Ατομική Διπλωματική Εργασία

ΑΝΑΛΥΣΗ ΚΑΙ ΧΑΡΑΚΤΗΡΙΣΜΟΣ ΠΡΟΓΡΑΜΜΑΤΩΝ ΤΟΥ SERVERLESSBENCH ΣΤΗΝ ΠΛΑΤΦΟΡΜΑ OPENWHISK

Βασίλης Χατζηπαντελής

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΥΠΡΟΥ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

Μάιος 2022

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΥΠΡΟΥ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

Ανάλυση και χαρακτηρισμός προγραμμάτων του ServelessBench στην πλατφόρμα OpenWhisk

Βασίλης Χατζηπαντελής

Επιβλέπων Καθηγητής
Βώλος Χάρης

Η Ατομική Διπλωματική Εργασία υποβλήθηκε προς μερική εκπλήρωση των απαιτήσεων απόκτησης του πτυχίου Πληροφορικής του Τμήματος Πληροφορικής του Πανεπιστημίου Κύπρου

Μάιος 2022

Ευχαριστίες

Ήθελα να ευχαριστήσω θερμά των επιβλέποντα καθηγητή μου, Χάρη Βώλο, που μου έδωσε την ευκαιρία να εκπονήσω μαζί του την διπλωματική μου εργασία. Η συμβουλές και η στήριξη του ήταν πολύ σημαντική για την ολοκλήρωση της. Επίσης ήθελα να ευχαριστήσω το τμήμα πληροφορικής του Πανεπιστημίου Κύπρου για τις γνώσεις που μου έδωσε οι οποίες με βοήθησαν για την αποπεράτωση της διπλωματικής. Τέλος θα να εκφράσω τις ευχαριστίες μου στην οικογένεια και τους φίλους μου που ήταν μαζί μου και μου συμπαραστέκονταν σε κάθε μου βήμα.

Περίληψη

Ο υπολογισμός Serverless είναι μία ανερχόμενο μοντέλο που έχει προσελκύσει την προσοχή τόσο ακαδημαϊκών όσο και της βιομηχανίας. Είναι ένα μοντέλο εκτέλεσης υπολογιστικού νέφους στο οποίο ο πάροχος cloud κατανέμει πόρους όποτε αυτοί χρειάζονται από τους χρήστες. Το "Serverless" ως όνομα μπορεί να παραπλανήσει αφού μπορεί να σκεφτεί κάποιος ότι δεν χρησιμοποιούνται διακομιστές (servers), αλλά οι διακομιστές εξακολουθούν να χρησιμοποιούνται από τους παρόχους υπηρεσιών cloud για εκτέλεση κώδικα των προγραμματιστών.

Στόχος της παρούσας διπλωματικής είναι να εξετάσω μια πλατφόρμα serverless έτσι ώστε να μελετηθούν οι προκλήσεις και πιθανά προβλήματα που υπάρχουν σε αυτό το μοντέλο και προκλήσεις και προβλήματα σε συγκεκριμένες πλατφόρμες. Μέσα από αυτή την μελέτη μπορούν να εξαχθούν συμπεράσματα έτσι ώστε να βοηθήσουν τους ερευνητές να προβούν σε βελτίωση των πλατφόρμων αυτών.

Στα πλαίσια αυτής της διπλωματικής έχω πάρει μετρικά όπως χρόνος εκτέλεσης, ενεργεία και performance counters για κάποια προγράμματα του ServerlessBench [1] οποίο διαθέτει ένα σύνολο από προγράμματα γραμμένα για να τρέχουν σε διάφορες Serverless πλατφόρμες όπως AWS Lambda [2], OpenWhisk [3], Fn [4] και Ant Financial. Σε αυτήν την εργασία αφοσιώθηκα στο να αξιολογήσω την πλατφόρμα OpenWhisk χρησιμοποιώντας μετρικά για κάποια από τα προγράμματα του ServerlessBench.

Μέσα από την μελέτη των μετρικών αυτών παρατηρήθηκε ότι υπάρχει μεγάλη αστοχία στο τελευταίο επίπεδο της κρυφής μνήμης στην πλατφόρμα OpenWhisk. Κάτι ακόμα σημαντικό είναι για να τρέξει κανείς μεγάλες εφαρμογές στο OpenWhisk ή εφαρμογές που χρησιμοποιούν μεγάλο αριθμό νημάτων ότι πρέπει να αλλαχθούν κάποια προεπιλεγμένα όρια σε αυτήν για να επιτραπεί να εκτελεστούν.

Περιεχόμενα

Κεφάλαιο 1

Εισαγωγή

1.1 Εισαγωγή

Το cloud computing και συγκεκριμένα το Serverless computing έχει γίνει ευρέως γνωστό και έχει εδραιωθεί στην αγορά. Ένας κύριος λόγο που έχει συμβεί αυτό είναι το μοντέλο pay-as-you-go, όπου ο χρήστης πληρώνει μόνο για τους πόρους τους οποίους χρησιμοποιεί.

Το serverless είναι ένα ανερχόμενο και πολλά υποσχόμενο παράδειγμα με μεγάλη χρήση στην βιομηχανίας των κοντέινερς και των microservises. Η χρήση serverless παρέχει ένα απλό προγραμματιστικό μοντέλο για την δημιουργία cloud εφαρμογών το οποίο χρησιμοποίει το μοντέλο pay-as-you-go χωρίς την επιπλέον δουλειά του ξεκινήματος και σταματήματος του διακομιστή. Οι προγραμματιστές που χρησιμοποιούν αυτό το παράδειγμα μπορούν να εξοικονομήσουν κόστος και επεκτασιμότητα χωρίς να χρειάζεται να έχουν υψηλό επίπεδο γνώσης στο cloud computing, επειδή δεν χρειάζεται να διαχειρίζονται διακομιστές.

Λόγω της απλότητας και των οικονομικών πλεονεκτημάτων του, το serverless έχει γίνει αρκετά δημοφιλές. Οι πιο μεγάλοι πάροχοι cloud η Google, η Microsoft, η

Amazon και η IBM έχουν ήδη κυκλοφορήσει δυνατότητες serverless και με αρκετές πρόσθετες προσπάθειες ανοιχτού κώδικα.

Στα πλαίσια αυτής της διπλωματικής χρησιμοποιήθηκε η πλατφόρμα OpenWhisk. Το Apache OpenWhisk είναι μια πλατφόρμα ανοιχτού κώδικα, κατανεμημένη serverless πλατφόρμα που εκτελεί συναρτήσεις ως απόκριση σε συμβάντα. Το OpenWhisk διαχειρίζεται την υποδομή, τους διακομιστές και την κλιμάκωση χρησιμοποιώντας κοντέινερ Docker.

1.2 Κίνητρο

Όλο και περισσότερος κόσμος στέφεται νέα τεχνολογίες και μοντέλα που του δίνουν περισσότερη ελευθερία και ευκαιρίες. Ένα τέτοιο μοντέλο είναι αυτού του cloud computing και συγκεκριμένα του serverless computing. Είχε ενδιαφέρον λοιπόν να εξεταστεί ένα τέτοιο μοντέλο μέσω μιας πλατφόρμας που χρησιμοποιεί αυτό το μοντέλο. Η πλατφόρμα που χρησιμοποιήθηκε είναι OpenWhisk. Στο πλαίσιο αυτής της εργασίας εξετάστηκαν προβλήματα και προκλήσεις που δυνατόν να προκύψουν κατά τη χρήση του OpenWhisk καθώς και αδυναμίες της πλατφόρμας αυτής. Συγκεκριμένα στόχος είναι να χαρακτηριστεί η πλατφόρμα OpenWhisk σε σύγχρονο επεξεργαστή, όσο αφορά χρήση ιεραρχίας μνήμης, στην κατανάλωση ενεργείας και στο πως συμπεριφέρονται οι εντολές διακλάδωσης.

