = clientset.AppsV1()
    return
}
```

Listing 10.1: Connect to Kubernetes API

Using the client returned by the `connectToAPI` function, the resource manager can retrieve the required data about the cluster and the applications running in it. The collected information is stored in map data types and include the node names and IPs, the pod names, statuses, labels and resource lists, as well as a deployment list and the containers belonging to each deployment. The above data is gathered by the functions presented in listing 10.2. Mapping the pods to the deployments they are managed by has been a challenging task, as it has been achieved through the containers. To be more specific, two mappings were created and combined accordingly: the pod each container belongs to and the deployment each pod is managed by.

```go
func getNodePodData (coreAPI corev1.CoreV1Interface)
                    (IPMap map[string]string,
                     podMap map[string][]string,
                     resourceMap map[string][]apicorev1.ResourceList,
                     labelMap map[string]string,
                     containerMap map[string]string) {

    IPMap = make(map[string]string)
    ...

    nodeNameList := []string{}
    nodeList, err := coreAPI.Nodes().List(metav1.ListOptions{})
    for _,node := range nodeList.Items {
        nodeNameList = append(nodeNameList, node.Name)
        IPMap[node.Name] = string(node.Status.Addresses[0].Address)
    }

    namespace := "default"
    for _,nodeName := range nodeNameList {
        podList, err := coreAPI.Pods(namespace).List(metav1.ListOptions{FieldSelector: "spec.
    nodeName=" + nodeName})

        for _,pod := range podList.Items {
            labelMap[pod.Name] = pod.Labels["class"]
            podMap[nodeName] = append(podMap[nodeName], pod.Name)
            containerMap[pod.Name] = pod.Spec.Containers[0].Name
            podResources := pod.Spec.Containers[0].Resources
            resourceMap[pod.Name] = append(resourceMap[pod.Name], podResources.Requests)
            resourceMap[pod.Name] = append(resourceMap[pod.Name], podResources.Limits)
        }
    }
    return
}

func getContainerMap (appsAPI appsv1.AppsV1Interface)
                      (containerDeploymentMap map[string]string) {

    containerDeploymentMap = make(map[string]string)

    namespace := "default"
    deploymentList, err := appsAPI.Deployments(namespace).List(metav1.ListOptions{})

    for _,deployment := range deploymentList.Items {
        for _,container := range deployment.Spec.Template.Spec.Containers {
            containerDeploymentMap[container.Name] = deployment.Name
        }
    }
    return
```

Listing 10.2: Retrieve the necessary cluster data

The next step is querying the Prometheus API about the tail latency of the LC application. Function getFromPromQuery (which is presented in listing 10.3) receives as input a query in string format, queries the Prometheus API and returns the result. The type of the result is model.Value, which is a Prometheus internal interface for values returned by query evaluation. This actual result can be a matrix, a vector, a scalar or a string.

Initially getFromPromQuery creates a new client for the Prometheus API. In the cluster used for the experiments of this thesis, Prometheus exposes the gathered metrics at the address 192.168.122.91 and port 9090. The query to Prometheus API is basically a HTTP request, which the Go programming language handles with the context package [1]. getFromPromQuery declares that the HTTP requests has a timeout of 10 seconds and gets cancelled if it is not served on time.

```go
func getFromPromQuery(query string) (result model.Value) {
    client, err := api.NewClient(api.Config{
        Address: "http://192.168.122.91:9090",
    })
    if err != nil {
        fmt.Printf("Error creating client: %v\n", err)
        panic(err.Error())
    }
    api := promv1.NewAPI(client)
    ctx, cancel := context.WithTimeout(context.Background(), 10*time.Second)
    defer cancel()
    result, warnings, err := api.Query(ctx, query, time.Now())
    if err != nil {
        fmt.Printf("Error querying Prometheus: %v\n", err)
        panic(err.Error())
    }
    if len(warnings) > 0 {
        fmt.Printf("Warnings: %v\n", warnings)
    }
    return
}
```

Listing 10.3: Query Prometheus API

The resource manager iterates over the cluster nodes and collects information about them, the deployments and the pods running on them as described before. If there is one or no pods on the node, no actions are taken. Otherwise, if there are two pods on the node and at least one of them is new, then the available vCPUs are split equally between them (as described in listing 10.4.)

```go
if !resourcesAllocated {
    deploymentNeedsUpdate = true
    latencyCriticalCPU = resource.NewMilliQuantity(availableCores * 500, resource.DecimalSI)
    bestEffortCPU = resource.NewMilliQuantity(availableCores * 500, resource.DecimalSI)
    fmt.Println("Resources are not allocated for all pods, going to split them equally")
    fmt.Println("Best effort pod vCPUs:", bestEffortCPU)
    fmt.Println("Latency critical pod vCPUs:", latencyCriticalCPU)
}
```

Listing 10.4: Allocate vCPUs equally between new pod

In case there are two pods on the node under investigation and none of them is new, the resource manager queries Prometheus about the 99%-quantile latency of the LC application during the last 30 seconds. Half a minute is a large enough time frame and thus gives statistically meaningful data. This information can be retrieved with the following query:

```
histogram_quantile(0.99, rate(flask_http_request_duration_seconds_bucket{method="POST",status
    ="200"}[30s])
```

Taking into account the answer from Prometheus API and the SLA, the resource manager decides whether the SLA is violated or not and stores it at the isSLAMet variable. Then the algorithm 1 can be reinforced in order to split the available vCPUs optimally. The implementation should mark a node as converged when the desired state is reached, with a view to avoiding oscillations between neighboring states. For this reason a convergenceData struct (see listing 10.5) is declared per node.

```
type convergenceData struct {
    converged bool
    previousState string
    convergenceTimestamp time.Time
}
```

Listing 10.5: convergenceData struct

When the SLA is violated, one vCPU is taken from the BE application and is given to the LC. If, nevertheless, there is only one vCPU allocated to the BE application, no changes are made and the algorithm converges. And if in the previous state the SLA was satisfied, then the node deployments roll back to the previous state and the algorithm converges. This procedure is done with the code in listing 10.6.

```
step := resource.NewMilliQuantity(1000, resource.DecimalSI)
if !isSLAMet {
    newLatencyCriticalCPU := *latencyCriticalCPU
    (&newLatencyCriticalCPU).Add(*step)
    if (&newLatencyCriticalCPU).Cmp(*availableCPU) < 0 {
        *latencyCriticalCPU = newLatencyCriticalCPU
        bestEffortCPU.Sub(*step)
        deploymentNeedsUpdate = true
        *nodeConvergeData = convergenceData{false, "violated", time.Now()}
        fmt.Println("SLA is violated, going to restrict best-effort pod")
        fmt.Printf("New best effort pod CPU: %v\n", *bestEffortCPU)
        fmt.Printf("New latency critical pod CPU: %v\n", *latencyCriticalCPU)
    } else {
        deploymentNeedsUpdate = false
        *nodeConvergeData = convergenceData{true, "violated", time.Now()}
        fmt.Println("SLA is violated, but no further actions can be taken")
    }

    if nodeConvergeData.previousState == "SLAMet" {
        *nodeConvergeData = convergenceData{true, "violated", time.Now()}
        fmt.Println("Going to roll back to previous state and converge")
    }
}
```

Listing 10.6: Actions taken for SLA violations

A similar procedure is followed for the case that the SLA is satisfied (listing 10.7). If in the previous state the SLA was violated, then the optimal state has been reached and no more changes are necessary. Else, when possible, one more vCPU is allocated to the BE application. Otherwise there is convergence.

```
if isSLAMet {
    if nodeConvergeData.previousState == "violated" {
        *nodeConvergeData = convergenceData{true, "SLAMet", time.Now()}
        fmt.Println("SLA is now met and the algorithm converged")
    } else {
        newBestEffortCPU := *bestEffortCPU
        (&newBestEffortCPU).Add(*step)
        if (&newBestEffortCPU).Cmp(*availableCPU) < 0 {
            *bestEffortCPU = newBestEffortCPU
            latencyCriticalCPU.Sub(*step)
            deploymentNeedsUpdate = true
            *nodeConvergeData = convergenceData{false, "SLAMet", time.Now()}
            fmt.Println("SLA is easily met, going to grow best-effort pod")
            fmt.Printf("New best effort pod CPU: %v\n", bestEffortCPU)
            fmt.Printf("New latency critical pod CPU: %v\n", latencyCriticalCPU)
        } else {
            deploymentNeedsUpdate = false
            *nodeConvergeData = convergenceData{true, "SLAMet", time.Now()}
            fmt.Println("SLA is easily met, but the best-effort pod cannot be grown")
        }
    }
}
```

Listing 10.7: Actions taken for SLA satisfaction

After all, the variable `deploymentNeedsUpdate` is true, if and only if the deployments need to be updated. Despite Kubernetes allowing the update of an existing deployment, this method is not suitable for the proposed resource manager. By updating the deployment, a new pod is created and the old one is destroyed after the new is successfully placed on the node. But all the available vCPUs of the node are allocated to the old pods and as a consequence the new pods cannot be scheduled and remain on pending mode forever. The approach that was finally adopted is explicitly destroying each old deployment and creating a new one with the same characteristics, except for the resources. This method is implemented by the function `createNewDeployments`, which is shown in listing 10.8.