1.3 Μεθοδολογία

Αρχικά έγινε μελέτη της βιβλιογραφίας σχετικά με το serverless computing. Έτσι κατανόησα την βασική ιδέα πίσω από αυτό το μοντέλο, την αρχιτεκτονική του, το προγραμματιστικό μοντέλο του, τις ευκαιρίες που προσφέρει, τις προκλήσεις, διάφορες εφαρμογές του. Επίσης στα άρθρα αυτά αναφέρονταν σε κάποιες πλατφόρμες που προσφέρουν serverless δυνατότητες όπως το Amazon Lambda [2], το Google Cloud Functions [5], το Microsoft Azure Functions [6], και το IBM OpenWhisk [3]. Επόμενο βήμα ήταν να βρούμε μια πλατφόρμα serverless την οποία θα εξετάσουμε. Μετά από παρότρυνση του επιβλέποντα καθηγητή μου επιλέξαμε το vHive [7]. Λόγο τεχνικών προβλημάτων που προέκυψαν κατά την εγκατάσταση του, επιλέξαμε το OpenWhisk το

οποίο αποτελεί μια πιο ώριμη πλατφόρμα και διαθέτει περισσότερη βιβλιογραφία. Την πλατφόρμα εγκατέστησα σε υπολογιστή του CloudLab [8] που είναι ένα έργο που υλοποιήθηκε σε συνεργασία πανεπιστημίων που σχεδιάστηκε να παρέχει διάφορες μηχανές στο νέφος σε διάφορους ερευνητές. Στη συνέχεια προσπάθησα να μελετήσω την πλατφόρμα τρέχοντας κάποια προγράμματα από το ServerlessBench το οποίο έχει υλοποιημένα κάποιο προγράμματα που τρέχουν σε πλατφόρμες serverless όπως AWS Lambda, OpenWhisk, Fn και Ant Financial. Στη διπλωματική μου χρησιμοποίησα της εφαρμογές που ήταν γραμμένες για το OpenWhisk τρέχοντας τις σε αυτήν, παίρνοντας για αυτές μετρικά όπως ο χρόνος εκτέλεσης, η ενέργεια που καταναλώνουν και performance counters.

1.4 Συνεισφορά

Στα πλαίσια αυτής της διπλωματικής εγκαταστάθηκε η πλατφόρμα OpenWhisk σε μηχανή του CloudLab και εξάχθηκαν μετρήσεις για προγράμματα του ServerlessBench. Αυτές αφορούσαν τον χρόνο εκτέλεσης, την κατανάλωση της ενέργειας καθώς και κάποιοι performance counters. Στα προγράμματα που έτρεξα στην πλατφόρμα αυτή παρατήρησα ότι στο τελευταίο επίπεδο κρυφής μνήμης υπάρχει μεγάλο ποσοστό αστοχίας που οφείλεται σε μεγάλο βαθμό από το docker. Αυτό θα μπορούσαν να το λάβουν υπόψιν τους οι κατασκευαστές τις πλατφόρμας ή ακόμα οι κατασκευαστές του docker για βελτίωση του. Επίσης παρουσιάζεται ένας οδηγός εγκατάστασης του OpenWhisk του σε κάποιο υπολογιστή, πράγματα που πρέπει να προσέξει κανείς κατά την εγκατάσταση και το πως μπορεί κάποιος να τρέξει εφαρμογές σε αυτήν την πλατφόρμα.

1.5 Δομή

Αυτή η διπλωματική εργασία αποτελείται από 5 κεφάλαια. Το Κεφάλαιο 1 περιέχει μια μικρή εισαγωγή για το serverless computing, το κίνητρο μου που επέλεξα αυτή την διπλωματική και την μεθοδολογία που χρησιμοποιήθηκε. Στο Κεφάλαιο 2 είναι το θεωρητικό υπόβαθρο του serverless, την αρχιτεκτονική, το προγραμματιστικό μοντέλο και τις ευκαιρίες που δίδει αυτό το μοντέλο. Στο Κεφάλαιο 3 γίνεται αναφορά στην αρχιτεκτονική της serverless πλατφόρμας που χρησιμοποιήθηκε, το OpenWhisk και το

πως μπορεί κάποιος να την εγκαταστήσει. Στο Κεφάλαιο 4 παρουσιάζονται το προγράμματα του ServerlessBench που εκτελέστηκαν στο OpenWhisk, παίρνοντας μετρικές του χρόνου εκτέλεσης, της ενέργειας που καταναλώνεται από το επεξεργαστή και την κύρια μνήμη και performance counters με τη χρήση του perf. Στο Κεφάλαιο 5 αναφέρονται τα συμπεράσματα της διπλωματικής καθώς και μελλοντική δουλειά.

Κεφάλαιο 2

Θεωρητικό υπόβαθρο

2.1 Serverless computing

Το Serverless computing είναι ένα είναι ένα ταχέως αναπτυσσόμενο μοντέλο ανάπτυξης εφαρμογών, εγγενές στο cloud. Η πλατφόρμα και η υποδομή στην οποία εκτελούνται οι υπηρεσίες δεν είναι ορατά στον προγραμματιστή. Αυτούς τους ενδιαφέρει μόνο το πως λειτουργεί το πρόγραμμα τους και τα υπόλοιπα είναι δουλεία του παροχέα υπηρεσιών. Υπάρχουν υλοποιήσεις αυτού του μοντέλου όπως το Lambda της Amazon, το Google Cloud Function, το Microsoft Azure Functions και το IBM OpenWhisk.

Ο σκοπός του serverless είναι να διευκολύνουν τους χρήστες των υπηρεσιών cloud έτσι ώστε να μην χρειάζεται να ανησυχούν για την υποδομή της πλατφόρμας και αυτόματη κλιμάκωση της υπηρεσίας. Επίσης παρέχει το μοντέλο pay-as-you-go (πληρώνεις για πόρους που είναι πραγματικά χρησιμοποιείς). Το serverless computing περιστρέφεται γύρω από τη χρήση συναρτήσεων. Ο πελάτης καταχωρεί τις λειτουργίες στον πάροχο υπηρεσιών. Αυτές οι λειτουργίες μπορούν να κληθούν από ένα συμβάν ή κατόπιν αιτήματος των χρηστών. Τα αποτελέσματα της εκτέλεσης αποστέλλονται πίσω στον πελάτη. Η επίκληση συνάρτησης εκχωρείται σε έναν κόμβο εντός του παρόχου

υπηρεσιών. Αυτοί οι κόμβοι είναι συνήθως κοντέινερς νέφους, όπως το Docker ή απομονωμένα περιβάλλοντα χρόνου εκτέλεσης [9]

2.5 Ορισμοί

"**FaaS.** Function as a service είναι ένα παράδειγμα στο οποίο οι πελάτες μπορούν να αναπτύξουν, να τρέξουν και διαχειριστούν τις λειτουργίες της εφαρμογής χωρίς να χρειάζεται να δημιουργήσουν και να διατηρήσουν την υποδομή." [9]

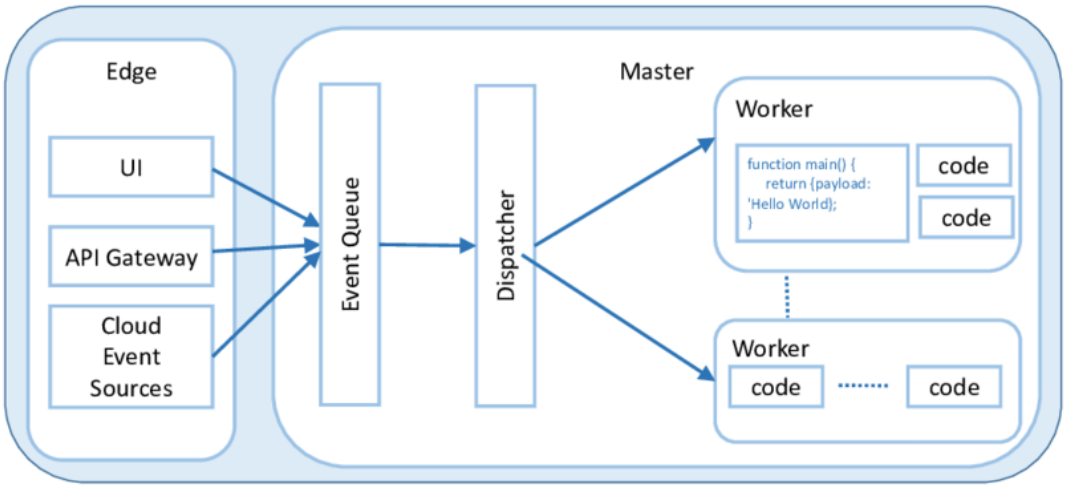
"**BaaS.** Το Backend as a Service (BaaS) είναι μια διαδικτυακή υπηρεσία που χειρίζεται μια συγκεκριμένη εργασία στο cloud, όπως η αυθεντικοποίηση και η ειδοποίηση.
Τόσο το BaaS όσο και το FaaS δεν απαιτούν διαχείριση πόρων από τους πελάτες. Ενώ το FaaS προσφέρει μόνο την εκτέλεση των λειτουργιών των χρηστών, το BaaS προσφέρει μια πλήρη ηλεκτρονική υπηρεσία." [9]

Serverless service. Ένα serverless service μπορεί να θεωρηθεί ως γενίκευση των FaaS και BaaS που τηρούν τα ακόλουθα χαρακτηριστικά. Πρώτο το περιβάλλον εκτέλεσης θα πρέπει να είναι κρυφό από τον πελάτη. Δεύτερο η υπηρεσία αυτόματης κλιμάκωσης πρέπει να παρέχεται από τον πάροχο. Τρίτον ο χρήστης πρέπει να χρεώνεται μόνο τον αριθμό των πόρων που χρησιμοποιεί (μοντέλο pay-as-you-go). Τέταρτο ο πάροχος καταβάλλει κάθε δυνατή προσπάθεια για να ολοκληρώσει την εργασία του πελάτη αμέσως μόλις λάβει το αίτηση και η διάρκεια εκτέλεσης είναι περιορισμένη. Πέμπτο τα βασικά στοιχεία στα serverless service είναι οι συναρτήσεις. Οι λειτουργίες δεν είναι κρυφές από το πάροχο. Ο πάροχος γνωρίζει τις εξαρτήσεις του από εξωτερικές βιβλιοθήκες, τα περιβάλλοντα χρόνου εκτέλεσης, και την κατάσταση κατά τη διάρκεια και μετά την εκτέλεση.

Μια serverless εφαρμογή συνήθως αποτελείται από δύο μέρη. Το ένα είναι ο client (πελάτης) ο οποίος υλοποιεί το μεγαλύτερο μέρος της λογικής της εφαρμογής. Αλληλεπιδρά με τον τελικό χρήστη και παρέχει μετάφραση μεταξύ λειτουργιών από τη μία πλευρά και χρήσιμων δεδομένων από την άλλη. Το άλλο είναι οι εγγεγραμμένες συναρτήσεις σε έναν πάροχο. Οι συναρτήσεις αναρτιούνται στον πάροχο και αυτός καλεί

ένα αντίγραφο της συνάρτησης σύμφωνα με το αίτημα του χρήστη ή με βάση ένα προκαθορισμένο συμβάν.

2.6 Αρχιτεκτονική του serverless



Σχήμα 2.1 Η αρχιτεκτονική του serverless

Στο Σχήμα 2.1 φαίνεται η αρχιτεκτινική του μοντέλου serverless. Η κύρια λειτουργηκότητα μια serverless δομής είναι ένα σύστημα επεξεργασίας events και διαχειρίσης συναρτήσεων (actions). Όταν ένα αίτημα παραλαμβάνεται μέσω HTTP από μια πηγή δεδομένων events (triggers), το σύστημα διευκρινίζει ποιο action πρέπει να διαχειριστεί το event, δημιουργεί ένα instance κοντέινερ, στέλνει το event στο instance μιας συνάρτησης, περιμένει για ένα αποτέλεσμα το οποίο θα δώσει στο χρήστη και θα σταματήσει τη συνάρτηση όταν αυτή δεν χρειάζεται.

Ενώ η αρχιτεκτονική είναι σχετικά απλή, η πρόκληση είναι να εφαρμοστεί λαμβάνοντας υπόψη μετρικές όπως το κόστος, η επεκτασιμότητα, η καθυστέρηση και η ανοχή σφαλμάτων. Για να απομονωθεί η εκτέλεση συναρτήσεων από διαφορετικούς χρήστες, συχνά χρησιμοποιούνται κοντέινερ όπως το Docker.

Τα κοντέινερ είναι πακέτα λογισμικού που περιέχουν ότι χρειάζεται για να εκτελεστούν σε οποιοδήποτε περιβάλλον. Με αυτόν τον τρόπο, τα κοντέινερ εικονικοποιούν το

λειτουργικό σύστημα και εκτελούνται οπουδήποτε, όπως σε ένα ιδιωτικό κέντρο δεδομένων, ένα δημόσιο cloud ή και σε κάποιο προσωπικό υπολογιστή.

Όταν παραλειφθεί ένα event γίνεται έλεγχος αυθεντικοποίησης και εξουσιοδότησης σε αυτό. Επίσης ελέγχονται το πόσοι πόροι χρειάζονται από κάποιο event. Μόλις γίνουν οι έλεγχοι, το event μπαίνει σε ουρά για εκτέλεση από την πλατφόρμα. Ένας εργάτης προσκομίζει το αίτημα, δημιουργεί τον κατάλληλο κοντέινερ, αντιγράφει τον κώδικα της συνάρτησης στον κοντέινερ και εκτελεί το event. Η πλατφόρμα διαχειρίζεται επίσης τη διακοπή και την αποδέσμευση πόρων για αδρανείς συναρτήσεις.

Η διαχείριση των νέων κοντέινερς για κάθε κάλεσμα συνάρτησης μπορεί να είναι ακριβή και να προσθέτει χρονική επιβάρυνση (cold start). Σε αντίθεση η ζεστοί (warm) κοντέινερς, είναι κοντέινερς που έχουν αρχικοποιηθεί και εκτέλεσαν μια συνάρτηση. Προβλήματα cold start μπορούν να αντιμετωπιστούν χρησιμοποιώντας μια δεξαμενή από κοντέινερς οι οποίοι έχουν αρχικοποιηθεί αλλά δεν χρησιμοποιήθηκαν από κάποιο χρήστη, ή να ξαναχρησιμοποιηθεί ένας κοντέινερ που έχει καλεστεί από τον ίδιο χρήστη. Κάτι ακόμα που μπορεί να επηρεάσει την καθυστέρηση είναι το ότι εξαρτάται η συνάρτηση από βιβλιοθήκες οι οποίες χρειάζεται να εγκατασταθούν. Για να μειωθεί αυτό μπορούν να γίνονται cached τα πιο σημαντικά πακέτα στους κόμβους εργάτες.

Συνήθως στο serverless οι χρήστες μπορούν μόνο να καθορίσουν το μέγεθος της κύριας μνήμης που θα χρησιμοποιήσει η συνάρτηση. Η πλατφόρμα θα δεσμεύσει τους άλλους πόρους συναρτήσει της κύριας μνήμης. Όσο μεγαλύτερη η μνήμη τόσο μεγαλύτερη δέσμευση επεξεργαστών. Η χρήση των πόρων μετριέται και χρεώνεται σε μικρά κομάτια και οι χρήστες πληρώνουν μόνο για τον χρόνο και τους πόρους που χρησιμοποιούνται όταν οι συναρτήσεις τους εκτελούνται. [10]

2.4 Προγραμματιστικό μοντέλο του serverless

Συνήθως το προγραμματιστικό μοντέλο του serverless αποτελείται από δύο μέρη, το action και το trigger. Το action είναι μια συνάρτηση χωρίς κατάσταση που εκτελεί κάποιο κώδικα. Τα actions μπορούν να κληθούν ασύγχρονα δηλαδή ο καλών δεν περιμένει απάντηση, ή σύγχρονα όπου ο καλών περιμένει απάντηση. Το trigger είναι ένα

σύνολο συμβάντων από διαφορετικές πηγές. Τα actions μπορούν να πραγματοποιηθούν απευθείας μέσω REST API, ή να εκτελεστούν μέσω trigger. Ένα συμβάν μπορεί επίσης να ενεργοποιήσει (trigger) πολλές λειτουργίες ή το αποτέλεσμα ενός action θα μπορούσε επίσης να ενεργοποιήσει μια άλλη συνάρτησης.