```go
func createNewDeployments(api appsv1.AppsV1Interface,
                          bestEffortName string, bestEffortCPU resource.Quantity,
                          latencyCriticalName string, latencyCriticalCPU resource.Quantity,
                          nodeName string) {
    namespace := "default"
    deletePolicy := metav1.DeletePropagationForeground
    deploymentsClient := api.Deployments(namespace)

    /* Get old deployments, keep their specifications and delete them */
    tmpDeployment,err := deploymentsClient.Get(bestEffortName, metav1.GetOptions{})
    bestEffortDeployment := tmpDeployment.DeepCopy()
    tmpDeployment,err = deploymentsClient.Get(latencyCriticalName, metav1.GetOptions{})
    latencyCriticalDeployment := tmpDeployment.DeepCopy()
    deleteOptions := metav1.DeleteOptions{PropagationPolicy: &deletePolicy,}
    err = deploymentsClient.Delete(latencyCriticalName, &deleteOptions)
    err = deploymentsClient.Delete(bestEffortName, &deleteOptions)

    /* Change the resources of the deployments */
    resourceList := make(apicorev1.ResourceList)
    resourceList[apicorev1.ResourceCPU] = bestEffortCPU
    bestEffortDeployment.Spec.Template.Spec.Containers[0].Resources.Limits = resourceList
    bestEffortDeployment.Spec.Template.Spec.Containers[0].Resources.Requests = resourceList

    resourceList := make(apicorev1.ResourceList)
    resourceList[apicorev1.ResourceCPU] = latencyCriticalCPU
    latencyCriticalDeployment.Spec.Template.Spec.Containers[0].Resources.Limits = resourceList
    latencyCriticalDeployment.Spec.Template.Spec.Containers[0].Resources.Requests =
    resourceList

    /* Make sure that the new pods will be scheduled to the same node as the old ones */
    nodeSelectorMap := make(map[string]string)
    nodeSelectorMap["kubernetes.io/hostname"] = nodeName
    bestEffortDeployment.Spec.Template.Spec.NodeSelector = nodeSelectorMap
    latencyCriticalDeployment.Spec.Template.Spec.NodeSelector = nodeSelectorMap

    /* The new deployments have the same labels and specifications as the old ones, but new
    names */
    newBestEffortDeployment := &apiappsv1.Deployment{
        ObjectMeta: metav1.ObjectMeta{
            Name: bestEffortDeployment.Name + "-1",
            Labels: bestEffortDeployment.Labels,
        },
        Spec: bestEffortDeployment.Spec,
    }
    _, err = deploymentsClient.Create(newBestEffortDeployment)
    newLatencyCriticalDeployment := &apiappsv1.Deployment{
        ObjectMeta: metav1.ObjectMeta{
            Name: latencyCriticalDeployment.Name + "-1",
            Labels: latencyCriticalDeployment.Labels,
        },
        Spec: latencyCriticalDeployment.Spec,
    }
    _, err = deploymentsClient.Create(newLatencyCriticalDeployment)
}
```

Listing 10.8: Delete old deployments and create new

Before the resource manager moves to the next cluster node, a sleep period is predicted. The aim of the sleep period is to provide some time for the new deployment creation, which is a demanding task for the master node. This period is calculation by the code in listing 10.9, so that all worker nodes are iterated within a minute (in our cluster there are 3 worker nodes and the sleep period is 20 seconds). It may need to be adjusted, according to the number of nodes in cluster, the number of master nodes and their computational power.

```
time.Sleep(60 * time.Second / len(nodeDataMap))
```

Listing 10.9: Sleep period

Chapter 11

Experiment Results

11.1 Latency-critical benchmark

11.1.1 Description

The latency-critical (LC) benchmark is a web application for statistical analysis of the transactions performed by registered and unregistered customers at a supermarket chain with many stores. All the data is stored in a database and the application user can modify the database contents. The application includes a website for data presentation and interaction with the database.

The web application is designed using the Flask [13] micro-framework (version 1.1.2), MySQL database [23] (version 8.0.19) and the Python programming language (version 3.8.0). The Flask code is able to connect to the MySQL database using the python-mysql-connector module.

11.1.2 Database architecture

The database is comprised of five entities: Store, Customer, Product, Category and Transaction. Price is a weak entity, in which the price history of each product can be stored. The above entities have attributes and are connected to each other with relations, as shown in the E-R model in fig. 11.1.

The relational model (fig. 11.2) of the database can be derived from the E-R model. The five strong entities correspond to database tables with primary keys. The weak entity (Price) becomes a table with the combination of barcode (foreign key belonging to Product) and timestamp as the primary key. Many-to-many relations (offers-products and buy-products) become tables as well and their primary keys are the combination of the keys belonging to the strong entities they connect. Finally, relations shop-with-card and shop-without-card do not become database tables; instead Store and Customer primary keys are added to Transaction as foreign keys.

11.1.3 Selected queries

For the default experiment three different URLs of the web application were chosen. Every time an external client requests one of the three URLs, a query to the MySQL database is performed. The fetched data fetched is then formatted and the resulting web page is returned to the client. To make the result presentation easier, the three HTTP requests are named “fast”, “medium” and “slow”. The latency of the requests varies, because each request results in different SQL queries.

- The “fast” request performs a simple selection query:

```
SELECT DATE(start_date), DATE(end_date), amount
FROM Price
WHERE barcode = <selected barcode>
```

- The “medium” request performs a more complex query, which includes a cartesian product, a nested selection subquery and a sorting:

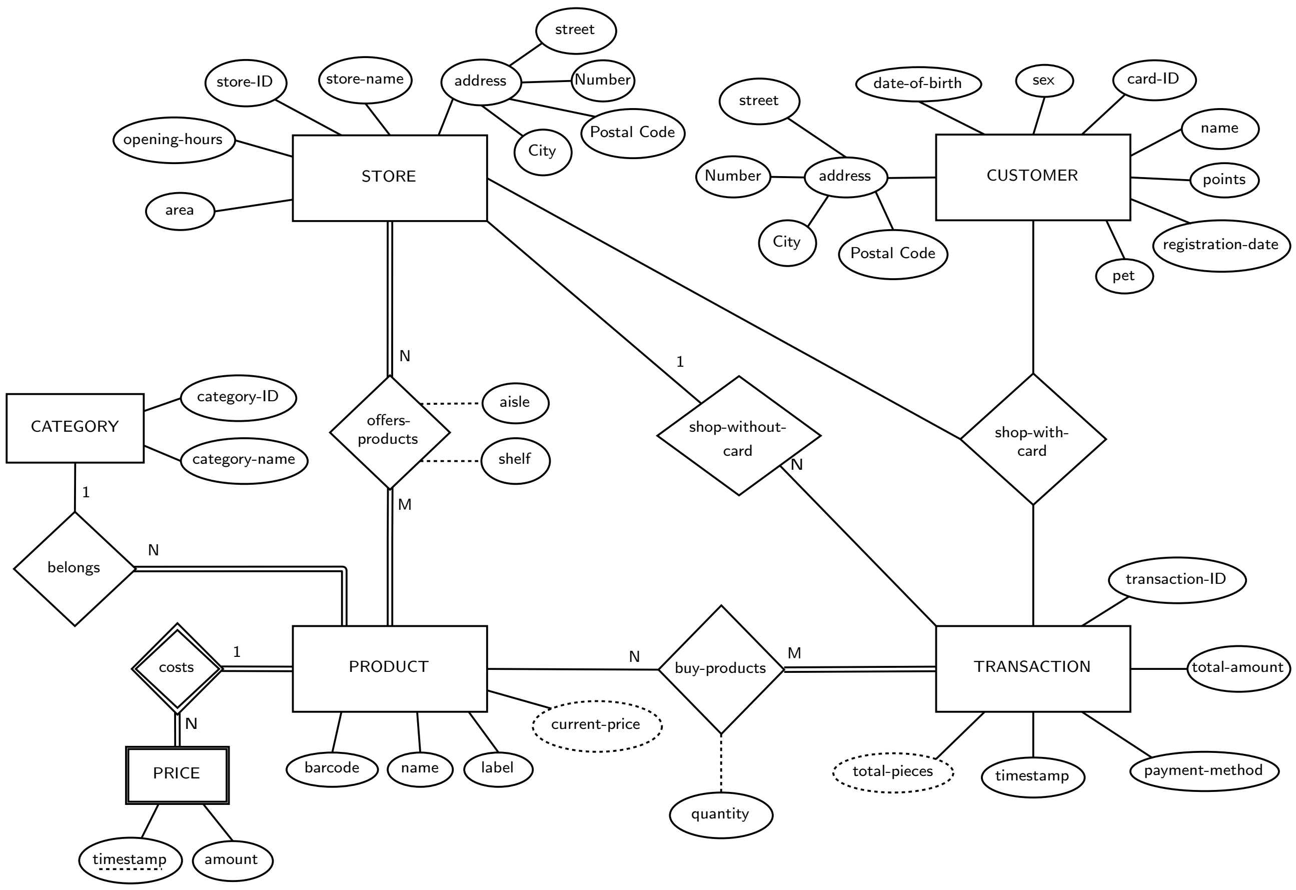


Figure 11.1: Database entity-relation model

```
SELECT P.barcode, P.product_name, sum(B.quantity) AS total_quantity
FROM buy_products AS B
INNER JOIN Product AS P ON B.barcode = P.barcode
AND B.transaction_id IN
(SELECT transaction_id FROM Transaction WHERE card_id = <selected_card>)
GROUP BY P.barcode
ORDER BY total_quantity DESC
LIMIT 10
```

- The “slow” request results in a quite complex query which computes the cartesian product of two tables that come from a natural join each:

```
WITH buy_products_names(barcode, name, transaction_id) AS
( SELECT P.barcode, P.product_name, B.transaction_id
  FROM buy_products AS B
  NATURAL JOIN Product AS P )
SELECT B1.barcode, B1.name, B2.barcode, B2.name, COUNT(*) AS pair_freq
FROM buy_products_names AS B1, buy_products_names AS B2
WHERE B1.transaction_id = B2.transaction_id and B1.barcode < B2.barcode
GROUP BY B1.barcode, B2.barcode
ORDER BY pair_freq DESC
LIMIT 10
```

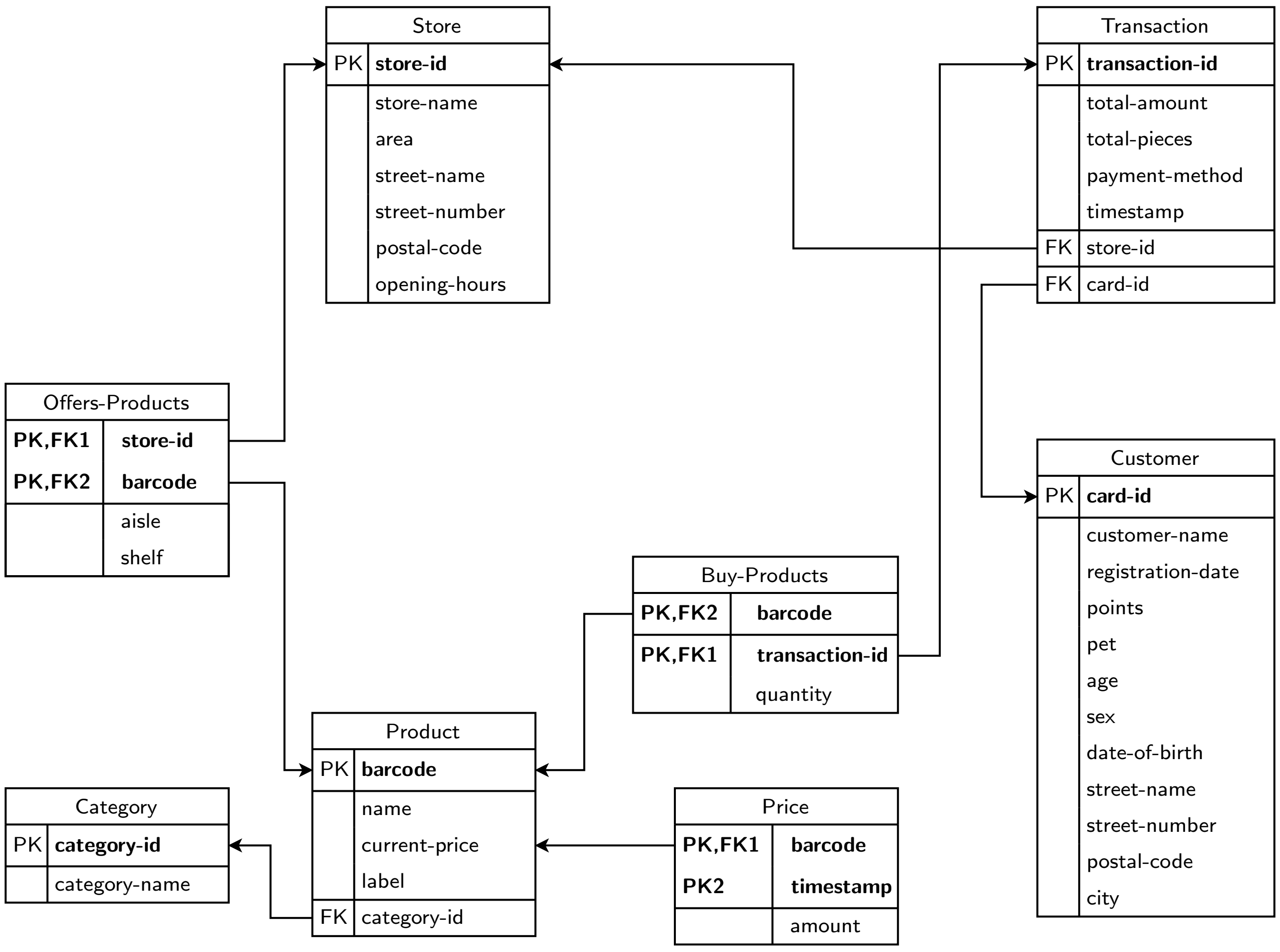


Figure 11.2: Database relational model

11.1.4 Kubernetes deployment

The web application get deployed on the Kubernetes cluster as two separate containers. The database lives on one container on its own and exposes port 3306 (the default MySQL port) so that applications living in other containers can connect to it. A Kubernetes internal service is necessary for fault-tolerance and scalability purposes. The Flask web application is deployed separately and connects to the database through the internal service. The web application should be reachable from outside the cluster, therefore an external service (LoadBalancer in Kubernetes jargon) is created as well.

The database should be able to survive, even if the container it lives on fails and needs to be recreated. Therefore the data should be stored in permanent storage, which is called “persistent volume” in Kubernetes terminology. A persistent volume of 5GB is claimed for the database and the relative container mounts this persistent volume, so as to read data from and write data to the permanent storage. More technical details about the implementation of the deployment can be found in the appendix.

11.2 Best-effort benchmark

Benchmarks from the PARSEC suite [27] [24] have been chosen as the best-effort (BE) tasks that will run along with latency-critical application. The PARSEC benchmarks are multi-threaded and can utilize many physical cores simultaneously. They come from a wide range of scientific domains and are not only focused in the domain of High-Performance Computing (HPC). As a result they can be considered representative of real-world best-effort applications.
The PARSEC suite contains the following benchmarks:

- **blackscholes**: A Black-Scholes partial differential equation solver for option pricing
- **bodytrack**: Body tracking of a person through an image sequence using computer vision algorithms
- **canneal**: Optimization for the routing cost of a chip design using the simulated annealing technique
- **dedup**: Data stream compression using "deduplication" method, which is also used in modern backup storage systems
- **facesim**: Physical simulation of human face motions
- **ferret**: Content-based similarity search, ideal for non-text data types
- **fluidanimate**: Animation of fluid dynamics using the Smoothed Particle Hydrodynamics technique
- **freqmine**: Data mining method for frequent itemset mining
- **raytrace**: Real-time raytracing method frequently used in computer games
- **streamcluster**: An online clustering algorithm for data streams
- **swaptions**: A Monte Carlo simulation which prices a portfolio of swaptions
- **vips**: Image processing application based on the VASARI Image Processing System
- **x264**: An H.264/AVC lossy video encoder

Out of these benchmarks, fluidanimate, streamcluster and swaptions were used for the experiments. With this choice of benchmarks, a data mining, an animation and a simulation application is used as the best-effort load.

11.3 Experiment setup

11.3.1 Experiment description

The aim of the experiments is to measure the latency of the HTTP requests and to monitor the utilization of the node resources under varying traffic scenarios. For the first experiments 100 threads are spawned on the client side and each thread generates an HTTP request per minute (randomly chosen out of the three possible). The number of requests per minute is increased to 250 during the second experiment and to 400 suring the third. Every experiment lasts for 10 minutes and measurements are collected until all threads are finished.

The following experiments have been carried out:

- **Baseline (alone) experiment**
  The first experiment aims at profiling the latency-critical application. The LC application runs alone on one node and it is free to consume all the available resources.
- **Colocated experiment**
  During the colocated experiment, the latency-critical and the best-effort applications run concurrently on the same node. Both applications can utilize as many resources as they wish.
- **Static experiment**
  In this experiment a static resource utilization restriction is imposed. To be more specific, the BE benchmark is pinned to one CPU core, while the LC is allowed to use the rest of the CPU cores.
- **Dynamic experiment**
  The last experiment uses the custom-made resource manager to allocate resources to the LC and the BE applications. Initially the CPU cores are split equally between the applications. The LC application is allocated one more core in each iteration, until the SLA is met or there are no more cores available. If the SLA is easily met from the beginning, the BE benchmark is grown.

11.3.2 Metrics