Οι serverless πλατφόρμες εκτελούν μια συνάρτηση main που παίρνει ένα λεξικό ως είσοδο και παράγει ένα λεξικό ως έξοδο. Δεν έχουν μεγάλη εκφραστικότητα καθώς είναι κατασκευασμένα για να κλιμακώνουν. Για να μεγιστοποιηθεί η κλιμάκωση, οι serverless συναρτήσεις δεν διατηρούν την κατάσταση μεταξύ των εκτελέσεων. Αντίθετα, ο προγραμματιστής μπορεί να γράψει κώδικα στη συνάρτηση για να ανακτήσει και ενημερώσει οποιαδήποτε κατάσταση απαιτείται. Η συνάρτηση μπορεί επίσης να έχει πρόσβαση σε ένα αντικείμενο περιβάλλοντος που αντιπροσωπεύει το περιβάλλον στο οποίο εκτελείται η συνάρτηση.

Οι πάροχοι cloud προσφέρουν ένα οικοσύστημα υπηρεσιών που υποστηρίζει τις διαφορετικές λειτουργίες που μπορεί να απαιτήσει ένας προγραμματιστής και είναι απαραίτητες σε αρκετές εφαρμογές. Για παράδειγμα, μια συνάρτηση μπορεί να χρειαστεί να ανακτήσει την κατάσταση από μια βάση δεδομένων ή ένας άλλος μπορεί να χρησιμοποιεί μηχανική μάθηση. Έτσι πρέπει το σύστημα αποθήκευσης να παρέχει αξιοπιστία και εγγυήσεις QoS για την εξασφάλιση της ομαλής λειτουργίας.

2.7 Ευκαιρίες

Χωρίς πολυπλοκότητα εγκατάστασης και συντήρησης

Πιο σημαντική ευκαιρία για του serverless είναι ότι ο προγραμματιστής δεν χρειάζεται να ασχοληθεί με την υποδομή, αλλά ακόμα πρέπει να διαχειρίζεται τους εικονικούς πόρους όπως βιβλιοθήκες και πακέτα. Στο serverless οι χρήστης χρειάζεται μόνο να αναρτήσει την συνάρτηση τους και μετά να πάρει τα συνθηματικά για να την καλέσει.

Προσιτή κλιμάκωση

Ακόμα μια ευκαιρία των serverless είναι ότι ο προγραμματιστής δεν χρειάζεται να ασχοληθεί για την κλιμάκωση της πλατφόρμα που χρησιμοποιεί. Η κλιμάκωση είναι ένα φυσική συνέπεια της αυτόματης κλιμάκωσης αυτό των υπηρεσιών. Η προσιτότητα στις serverless υπηρεσίες οφείλεται στα μειωμένα κόστη των προμηθευτών. Υπάρχουν δύο κύριοι λόγοι για τα μειωμένα κόστη, η πολυπλεξία πόρων και η ετερογένεια υποδομής. Η πολυπλεξία πόρων έχει ως αποτέλεσμα την μεγαλύτερη χρησιμοποίηση των διαθέσιμων πόρων. Η ετερογένεια υποδομής σημαίνει ότι οι πάροχοι μπορούν να χρησιμοποιήσουν και παλιότερες μηχανές για να μειώσουν τα κόστη τους. Είναι σημαντικό να σημειωθεί ότι η προσιτότητα της serverless προσέγγισης εξαρτάτε από το σενάριο.

Νέες αγορές

Με την έλευση των σύγχρονων λειτουργικών συστημάτων για φορητές συσκευές όπως το Android ή το iOS, ανοίχτηκαν διάφορες θέσεις αγοράς για εφαρμογές, που τρέχουν σε αυτά τα λειτουργικά συστήματα. Αυτό ήδη εμφανίζεται για την προσέγγιση serverless αφού έχουν προκύψει νέες αγορές για αυτές τις συναρτήσεις. Σε αυτές τις αγορές, οι προγραμματιστές μπορούν να τα πουλήσουν και να αναπτύξουν συναρτήσεις για άλλους. Το AWS Serverless Application Repository είναι ένα παράδειγμα τέτοιας δυνατότητας το οποίο παρουσιάζει μια ποσοτική ανάλυση των λειτουργιών που είναι διαθέσιμες μέσα σε αυτό. Ο ανταγωνισμός που θα επέλθει από αυτές τις αγορές θα έχει ως αποτέλεσμα την δημιουργία υψηλής ποιότητας συναρτήσεων.

Κεφάλαιο 3

Περιγραφή και εγκατάσταση OpenWhisk

3.1 Γενική περιγραφή του OpenWhisk

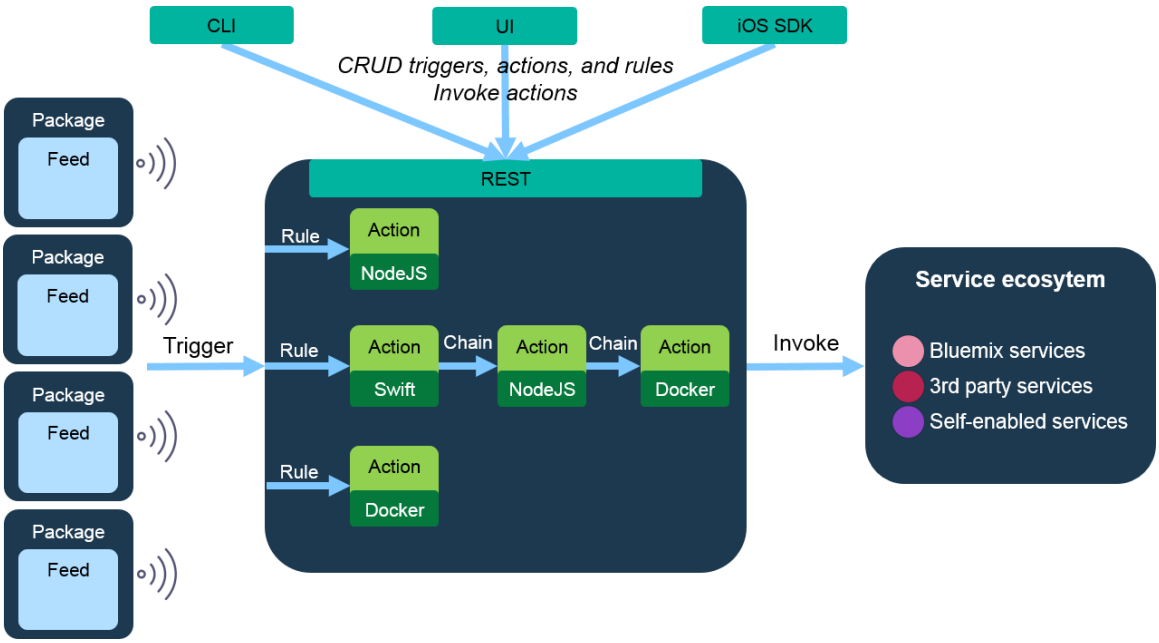
Το OpenWhisk είναι ένα έργο του Apache Software Foundation. Είναι μια εφαρμογή ανοιχτού κώδικα μιας κατανεμημένης υπηρεσίας, βασισμένης σε events που μπορεί να εκτελεστεί στον υπολογιστή κάποιου ή στο cloud.

Το OpenWhisk παρέχει μια πλατφόρμα εκτέλεσης που βασίζεται σε συμβάντα για εφαρμογές ιστού και κινητών. Οι υπηρεσίες IBM Cloud και οι εξωτερικές πηγές μπορούν να παρέχουν συμβάντα. Οι προγραμματιστές μπορούν να επικεντρωθούν στη προγραμματισμό της εφαρμογής και στη δημιουργία ενεργειών που εκτελούνται on-demand. Τα οφέλη είναι ότι οι διακομιστές δεν παρέχονται ρητά και δεν χρειάζεται ο προγραμματιστής να ανησυχεί για την κλιμάκωση, την υψηλή διαθεσιμότητα, τις ενημερώσεις, τη συντήρηση και την πληρωμή για ώρες χρήσης του διακομιστή. Ο κώδικας εκτελείται αυτόματα όταν συμβεί μια κλήση HTTP, όταν αλλάζει η κατάσταση βάσης δεδομένων ή ένας άλλος τύπος συμβάντος.

Το μοντέλο προγραμματισμού είναι κατάλληλο για μικροϋπηρεσίες, IoT, κινητές συσκευές και άλλες εφαρμογές που χρειάζονται αυτόματη κλιμάκωση και

εξισορρόπηση φορτίου. Για να ξεκινήσει κάποιος το μόνο που χρειάζεται είναι να δώσει τον κώδικα στον προμηθευτή[11]

3.2 Υψηλού επιπέδου περιγραφή της αρχιτεκτονικής του OpenWhisk



Σχήμα 3.1 Αναπαράσταση υψηλού επιπέδου αρχιτεκτονικής του Openwhisk

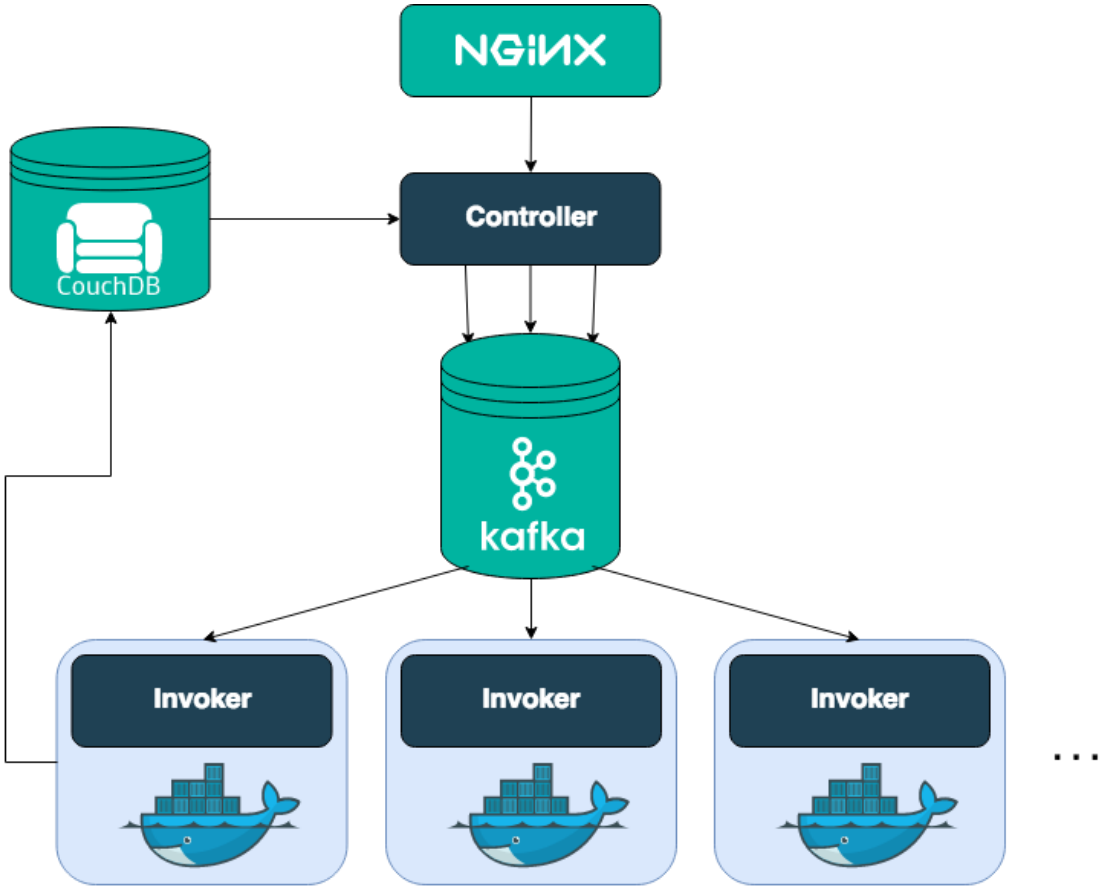
Το Σχήμα 3.1 δείχνει την υψηλού επιπέδου αρχιτεκτονική του OpenWhisk. Τα γεγονότα που πυροδοτούν ένα συμβάν είναι αυτά που θέτουν τους κανόνες που επιτρέπουν στις ενέργειες να πραγματοποιηθούν.

Οι ενέργειες μπορεί να είναι μικρά κομμάτια κώδικα ή δυαδικός κώδικας ενσωματωμένος σε ένα κοντέινερ Docker. Οι ενέργειες στο OpenWhisk εκτελούνται αμέσως κάθε φορά που ενεργοποιείται μια σκανδάλη.

Μπορεί να χρησιμοποιηθεί το OpenWhisk API, CLI ή iOS SDK για να καλέστει μια ενέργεια απευθείας. Ακόμα ένα σύνολο ενεργειών μπορεί επίσης να συνδεθεί με αλυσίδα. Κάθε ενέργεια στην αλυσίδα καλείται διαδοχικά με την έξοδο μιας ενέργειας να περνά ως είσοδος στην επόμενη της ακολουθίας.

Πρόσθετες υπηρεσίες και πάροχοι συμβάντων πακέτων μπορούν να προστεθούν σε ενσωματώσεις με την πλατφόρμα. Μια ροή είναι ένα κομμάτι κώδικα που διαμορφώνει μια εξωτερική πηγή συμβάντος για να ενεργοποιεί συμβάντα σκανδάλων. Οι ενέργειες πακέτου που παρέχονται από έναν πάροχο υπηρεσιών μπορούν να χρησιμοποιηθούν από τους προγραμματιστές τόσο ως πηγή συμβάντων όσο και ως API.

3.3 Η εσωτερική ροή της επεξεργασίας στο OpenWhisk



Σχήμα 3.2 Αναπαράσταση εσωτερική ροής της επεξεργασίας στο OpenWhisk

Nginx

Το OpenWhisk API είναι χτισμένο σε HTTP και ακολουθεί το μοντέλο RESTful. Χρησιμοποιείται το wsk CLI για να σταλθεί ένα αίτημα HTTP στο σύστημα OpenWhisk.

Το πρώτο σημείο πρόσβασης στο σύστημα είναι μέσω του nginx το οποίο είναι HTTP και reverse proxy server. Η κύρια λειτουργία του είναι ο τερματισμός SSL και η προώθηση κατάλληλων κλήσεων HTTP στο επόμενο στοιχείο.

Controller

Το αίτημα HTTP μεταβιβάζεται από το nginx στο controller. Είναι γραμμένο σε scala και υλοποιεί το REST API. Είναι μια διεπαφή για οτιδήποτε μπορεί να κάνει ο χρήστης όπως η κλήση actions και αιτήσεις CRUD (create,read,update,delete). Ο Controller αποσαφηνίζει τι προσπαθεί να κάνει ο χρήστης μέσω της μεθόδου HTTP στο αίτημα HTTP.

CouchDB

Ο Controller επαληθεύει ποιος είναι ο χρήστης (Authentication) και εάν έχει το προνόμιο να κάνετε αυτό που θέλει (Authorization). Τα credentials που περιλαμβάνονται στο αίτημα επαληθεύονται χρησιμοποιώντας τη βάση δεδομένων θεμάτων στο CouchDB instance.

Μετά φορτώνει το action από τη βάση whisks στην CouchDB. Η εγγραφή του action περιλαμβάνει το κώδικα, τις παραμέτρους και κάποιους περιορισμούς πόρων.

Load Balancer

Ο Load Balancer που είναι μέρος του Controller παρακολουθεί συνεχώς την κατάσταση της υγείας των εκτελεστών που είναι διαθέσιμοι στο σύστημα. Αυτοί οι εκτελεστές ονομάζονται Invokers. Ο Load Balancer, γνωρίζοντας ποιοι Invokers είναι διαθέσιμοι, επιλέγει έναν από αυτούς για να καλέσει το action που ζητήθηκε.