The performance of LC and the BE applications is continuously monitored during the experiments. The following metrics are collected using Prometheus:

- Histogram of the HTTP request latency for each of the three URLs
- 0.99-quantile latency of the above HTTP requests
- 1-minute moving CPU utilization for each node processor
- 1-minute moving average system load for each node

11.3.3 Infrastructure setup

The experiments were conducted on virtual machines (VM) running Ubuntu Linux 18.04. The VM generation and configuration is controlled by libvirt over Qemu-KVM. The VMs run concurrently on the same physical server and are organized in a Kubernetes cluster (see Appendix), with one virtual machine acting as the master node and the other three as workers. There is also a fifth VM, which does not belong to the cluster and acts as a client that generates HTTP requests to the LC application.

The server is equipped with Intel Xeon X5650 chips running at a clock frequency of 2.67 GHz. The server has 12 physical cores organized in 2 NUMA sockets. Each physical core has 2 hardware threads, so there are 24 logical cores in total. The hypervisor (Qemu-KVM) identifies 24 vCPUs (virtual centralized processing units) and allocates 6 vCPUs to each worker VM and 3 vCPUs to the master and the client nodes.

Every physical core comes with 32KB of L1 instruction cache, 32KB of L1 data cache and 256KB of L2 cache. The L3 cache is 12MB and is shared between the 6 physical cores of each NUMA socket. The main menory of the server is 48GB and it is shared by all physical cores. Each VM is given 8GB of memory.

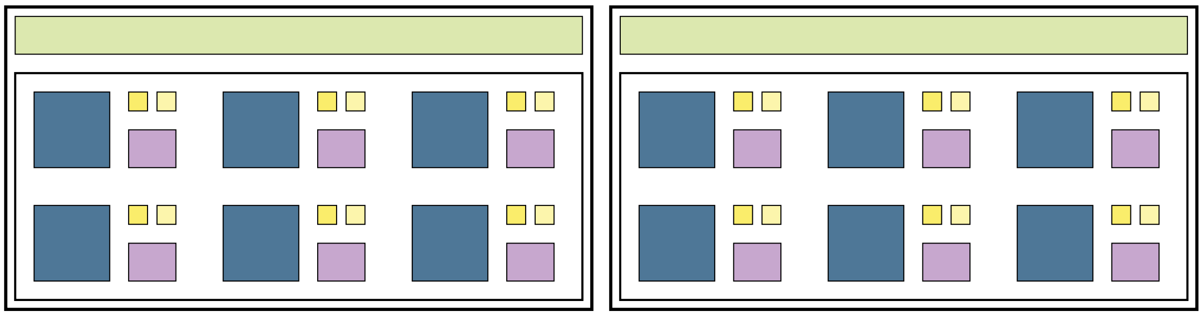


Figure 11.3: Topology of the server. The blue elements are the cores, the yellow are L1I and L1D caches, the violet are L2 caches and the green are L3 caches. The server consists of 2 NUMA sockets.

11.3.4 Service Level Agreements

Latency-critical applications operate with strict service level agreements (SLA) on tail latency. The SLA in these experiments is determined with reference to the 99%-quantile latency achieved during the ideal execution - when the LC application run on the node alone. The SLA is determined by multiplying the ideal 99%-quantile with the following coefficients:

- 1,50 if the traffic is low (i.e. 100 requests per minute)
- 1,75 if the traffic is in between (i.e. 250 requests per minute)
- 2,00 if the traffic is high (i.e. 400 requests per minute)

	100 RPM	250 RPM	400 RPM (node)	400 RPM (cluster)
fast	0,075	0,090	0,100	0,100
medium	0,40	0,45	1,00	0,50
slow	3,00	3,50	6,00	5,00

Table 11.1: SLA for each HTTP request and traffic load (in sec)

11.4 Single-node results

Figure 11.4 shows the number of CPU cores that the resource manager chose to allocate to the latency-critical application in the various colocation scenarios. The number of available cores is 6 and the number of allocated cores ranges from 2 to 5. The resource manager opted for less cores when the traffic load was low and more when the traffic was high, when the LC application run on the same node as streamcluster and fluidanimate. It is worth noting that the resource manager allocated 4 cores to LC benchmark when it was colocated with swaptions benchmark, irrespective of the traffic load.

11.4.1 Average latency results

Bar plots in figure 11.5 show the average 99%-quantile latency of the HTTP requests for all experiments. The bar with the light shade represents the static experiment, the dark shade represents the colocation experiment and the medium shade represents the resource manager experiment. For the resource manager experiment, only the last half of the measurements are taken into consideration due to a transitional phase that will be discussed later. The grey line on the background shows the agreed SLA for each configuration (query and traffic load combination).

The SLA is once violated when the BE benchmark is given only 1 CPU core (4%). But when the resources are freely utilized by both applications, the SLA is violated in 19 out of 27 cases (70%). The resource manager manages to reduce this number to 3 cases out of 27 (11%).

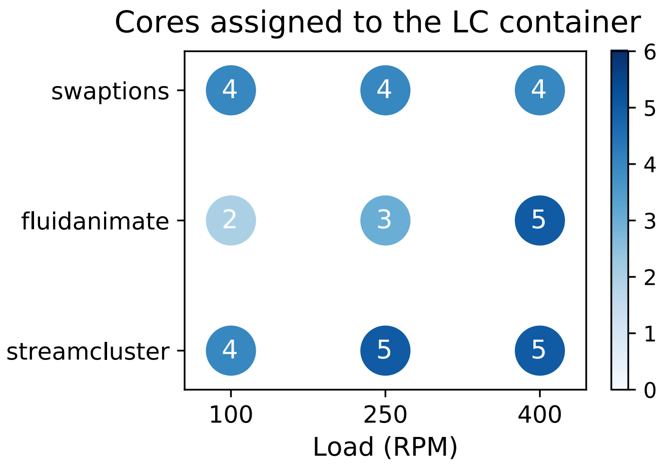


Figure 11.4: Number of cores assigned to the LC benchmark by the resource manager for all colocation combinations

11.4.2 Latency results against time

The resource manager decides on how to allocate the CPU resources using a trial-and-error method. When the number of allocated resources is changed, a new container needs to be created. This procedure takes less than half a minute, but results in some latency spikes during the first minutes of the experiment - this period is called transitional phase. But soon the resource manager algorithm converges, no new containers need to be spawned and the experiment enters its steady phase.

Heatmaps in figures 11.6, 11.7 and 11.8 depict the distribution of the HTTP latency against the experiment time. Each column on the heatmap represents one minute and the rows show the distribution of the latency during this minute. The dot shows in which latency bin the 0.99-quantile latency falls into during this minute.