Kafka

Η Kafka είναι ένα“high-throughput, distributed, publish-subscribe messaging system”. Η αρχιτεκτονική publish-subscribe είναι ένα μοτίβο ανταλλαγής μηνυμάτων όπου οι

αποστολείς μηνυμάτων, δεν προγραμματίζουν τα μηνύματα να σταλούν απευθείας σε συγκεκριμένους δέκτες, αλλά χωρίζουν τα μηνύματα σε κατηγορίες χωρίς να γνωρίζουν ποιο είναι οι δέκτες. Το ίδιο ισχύει και για τους δέκτες οι οποίοι εκδηλώνουν ενδιαφέρον για μία ή περισσότερες κατηγορίες και λαμβάνουν μόνο μηνύματα που παρουσιάζουν ενδιαφέρον, χωρίς να γνωρίζουν ποιο είναι οι αποστολείς. Η Kafka είναι υπεύθυνη να χειριστεί δύο προβλήματα που μπορεί να προκύψουν. Υπάρχει πιθανότητα το σύστημα να καταρρεύσει, κάτι που θα σήμαινε ότι η κλήση θα χαθεί και το σύστημα μπορεί να είναι πολύ απασχολημένο για να δεχτεί άλλες κλήσεις και έτσι η κλήση πρέπει να περιμένει να τελειώσουν πρώτα άλλες κλήσεις.

Ο Controller και ο Invoker επικοινωνούν μόνο μέσω μηνυμάτων που είναι buffered και αποθηκεύονται στην Kafka. Αυτό λύνει το πρόβλημα buffering στη μνήμη, τόσο από τον Controller όσο και από τον Invoker και διασφαλίζει ότι τα μηνύματα δεν θα χαθούν σε περίπτωση σφάλματος του συστήματος.

Στη συνέχει για να ενεργοποιηθεί το action, ο Controller δημοσιεύει ένα μήνυμα στην Kafka, το οποίο περιέχει το action προς εκτέλεση και τις παραμέτρους που πρέπει να περάσουν σε αυτό. Αυτό το μήνυμα παραλαμβάνεται από Invoker που επέλεξε ο Controller παραπάνω από τη λίστα των διαθέσιμων invokers.

Μόλις η Kafka επιβεβαιώσει ότι έλαβε το μήνυμα, επιστρέφεται στον χρήστη ένα activationID. Ο χρήστης θα το χρησιμοποιήσει αργότερα, για να αποκτήσει πρόσβαση στα αποτελέσματα αυτής της συγκεκριμένης επίκλησης.

Invoker

Η δουλεία του Invoker είναι να καλεί actions. Το Docker χρησιμοποιείται για την εκτέλεση actions με απομονωμένο και ασφαλή τρόπο.

Το Docker χρησιμοποιείται για τη ρύθμιση ενός container για κάθε ενέργεια Με άλλα λόγια, για κάθε κλήση action δημιουργείται ένα container Docker, αντιγράφεται ο

κώδικας του action σε αυτό, εκτελείται χρησιμοποιώντας τις παραμέτρους που του μεταβιβάζονται, προκύπτει το αποτέλεσμα και το container καταστρέφεται.

CouchDB

Όταν το αποτέλεσμα λαμβάνεται από τον Invoker, αποθηκεύεται στη βάση δεδομένων activations που βρίσκεται στο CouchDB ως ενεργοποίηση κάτω από το ActivationId που αναφέρεται παραπάνω.

3.4 Εγκατάσταση του OpenWhisk

Σε αυτό το μέρος θα αναφέρω τα βήματα που χρειάζονται για να εγκατασταθεί το OpenWhisk σε ένας υπολογιστή. Καταρχήν για να τρέξει το apache OpenWhisk χρειάζεται να υπάρχει κατεβασμένο το docker, η java 8 και το node.js.

Εγκαθίδρυση του repository

Ενημέρωση του ευρετήριου πακέτων apt και εγκατάσταση πακέτων για να επιτραπεί στην apt να χρησιμοποιεί repository μέσω HTTPS

```
sudo apt-get update
sudo apt-get install \
   apt-transport-https \
   ca-certificates \
   curl \
   gnupg \
   lsb-release
```

Προσθήκη του επίσημου κλειδιού GPG του Docker

```
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor
-o /usr/share/keyrings/docker-archive-keyring.gpg
```

Χρήση της ακόλουθης εντολής για ρύθμιση του σταθερού repository.

```
echo \
  "deb [arch=amd64 signed-by=/usr/share/keyrings/docker-archive-keyring.gpg]
https://download.docker.com/linux/ubuntu \
  $(lsb_release -cs) stable" | sudo tee /etc/apt/sources.list.d/docker.list
> /dev/null
```

Εγκατάσταση μιας μηχανής docker

Ενημέρωση του ευρετηρίου πακέτου apt και εγκατάσταση της πιο πρόσφατης έκδοσης της μηχανής Docker και του containerd

```
sudo apt-get update
sudo apt-get install docker-ce docker-ce-cli containerd.io
```

Δημιούργια της ομάδας docker και να προσθήκη του χρήστη

```
sudo groupadd docker
sudo usermod -aG docker $USER
```

Να κατεβούν τα παρακάτω dependencies

```
sudo apt install openjdk-8-jdk -y
sudo apt-get install jq -y
sudo apt install nodejs npm -y
sudo apt install xdg-utils -y
```

Να κατευεί το openwhisk και το openwhisk-cli

```
git clone https://github.com/apache/openwhisk.git
wget https://github.com/apache/openwhisk-
cli/releases/download/1.2.0/OpenWhisk_CLI-1.2.0-linux-amd64.tgz
tar zxvf OpenWhisk_CLI-1.2.0-linux-amd64.tgz
sudo cp wsk /usr/local/bin
```

Αύξηση κάποιον ορίων του OpenWhisk μέσω μεταβλητών περιβάλλοντος

Στην προσπάθεια μου να τρέξω κάποια προγράμματα του ServerlessBench διαπίστωσα ότι για πολλά καλέσματα actions ή όταν προσπαθούσα να τα τρέξω για πολλά νήματα παρουσιάζονταν σφάλματα. Διερεύνησα το θέμα και ότι αυτό οφειλόταν σε χαμηλά προκαθορισμένα όρια του OpenWhisk και έτσι μέσω των πιο κάτω μεταβλητών περιβάλλοντος τα αύξησα. Το `LIMITS_ACTIONS_INVOKES_PERMINUTE υποδηλώνει` πόσο actions μπορούν να καλεστούν σε ένα λεπτό, το `LIMITS_ACTIONS_INVOKES_CONCURRENT` το πόσα actions μπορούν τα καλεστούν ταυτόχρονα, το `LIMITS_TRIGGERS_FIRES_PERMINUTE` το πόσα triggers μπορούν να ενεργοποιηθούν το λεπτό, το `LIMITS_ACTIONS_INVOKES_CONCURRENTINSYSTEM` το ποσά actions μπορούν να καλεστούν ταυτόχρονα στο σύστημα και το `LIMITS_ACTIONS_SEQUENCE_MAXLENGTH` το πόσα actions μπορούν να κληθούν σε ακολουθία.

```
export LIMITS_ACTIONS_INVOKES_PERMINUTE=60000
export LIMITS_ACTIONS_INVOKES_CONCURRENT=5000
export LIMITS_TRIGGERS_FIRES_PERMINUTE=60000
export LIMITS_ACTIONS_INVOKES_CONCURRENTINSYSTEM=5000
export LIMITS_ACTIONS_SEQUENCE_MAXLENGTH=200
```

Εκτέλεση του OpenWhisk

```
cd openwhisk
 ./gradlew core:standalone:bootRun
```

Μόλις τελειώσει η εκτέλεση του `./gradlew core:standalone:bootRun` θα εμφανιστεί η πιο κάτω εντολή η οποία πρέπει να αντιγραφεί στην γραμμή εντολών.

```
wsk property set \  --apihost 'http://localhost:3233' \  --auth '23bc46b1-
71f6-4ed5-8c54-
816aa4f8c502:123zO3xZCLrMN6v2BKK1dXYFpXlPkccOFqm12CdAsMgRU4VrNZ9lyGVCGuMDGIw
P'
```

3.5 Εγκατάσταση εφαρμογής στο OpenWhisk

```
function main(args) {
 if (args && args.name) {
 console.log(`hello ${args.name}`);
 return { msg: `hello ${args.name}` };
 } else {
 console.log(`hello world`);
 return { msg: `hello world` };
 }
}
```

Ας υποθέσουμε ότι έχουμε το παραπάνω πρόγραμμα γραμμένο σε javascript με όνομα hello.js.

```
wsk action create hello hello.js
```

Η παραπάνω εντολή θα δημιουργήσει το action για hello.js και θα το δώσει στο action το όνομα hello.

```
$ wsk action invoke hello -r
```

Η παραπάνω εντολή θα καλέσει το action με το όνομα hello που πριν δημιουργήσαμε. Η σημαία –r θα τυπώσει το αποτέλεσμα που είναι το παρακάτω.

```
{
 "msg": "hello world"
}
```

```
$ wsk action invoke hello -r -p name john
```

Η παραπάνω εντολή θα καλέσει το action με το όνομα hello. Η σημαία –p που προσθέσαμε θα περάσει στο πρόγραμμα στην μεταβλητή name το john. Το αποτέλεσμα αυτού φαίνεται παρακάτω.

```
{
 "msg": "hello john"
}
```

Κεφάλαιο 4

Εκτέλεση και λήψη μετρικών για προγράμματα του ServerlessBench στο OpenWhisk

4.1 Εισαγωγή

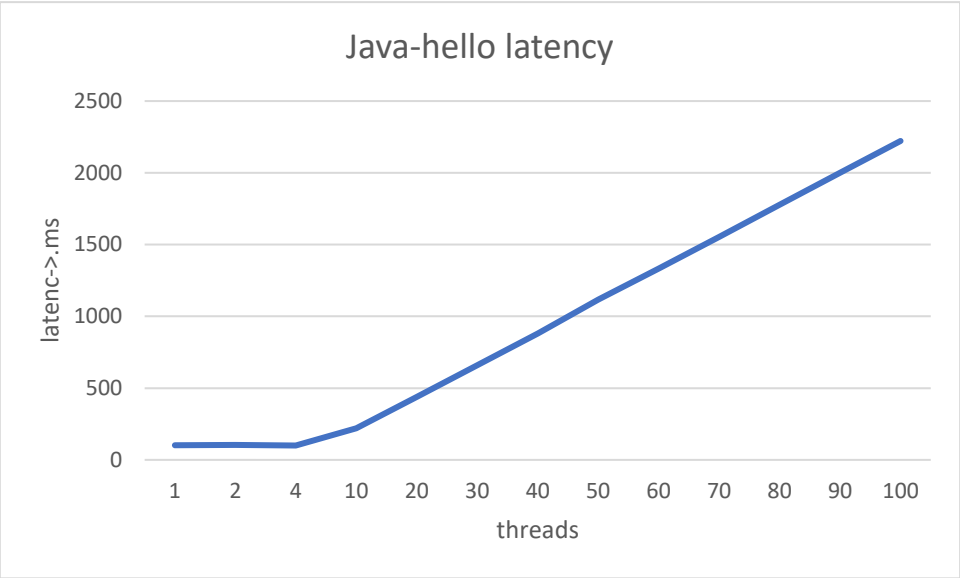
Το ServerlessBench είναι ένα ανοικτού κώδικα σύνολο από μετροπρογράμματα που δημιουργήθηκαν για χαρακτηρισμό serverless πλατφόρμων. Αυτά τα προγράμματα έχουν εφαρμοστεί σε τέσσερις πλατφόρμες serverless το AWS Lambda, το OpenWhisk το Fn και το Ant financial. Στη διπλωματική αφοσιώθηκα στην χρήση τους για μελέτη και χαρακτηρισμό την πλατφόρμας OpenWhisk και προσπάθησα να έξαξω κάποια συμπεράσματα.

Συγκεκριμένα μελέτησα την συμπεριφορά τριών προγραμμάτων στο OpenWhisk. Όλα τα προγράμματα και το OpenWhisk τα έτρεχα σε μηχανές του CloudLab που είναι ένα έργο των πανεπιστημίων University of Utah, Clemson University, University of Wisconsin Madison, the University of Texas at Austin, the University of Massachusetts Amherst και US Ignite που σχεδιάστηκε να παρέχει διάφορες μηχανές στο νέφος σε διάφορους ερευνητές. Στη διπλωματική μου έχω χρησιμοποιήσει μηχανές από την συστοιχία των μηχανών του Wisconsin και συγκεκριμένα από τους υπολογιστές c220g5, οι οποίοι έχουν δύο δεκαπύρηνους επεξεργαστές Intel Xeon Silver 4114 αρχιτεκτονικής x86_64 με συχνότητα 2.20GHz και έχουν εγκατεστημένο το λειτουργικό σύστημα UBUNTU18-64-STD. Χρησιμοποίησα μόνο ένα τέτοιο υπολογιστή στον οποίο ήταν ενεργοποιημένο το hyperthreading, δηλαδή για κάθε φυσικό επεξεργαστή υπήρχαν 2 λογικοί επεξεργαστές και άρα συνολικά υπήρχαν 40 λογικοί επεξεργαστές. Μελέτησα τρία είδη μετρικών τον χρόνο εκτέλεσης, την ισχύ και κάποιο performance counters. Για την μέτρηση της ισχύς και των performance counters χρησιμοποίησα το perf το οποίο έτρεχα στο παρασκήνιο και το ξεκινούσα πριν την έναρξη του προγράμματος και το σταματούσα μετά το τέλος του. Οι μετρήσεις μετρούν στατιστικά ολόκληρου του συστήματος πράγμα που μπορεί να επηρεάζει σε κάποιο βαθμό τις μετρήσεις.

4.2 Εκτέλεση προγράμματος Java-hello

Το πρώτο πρόγραμμα που χρησιμοποίησα για να δω την συμπεριφορά του στο OpenWhisk είναι το Java-hello το οποίο τυπώνει “greeting: Hello stranger!” αν το πρόγραμμα δε πάρει όρισμα και “greeting: Hello input!” όπου input το όρισμα που θα πάρει το πρόγραμμα.