The first 3 to 4 minutes of the experiment correspond to the transitional phase and the rest to the steady one. During the transitional phase the 99%-quantile latency increases and the latency distribution shows greater variance. But after the resource manager algorithm converges, the 99%-quantile latency decreases quickly and the remains relatively steady.

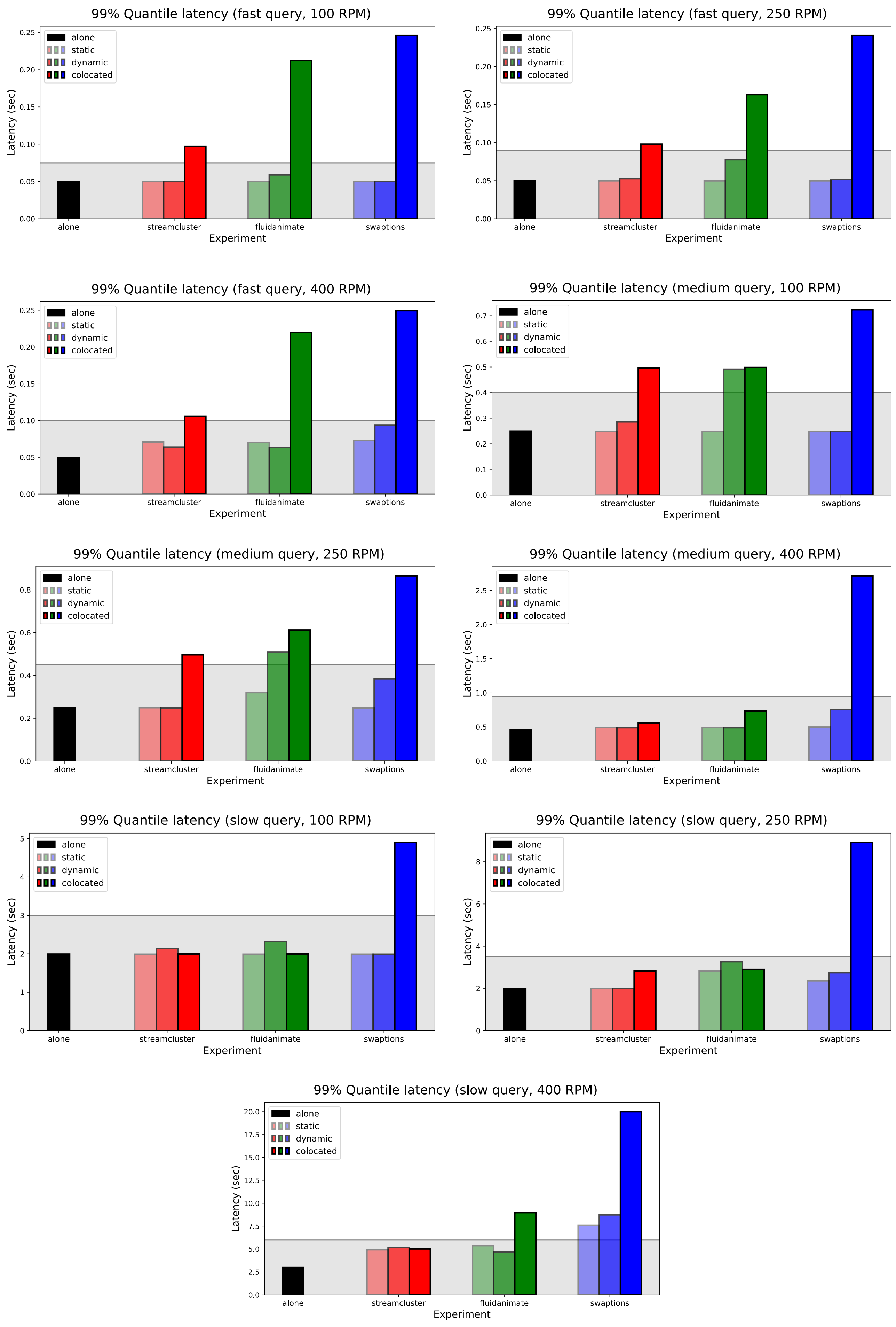
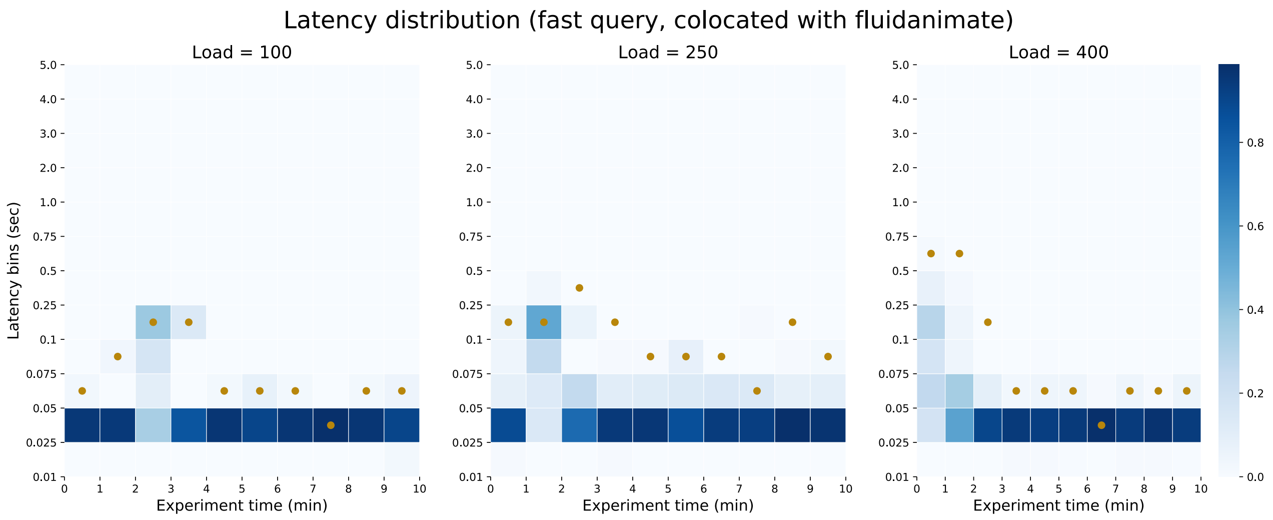
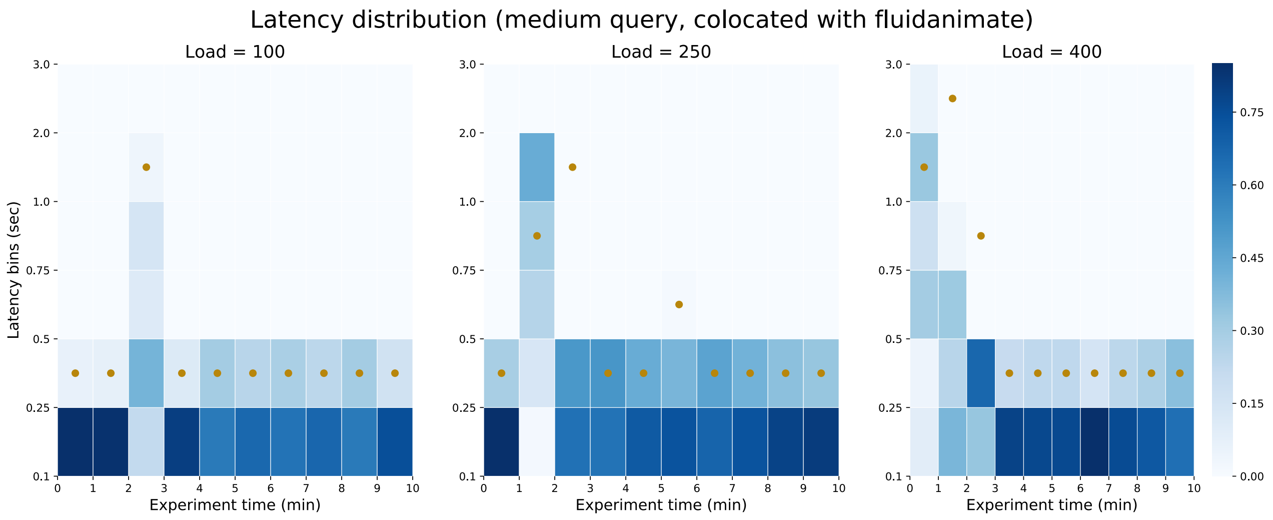


Figure 11.5: 99%-quantile average latency





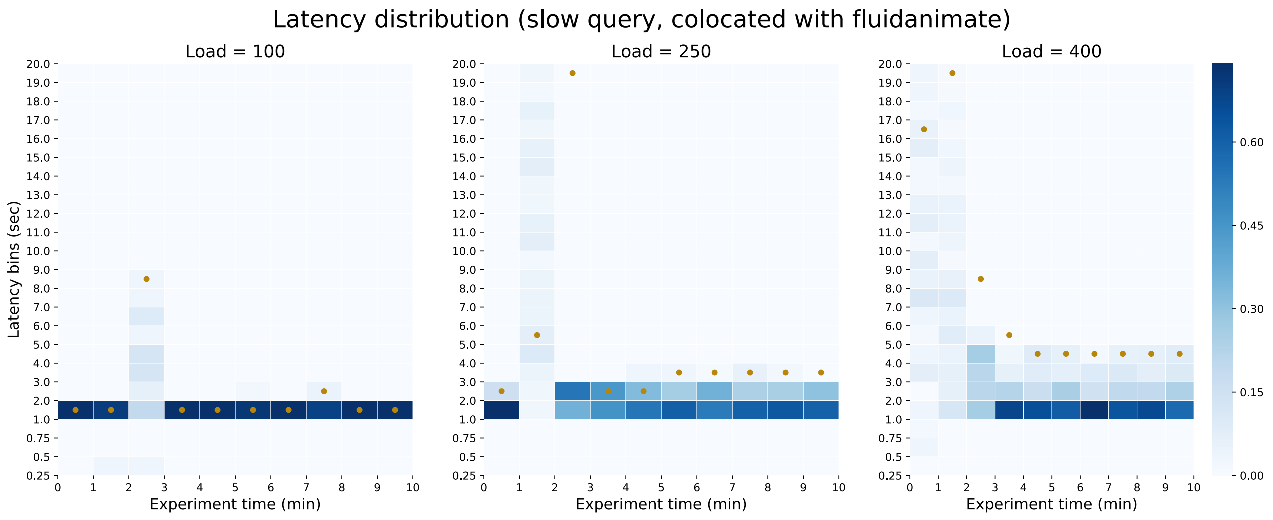


Figure 11.6: Latency distribution against time when LC is colocated with fluidanimate. The dots represent the 99%-quantile latency.

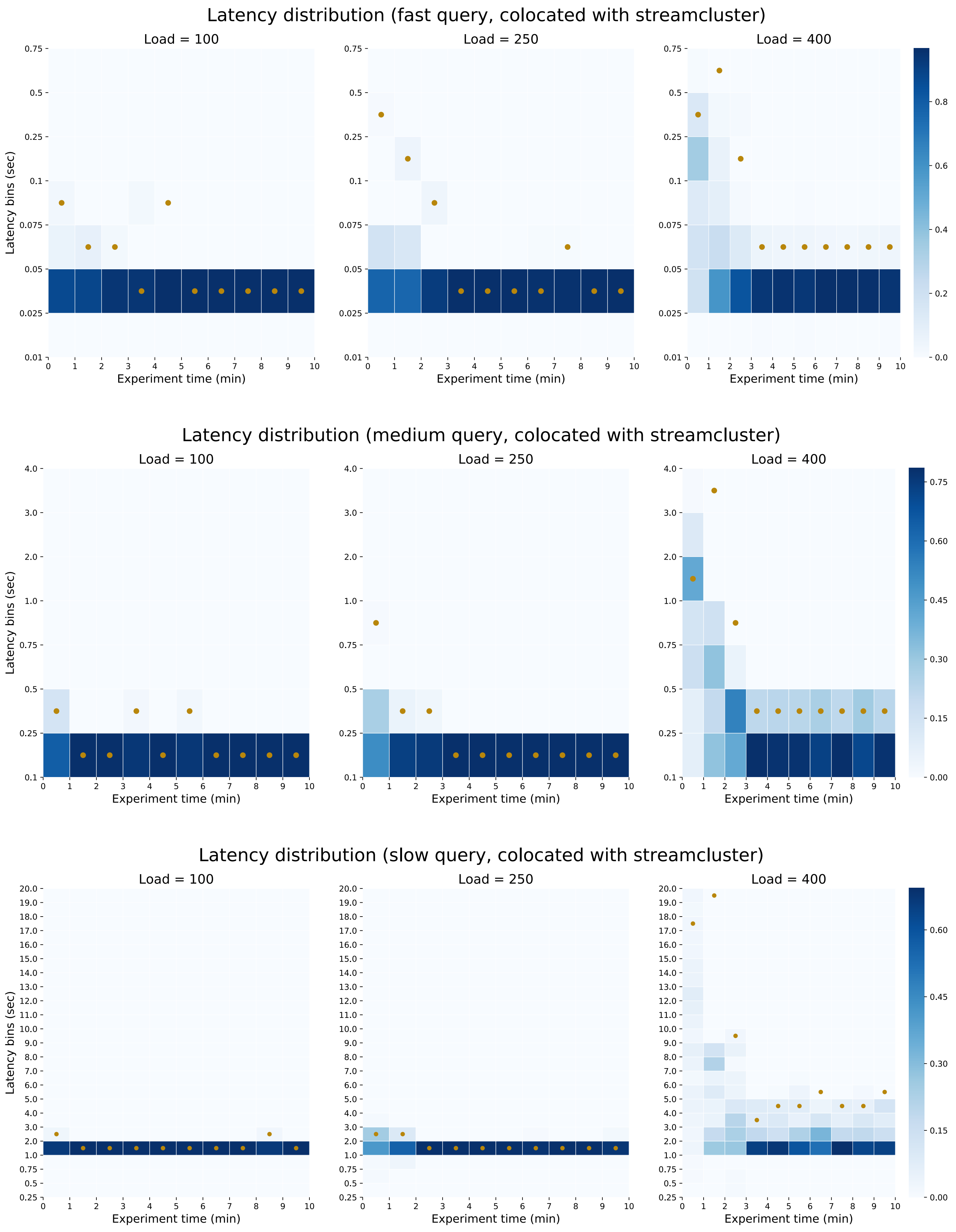


Figure 11.7: Latency distribution against time when LC is colocated with streamcluster. The dots represent the 99%-quantile latency.

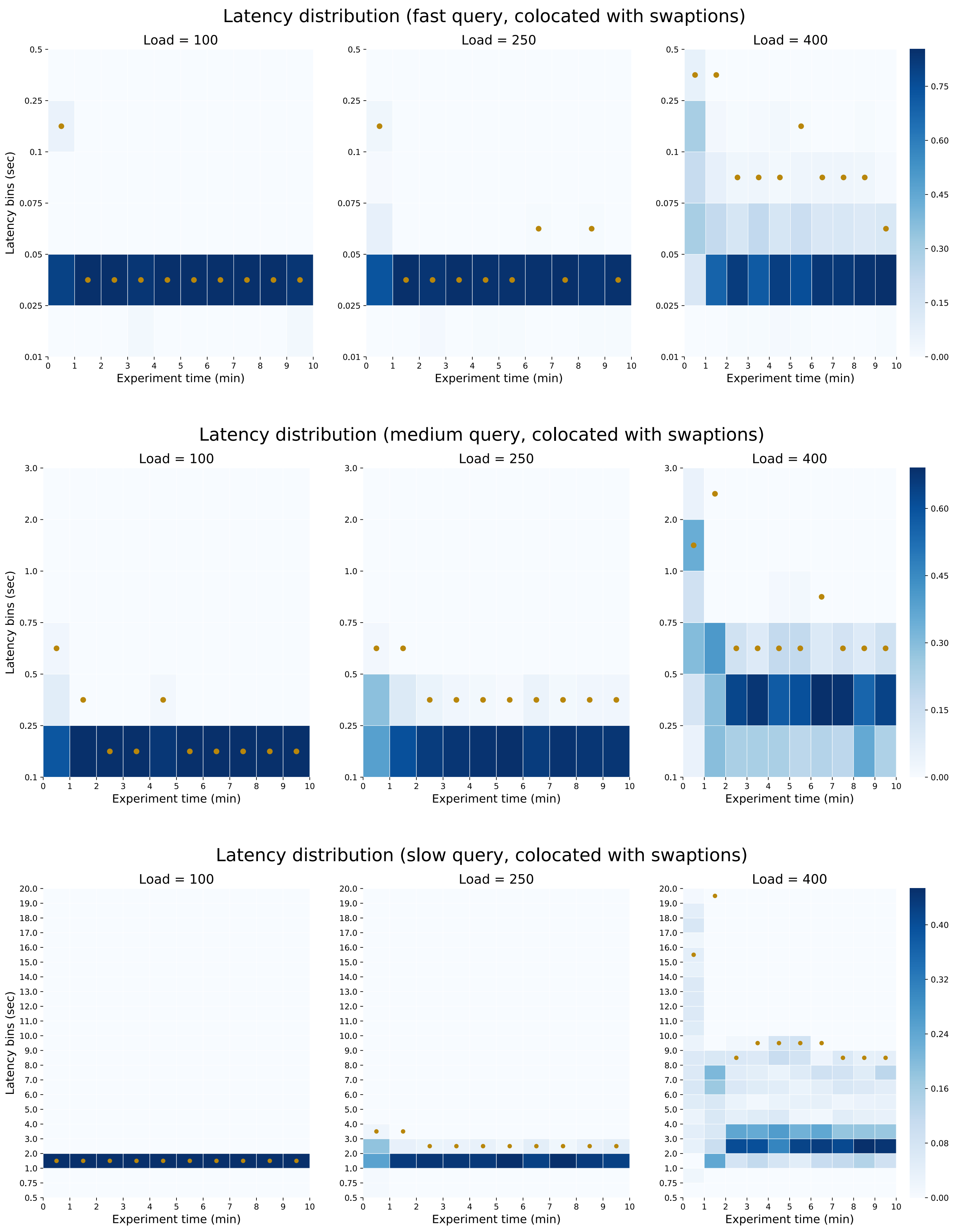


Figure 11.8: Latency distribution against time when LC is colocated with swaptions. The dots represent the 99%-quantile latency.

11.4.3 Best-effort benchmark performance

The performance of the BE benchmarks (fig. 11.9) is calculated with the reference to the ideal execution time (when they run alone on one node and are allowed to take up all the available resources). The following plots show the performance slowdown during the experiments. The y-axis represents the execution time and the number above the dot shows the average slowdown when compared to the ideal execution. In order to compute the average, each benchmark was left to complete 5 full runs. The variance of the measurements is not shown, because it is considered negligible.

The performance of the BE application is best when it is colocated with the LC without any restrictions. However this scenario results in SLA violations. The static configuration restricts the BE application very much and degrades its performance significantly. The resource manager achieves a trade-off with smaller slowdown than the static scenario and only a few SLA violations.

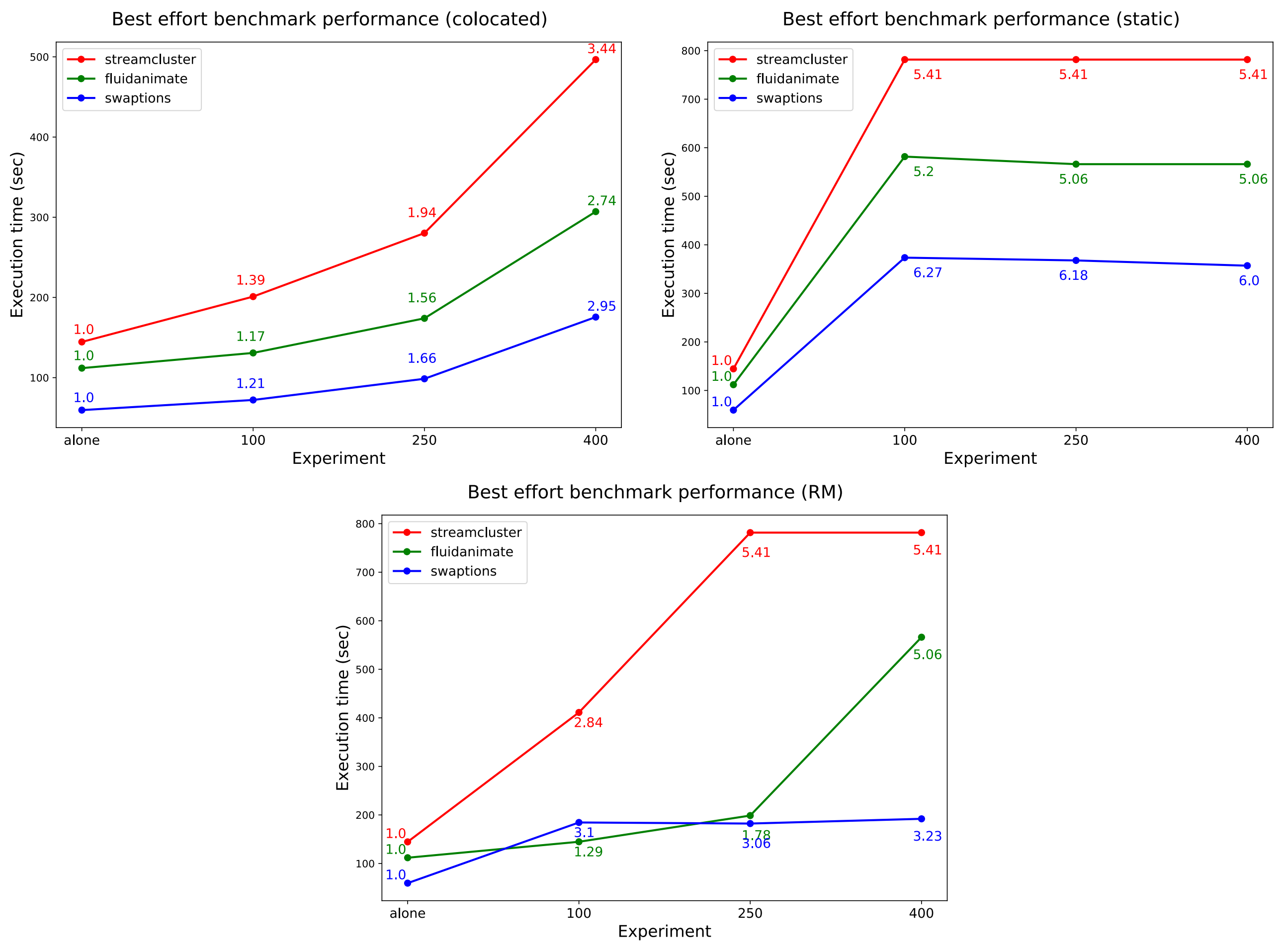


Figure 11.9: BE benchmark performance. The y-axis shows the average execution time and the number above the dots represents the slowdown with regard to the ideal execution.

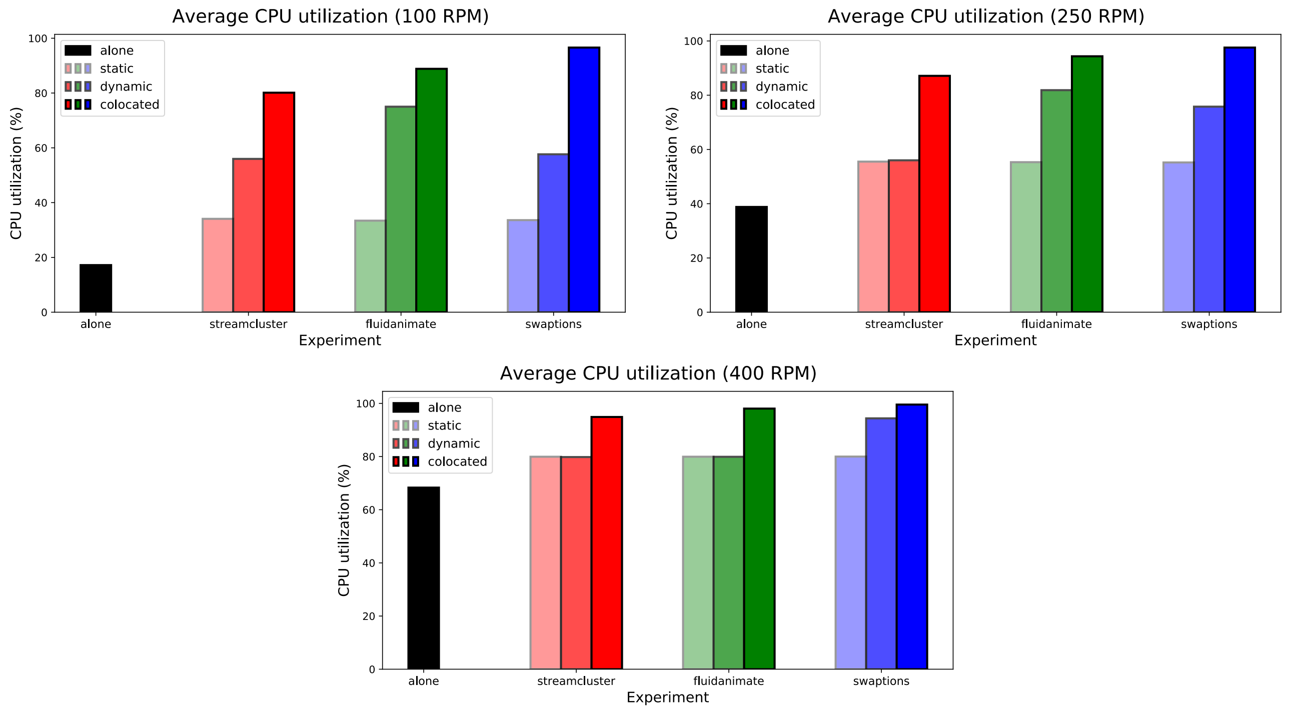


Figure 11.10: Average CPU utilization

11.4.4 Average CPU utilization

Figure 11.10 shows the average CPU utilization during the above mentioned experiments. The CPU utilization is lower during the static experiment and higher during the colocated one. In other words the static configuration underutilizes the available resources in order to achieve the SLA. On the contrary the freely colocated configuration cannot control the CPU utilization and thus violates the SLA. The resource manager achieves a trade-off between resource utilization and latency. Compared to the conservative static allocation, it increases the CPU utilization up to 2,2x, with the average increase being 1,6x.

11.4.5 Average system load

The average system load shows how many tasks are in running or waiting mode. When the system load is 0%, then the system is idle. When the system load is 100%, every processor is occupied by exactly one task in running or waiting mode. When the system load is more than 100%, waiting queues are formed and therefore performance problems may start to appear.

Figure 11.11 shows the average system load during the above mentioned experiments. The average system load is below 50% for most experiments, which means that the CPUs remain idle for more that 50% of the time. It is worth noting that the system load remains low even when the workloads are freely colocated and the SLA violations are more than frequent. This behaviour can be explained by the nature of the workloads: the LC workload is a database generating many I/O time-consuming requests and the presence of queues can severely degrade its performance.

11.5 Multi-node results

The proposed resource manager has also been tested for applications running on multiple nodes. As explained in paragraph 10.2, the resource manager will not work if the LC application is scaled by increasing the number of pod replicas on the same deployment. Instead, more deployments should be created and the incoming traffic should be distributed by the external service. For the purposes of the experiment, a second identical (except for the deployment and container names) deployment is

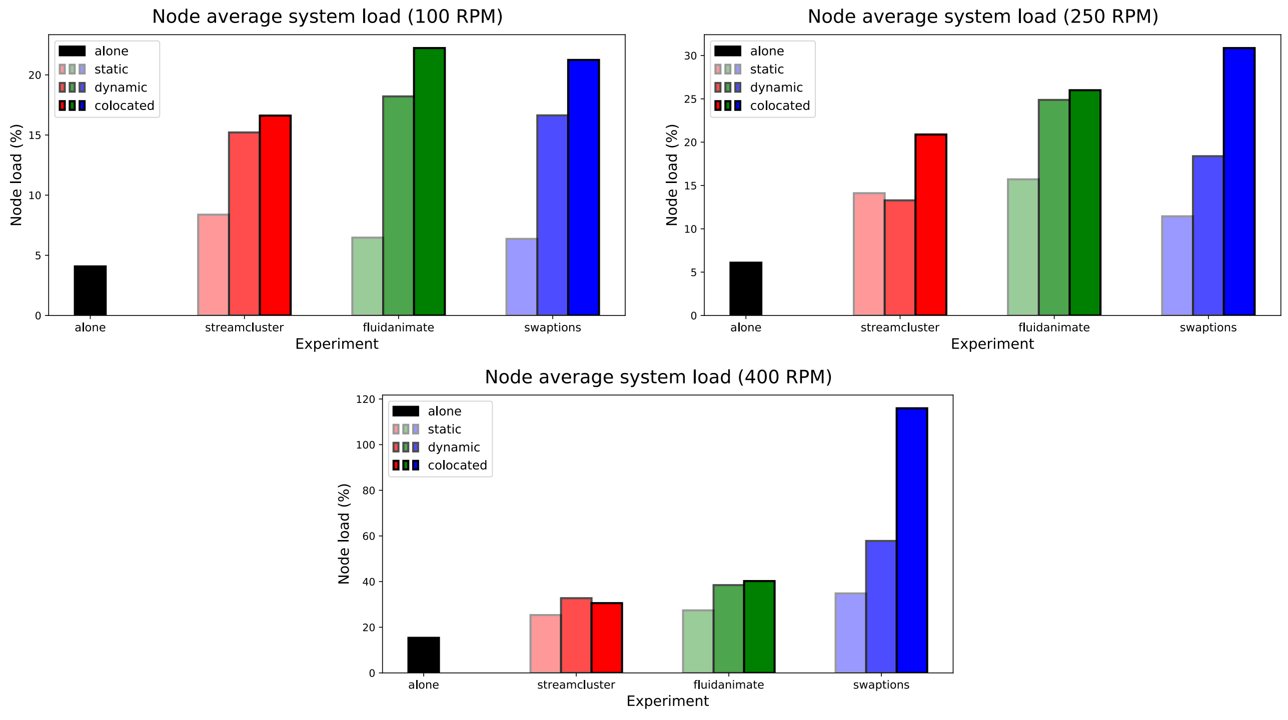


Figure 11.11: Average system load

created, so that the LC application is hosted on nodes `worker-1` and `worker-2`. Moreover two BE deployments are created, with the one on `worker-1` running the `streamcluster` benchmark and the one on `worker-1` running `swaptions`.

During the experiment, 400 requests per minute were being served. The number could not be raised, as the cluster nodes were actually hosted on the same physical machine and larger load led to some of them failing. The increased pressure to the I/O subsystem is to blame for this behaviour. In correspondence to the previous experiments, the experiments was made up of the following scenarios:

- The LC ran initially alone on the nodes ("alone")
- The LC and BE application were colocated freely ("colocated")
- All but one of the available vCPUs of each node was assigned to the LC application and the remaining to the BE ("static")
- The resource manager decided how many vCPUs to allocate to each deployment ("RM")

The resource manager decided to allocate 4 vCPUs to the LC application and 2 to the BE on both nodes. The results are presented in figures 11.12 and 11.13. The "colocated" scenario results once more in increased latency and the resource manager achieves latency one the steady phase comparable to the "static" scenario. Yet, the transitional phase persists although there is a sleep period between the investigation of different nodes by the resource. More efforts are needed, so as to eliminate or minimize this phase.

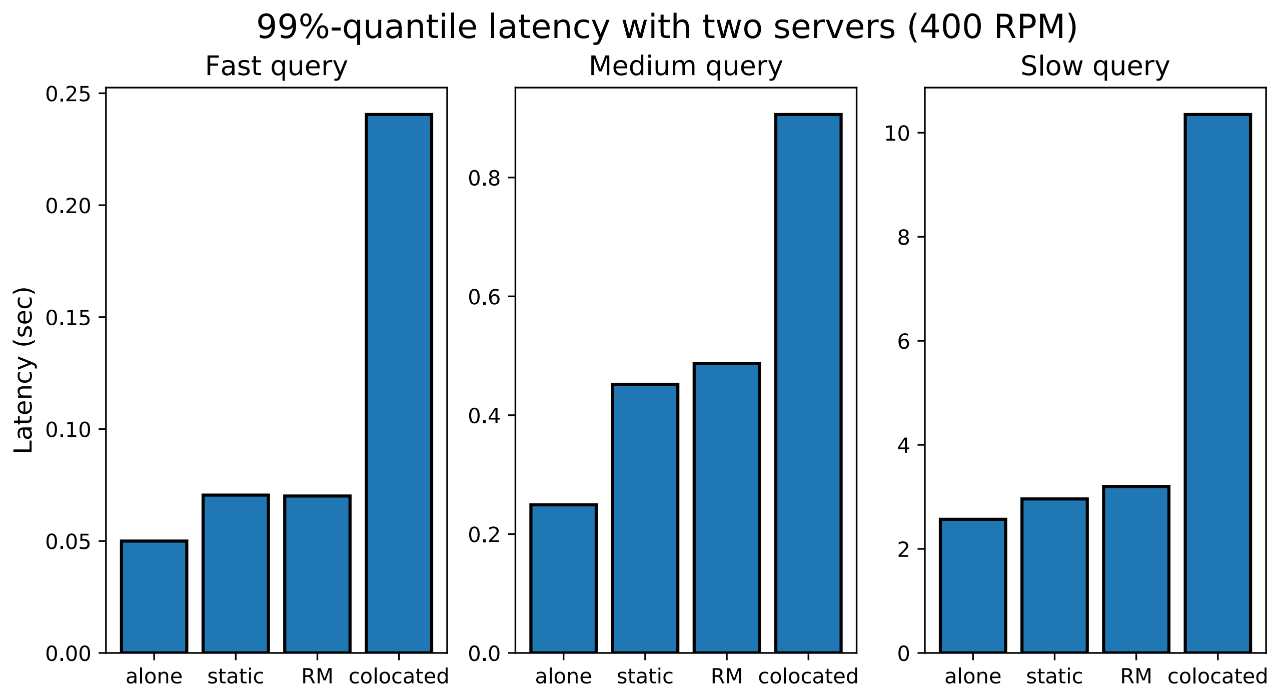


Figure 11.12: Average 99%-quantile latency for different scenarios on the steady phase with two nodes

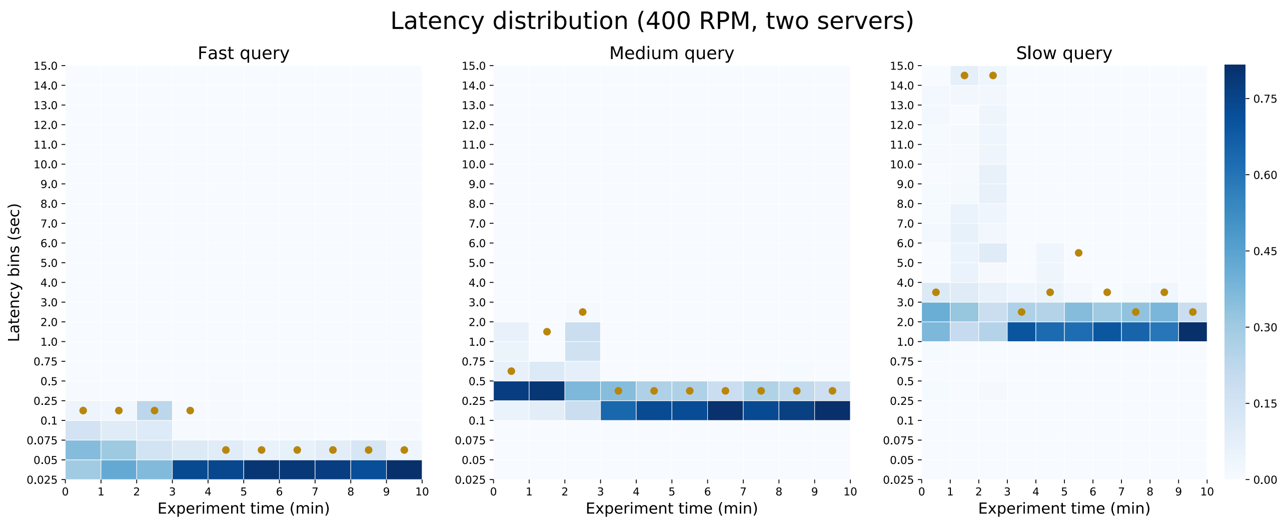


Figure 11.13: Latency distribution against time for the two-node experiment

Chapter 12

Conclusion & Future Work

12.1 Summary

This thesis discusses the topic of efficient colocation of latency-critical workloads with best-effort batch jobs in clusters orchestrated by Kubernetes. First, Docker (a container runtime) and Kubernetes (a container orchestrator) are presented. Moreover, selected scientific reports on this subject are briefly summarized. Finally, this thesis proposes a resource manager for Kubernetes clusters, explains the software design details and evaluates it in a small cluster.

The proposed resource manager follows a performance-centric approach and asks users to define a latency target for their LC applications. The LC workloads should expose latency metrics, so that Prometheus can scrape them. Understanding that hardware counters are not available in public clouds, the resource manager requires no measurements and leverages only latency measurements collected by Prometheus. The resource manager consists of a state exploration algorithm that minimizes the number of vCPUs assigned to the LC workload, with the constraint that the latency target must be met. To avoid oscillations, the algorithm converges when the target satisfaction status changes (i.e. from violated to satisfied or vice versa).

Experiments in a 3-node cluster have been carried out to evaluate the resource manager. The resource manager achieves a target violation rate of 11%, compared to 4% of a conservative static approach and 70% of a liberal one with no restrictions in resource usage. In addition, more cores are assigned to batch jobs in most cases, compared to the static approach and the overall CPU utilization is improved by 1.6x on average. A transitional period appears until the algorithm converges; during this phase, latency temporarily increases. Therefore, the proposed resource manager is better suited for long-running applications.

12.2 Future work

This thesis is endowed with a functional resource manager for Kubernetes clusters, which can be used as a foundation for future expansions. Some ideas for future work are suggested in this section, categorized in development optimizations and relevant research topics.

12.2.1 Development optimizations

The proposed resource manager has been evaluated on a small cluster with 3 nodes. It would be interesting to evaluate its performance on a public cloud cluster with a large number of nodes and diverse workloads. The scalability and computational needs of the resource manager could be established through this experiment.

Furthermore, a database could be used to store the cluster and container data. In the proposed implementation, the container data (e.g. which containers are running on which server and for how long, and the latency metrics from Prometheus) are temporarily stored in variables. Instead, all this tracing data could be permanently stored in a database for future analysis.

12.2.2 Research topics

A topic for future research is the effectiveness of a resource manager that extensively profiles incoming containers before scheduling them. This approach has already been suggested in Quasar [11] and Pythia [31], but has not been tested on Kubernetes clusters. A wise scheduling decision could be valuable, because Kubernetes has no live migration mechanisms allowing disturbing containers to be moved without losing their progress.

Another suggestion is the development of a such migration mechanism for Kubernetes. This would allow the migration of contentious containers to other servers when performance degradation is detected, as proposed in Proctor [20]. Except for this mechanism, one for pausing a container could be engineered as well. Again, Kubernetes does not allow the pausing of a running container without losing its state; the only available solution is scaling the deployment to zero replicas, which results in destroying all the containers.

Appendix A

Kubernetes setup

Various tools (e.g. kubeadm, Kubespray, minikube) can be used to create and manage a Kubernetes cluster. In this thesis kubeadm has been chosen. The necessary packages can be installed by running script A.1.

```
# Run the following script as root

### Install required packages
apt-get update && apt-get install -y apt-transport-https ca-certificates curl software-
    properties-common gnupg2

### Install Docker runtime

# Add Dockers official GPG key
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | apt-key add -

# Add Docker apt repository.
add-apt-repository \
"deb [arch=amd64] https://download.docker.com/linux/ubuntu \
$(lsb_release -cs) \
stable"

# Install Docker CE.
apt-get update && apt-get install -y \
containerd.io=1.2.10-3 \
docker-ce=5:19.03.4~3-0~ubuntu-$(lsb_release -cs) \
docker-ce-cli=5:19.03.4~3-0~ubuntu-$(lsb_release -cs)

# Setup daemon.
cat > /etc/docker/daemon.json <<EOF
{
    "exec-opts": ["native.cgroupdriver=systemd"],
    "log-driver": "json-file",
    "log-opts": {
        "max-size": "100m"
    },
    "storage-driver": "overlay2"
}
   EOF

mkdir -p /etc/systemd/system/docker.service.d

# Restart docker.
systemctl daemon-reload
systemctl restart docker

### Install kubectl
curl -LO https://storage.googleapis.com/kubernetes-release/release/`curl \
-s https://storage.googleapis.com/kubernetes-release/release/stable.txt`/bin/linux/amd64/
    kubectl
chmod +x ./kubectl
sudo mv ./kubectl /usr/local/bin/kubectl

### Install kubeadm
curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | sudo apt-key add -
cat <<EOF | sudo tee /etc/apt/sources.list
deb https://apt.kubernetes.io/ kubernetes-xenial main
EOF
```

```
sudo apt-get update
sudo apt-get install -y kubelet kubeadm kubectl
sudo apt-mark hold kubelet kubeadm kubectl
```

Listing A.1: Install Kubernetes components

After all the required components have been installed, it is time to create the Kubernetes cluster. On the master node kubeadm init command should be executed, while worker nodes join the cluster by run the kubeadm join command. The commands to run on the master can be found on script A.2.

```
# Disable swap
sudo swapoff -a

# Initialize kubeadm
kubeadm init --pod-network-cidr=10.244.0.0/16 >> cluster_initialized.txt

# Make configuration file available to kubelet
mkdir -p $HOME/.kube
sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
sudo chown $(id -u):$(id -g) $HOME/.kube/config

# Install intra-cluster network add-on
# weave-net is used, other alternatives exist
kubectl apply -f "https://cloud.weave.works/k8s/net?k8s-version=$(kubectl version | base64 |
    tr -d '\n')"

# Check if the master node is up
kubectl get nodes

# Follow instructions on cluster_initialized to add worker nodes
```

Listing A.2: Initialize cluster on the master node

Appendix B

Deployment configuration files

B.1 Backend configuration

First a PersistentVolumeClaim (PVC) and PersistentVolume (PV) should be configured, so that there is permanent storage for the database. The PVC and PV are configured using the YAML file in listing B.1.

```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: mysql-pv-volume
  labels:
    type: local
spec:
  storageClassName: manual
  capacity:
    storage: 5Gi
  accessModes:
    - ReadWriteMany
  hostPath:
    path: "/mnt/data"
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mysql-pv-claim
spec:
  storageClassName: manual
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 5Gi
```

Listing B.1: PVC and PV configuration

Then the main deployment can be created using the YAML file ... The deployment mounts the PersistentVolume declared before. Some environment variables need to be declared, so that the container can connect to MySQL database. The required database are stored in a secret file for security reasons. Furthermore, an internal service is configured, so that other containers can have access to the database.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mysql-deployment
  labels:
    app: mysql
spec:
  replicas: 1
  selector:
    matchLabels:
      app: mysql
  template:
    metadata:
      labels:
        app: mysql
    spec:
      containers:
      - name: mysql
        image: mysql
        imagePullPolicy: IfNotPresent
        ports:
        - containerPort: 3306
        env:
        - name: MYSQL_ROOT_PASSWORD
          valueFrom:
            secretKeyRef:
              name: mysql-secret
              key: mysql-root-password
        - name: MYSQL_USER
          valueFrom:
            secretKeyRef:
              name: mysql-secret
              key: mysql-username
        - name: MYSQL_PASSWORD
          valueFrom:
            secretKeyRef:
              name: mysql-secret
              key: mysql-password
        volumeMounts:
        - name: mysql-persistent-storage
          mountPath: /var/lib/mysql
      volumes:
      - name: mysql-persistent-storage
        persistentVolumeClaim:
          claimName: mysql-pv-claim
---
apiVersion: v1
kind: Service
metadata:
  name: mysql-service
spec:
  selector:
    app: mysql
  ports:
    - protocol: TCP
      port: 3306
      targetPort: 3306
```

Listing B.2: Database deployment and internal service

Since the internal service has been configured, other containers can connect to the database using the IP of the service. This IP is not static, thus a shortcut to it can be defined on a ConfigMap. The ConfigMap also contains the name of the database that will be used by the frontend container.

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: mysql-configmap
data:
  database-name: supermarketDB
  database-url: mysql-service
```

Listing B.3: ConfigMap with the database IP and name

B.2 Frontend configuration

The frontend part of the application is a Flask container that collects the HTTP requests, translates them in SQL queries, sends them to the backend container and presents the returned data. The frontend container is generated through the Dockerfile shown in listing B.4. The necessary code can be found on the author's GitHub account.

```
# Use an official Python runtime as an image
FROM python:3.8

# The EXPOSE instruction indicates the ports on which a container
# will listen for connections
# Since Flask apps listen to port 5000  by default, we expose it
EXPOSE 5000

# Sets the working directory for following COPY and CMD instructions
# Notice we  h a v e n t  created a directory by this name - this instruction
# creates a directory with this name if it  d o e s n t  exist
WORKDIR /app

# Install any needed packages specified in requirements.txt
COPY requirements.txt /app
RUN pip install -r requirements.txt

# Run app.py when the container launches
COPY src/ /app
CMD python backend.py
```

Listing B.4: Dockerfile for the frontend container

```
Flask
mysql-connector-python
prometheus_flask_exporter
```

Listing B.5: requirements.txt

The resulting container image is stored on DockerHub. The frontend Kubernetes deployment can now be created using the YAML file in listing... The deployment and the collected Prometheus metrics are desired to be accessible from outside the cluster and therefore two external services should be created.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: webapp-deployment
  labels:
    app: webapp
spec:
  replicas: 1
  selector:
    matchLabels:
      app: webapp
  template:
    metadata:
      labels:
        app: webapp
    spec:
      containers:
      - name: lc-benchmark
        image: gkanel/lc-benchmark:v1.6
        ports:
        - name: flask
          containerPort: 5000
        - name: metrics
          containerPort: 5099
        env:
        - name: MYSQL_USERNAME
          valueFrom:
            secretKeyRef:
              name: mysql-secret
              key: mysql-username
        - name: MYSQL_PASSWORD
          valueFrom:
            secretKeyRef:
              name: mysql-secret
              key: mysql-password
        - name: MYSQL_URL
          valueFrom:
            configMapKeyRef:
              name: mysql-configmap
              key: database-url
        - name: MYSQL_DATABASE
          valueFrom:
            configMapKeyRef:
              name: mysql-configmap
              key: database-name
```

```yaml
---
apiVersion: v1
kind: Service
metadata:
  name: webapp-service
  labels:
    app: webapp
spec:
  selector:
    app: webapp
  ports:
  - name: flask
    protocol: TCP
    port: 5000
    targetPort: 5000
  externalIPs:
  - 192.168.122.91
---
apiVersion: v1
kind: Service
metadata:
  name: webapp-metrics-service
  labels:
    app: webapp
spec:
  selector:
    app: webapp
  ports:
  - name: metrics
    protocol: TCP
    port: 5099
    targetPort: metrics
  externalIPs:
  - 192.168.122.91
```

Listing B.6: Configuration for the frontend deployment and external services

Bibliography

[1] Sameer Ajmani. *Go Concurrency Patterns: Context.* 2014. URL: `blog.golang.org/context`.

[2] Michael Armbrust et al. "Above the Clouds: A Berkeley View of Cloud Computing". In: *University of California at Berkeley UCB/EECS-2009-28, February* 28 (Jan. 2009).

[3] Richard Brown et al. "Report to Congress on Server and Data Center Energy Efficiency: Public Law 109-431". In: (Jan. 2007). DOI: `10.2172/929723`.

[4] Eric Chung et al. "Serving DNNs in Real Time at Datacenter Scale with Project Brainwave". In: *IEEE Micro* 38 (Mar. 2018), pp. 8–20. DOI: `10.1109/MM.2018.022071131`.

[5] *Client-go GitHub repository.* URL: `github.com/kubernetes/client-go`.

[6] *Cloud Native Computing Foundation.* URL: `cncf.io`.

[7] McKinsey Company. "Revolutionizing data center efficiency". In: 2008.

[8] *Considerations for large clusters.* URL: `kubernetes.io/docs/`.

[9] *Containers at Google.* URL: `https://cloud.google.com/containers`.

[10] *Containers vs. virtual machines.* 2019. URL: `https://docs.microsoft.com/en-us/virtualization/windowscontainers/about/containers-vs-vm`.

[11] Christina Delimitrou and Christos Kozyrakis. "Quasar: Resource-Efficient and QoS-Aware Cluster Management". In: vol. 49. Feb. 2014, pp. 127–144. DOI: `10.1145/2541940.2541941`.

[12] *Docker: Empowering App Development for Developers.* URL: `https://www.docker.com`.

[13] *Flask documentation.* URL: `flask.palletsprojects.com/en/1.1.x/`.

[14] Yu Gan et al. "Leveraging Deep Learning to Improve Performance Predictability in Cloud Microservices with Seer". In: *ACM SIGOPS Operating Systems Review* 53 (July 2019), pp. 34–39. DOI: `10.1145/3352020.3352026`.

[15] *Golang Microservices Example.* URL: `https://github.com/harlow/go-micro-services`.

[16] Henry Hoffmann et al. "Application heartbeats: A generic interface for specifying program performance and goals in autonomous computing environments". In: Jan. 2010, pp. 79–88. DOI: `10.1145/1809049.1809065`.

[17] *How Kubernetes Deployments work.* URL: `https://thenewstack.io/kubernetes-deployments-work/`.

[18] Eric Jonas et al. "Cloud Programming Simplified: A Berkeley View on Serverless Computing". In: (Feb. 2019).

[19] Norman Jouppi et al. "In-Datacenter Performance Analysis of a Tensor Processing Unit". In: June 2017, pp. 1–12. DOI: `10.1145/3079856.3080246`.

[20] Ram Kannan et al. "Proctor: Detecting and Investigating Interference in Shared Datacenters". In: Apr. 2018, pp. 76–86. DOI: 10.1109/ISPASS.2018.00016.

[21] *Kubernetes Components*. URL: https://www.leverege.com/iot-ebook/kubernetes-components.

[22] David Lo et al. "Improving Resource Efficiency at Scale with Heracles". In: *ACM Transactions on Computer Systems* 34 (May 2016), pp. 1–33. DOI: 10.1145/2882783.

[23] *MySQL*. URL: mysql.com.

[24] *PARSEC Wiki*. 2010. URL: wiki.cs.princeton.edu/index.php/PARSEC.

[25] *Persistent Volumes on Kubernetes*. URL: https://docs.ovh.com/gb/en/kubernetes/ovh-kubernetes-persistent-volumes/.

[26] *Prometheus: Monitoring system time series database*. URL: https://www.prometheus.io.

[27] *The PARSEC Benchmark Suite*. 2007. URL: parsec.cs.princeton.edu.

[28] Muhammad Tirmazi et al. "Borg: the next generation". In: Apr. 2020, pp. 1–14. DOI: 10.1145/3342195.3387517.

[29] A. Vasan et al. "Worth their watts? - an empirical study of datacenter servers". In: *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*. 2010, pp. 1–10. DOI: 10.1109/HPCA.2010.5463056.

[30] *What is Platform-as-a-Service (PaaS)*. URL: https://www.cloudflare.com/learning/serverless/glossary/platform-as-a-service-paas.

[31] Ran Xu et al. "Pythia: Improving Datacenter Utilization via Precise Contention Prediction for Multiple Co-located Workloads". In: Nov. 2018. DOI: 10.1145/3274808.3274820.

[32] Hailong Yang et al. "Bubble-flux: Precise Online QoS Management for Increased Utilization in Warehouse Scale Computers". In: vol. 41. July 2013, pp. 607–618. ISBN: 9781450320795. DOI: 10.1145/2485922.2485974.