4.2.1 Χρόνος εκτέλεσης Java-hello

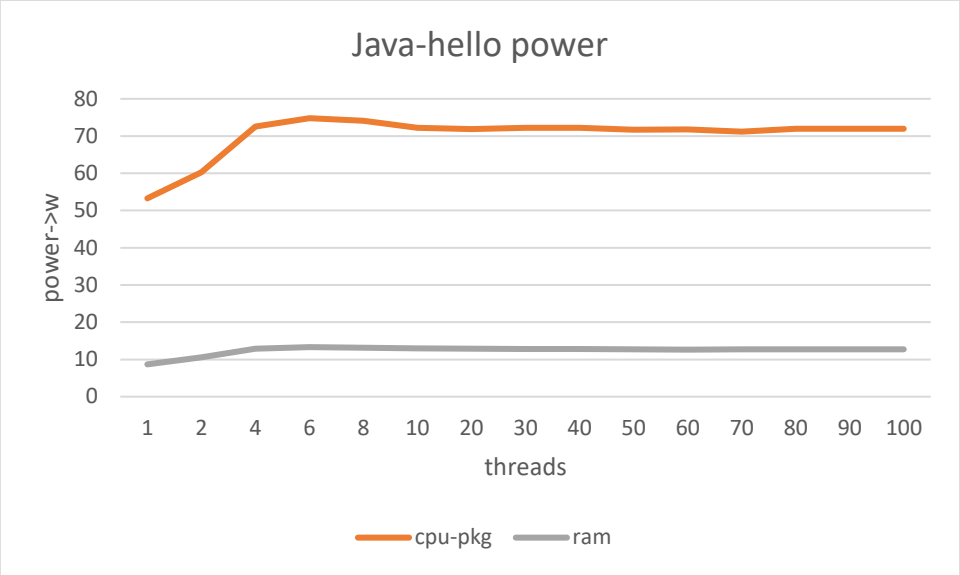


Σχήμα 4.1 Χρόνος εκτέλεσης του προγράμματος Java-hello για 1-100 νήματα

Στο Σχήμα 4.1 φαίνεται ο χρόνος εκτέλεσης του προγράμματος για 1-100 νήματα τα οποία δεν μοιράζονται το φόρτο εργασίας. Αν εκτελούνται Ν νήματα τότε γίνεται κλήση Ν actions στο OpenWhisk άρα εκτελείτε Ν φορές το ίδιο πρόγραμμα. Παρατηρούμε ότι από 1 μέχρι 4 νήματα ο χρόνος παραμένει σταθερός στα περίπου 112 ms και από 4 μέχρι 100 νήματα αυξάνεται από περίπου 122 ms ως 2130 ms. Για 1 μέχρι 40 νήματα θα αναμέναμε ο χρόνος εκτέλεσης να είναι σταθερός αφού έχουμε 40 λογικούς πυρήνες, αλλά έχουμε σταθερό χρόνο μόνο μέχρι τα 4 νήματα. Για 4 μέχρι 100 νήματα χρόνος εκτέλεσης μεγαλώνει και είναι ανάλογος με το αριθμό των νημάτων. Αυτό καταδεικνύει ότι τα προγράμματα μέχρι 4 νήματα εκτελούνται παράλληλα ενώ μετά τα 4 δεν εκτελούνται παράλληλα. Αυτό πιθανό να οφείλεται στο ότι ο χρόνος εκτέλεσης του προγράμματος είναι τόσο μικρός που όταν στέλνεται αίτημα για άλλη εκτέλεση νήματος οι προηγούμενες έχουν είδη τελειώσει, άρα δεν μπορεί να επιτευχθεί μεγαλύτερη παραλληλία.

Actions on openwhisk εκτός από NodeJS δεν έχουν intra-concurrency και αυτό ίσως εξηγεί την επιβράδυνση

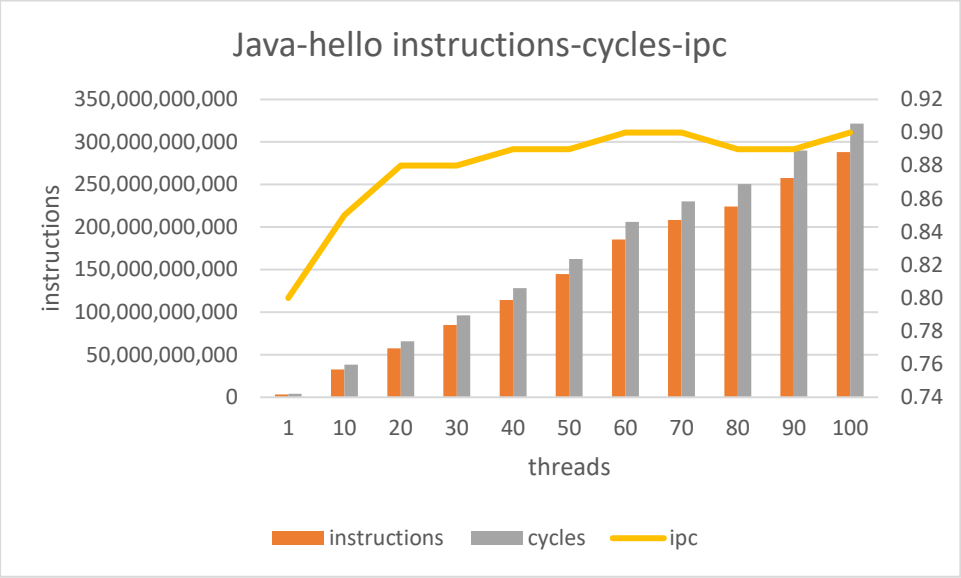
4.2.2 **Κατανάλωση ισχύς Java-hello**



Σχήμα 4.2 Ισχύς που καταναλώνεται πακέτο του επεξεργαστή και στην κύρια μνήμη του προγράμματος Java-hello για 1-100 νήματα

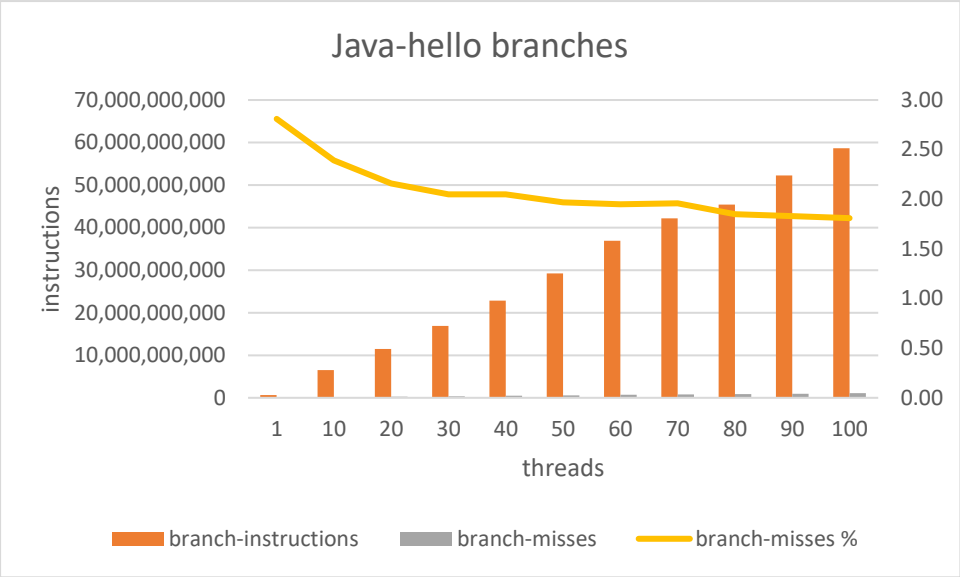
Στο Σχήμα 4.2 φαίνεται η ισχύς που καταναλώνεται από το πακέτο του επεξεργαστή και από την κύρια μνήμη του προγράμματος Java-hello για 1 μέχρι 100 νήματα. Η ισχύς που καταναλώνεται στο πακέτο του επεξεργαστή σε αδράνεια είναι περίπου 44.5 Watts και στην κύρια μνήμη είναι περίπου 6.1 Watts. Η ισχύς που καταναλώνεται από το πακέτο του επεξεργαστή αυξάνεται από περίπου 53 Watts για 1 νήμα σε περίπου 73 Watts για 4 νήματα. Για 4 μέχρι 100 νήματα η ισχύς που καταναλώνεται από το πακέτο του επεξεργαστή παραμένει σταθερή περίπου στα 72 Watts. Η ισχύς που καταναλώνεται από την κύρια μνήμη αυξάνεται από περίπου 8.6 Watts για 1 νήμα σε περίπου 12.8 Watts για 4 νήματα. Για 4 μέχρι 100 νήματα η ισχύς που καταναλώνεται από την κύρια μνήμη παραμένει σταθερή στα περίπου στα 13 Watts. Η αύξηση της ισχύς από 1 μέχρι 4 νήματα καταδεικνύει την παραλληλία μέχρι 4 νήματα, επειδή περισσότεροι πόροι χρησιμοποιούνται αυξάνεται και η ισχύς. Μετά τα 4 νήματα έχουμε σειριοποίηση της εκτέλεσης. Θα αναμέναμε τόσο στο πακέτο του επεξεργαστή όσο και στην κύρια μνήμη η ισχύς να αυξάνεται μέχρι τα 40 νήματα επειδή έχουμε 40 λογικούς πυρήνες, έτσι μέχρι να 40 νήματα θα χρησιμοποιούνταν περισσότεροι πυρήνες και θα καταναλωνόταν περισσότερη ισχύς. Αυτό δεν συμβαίνει πιθανό διότι ο χρόνος εκτέλεσης του προγράμματος είναι πολύ μικρός και έτσι δεν μπορεί να επιτευχθεί μεγάλη παραλληλία.

4.2.3 Performance counters Java-hello



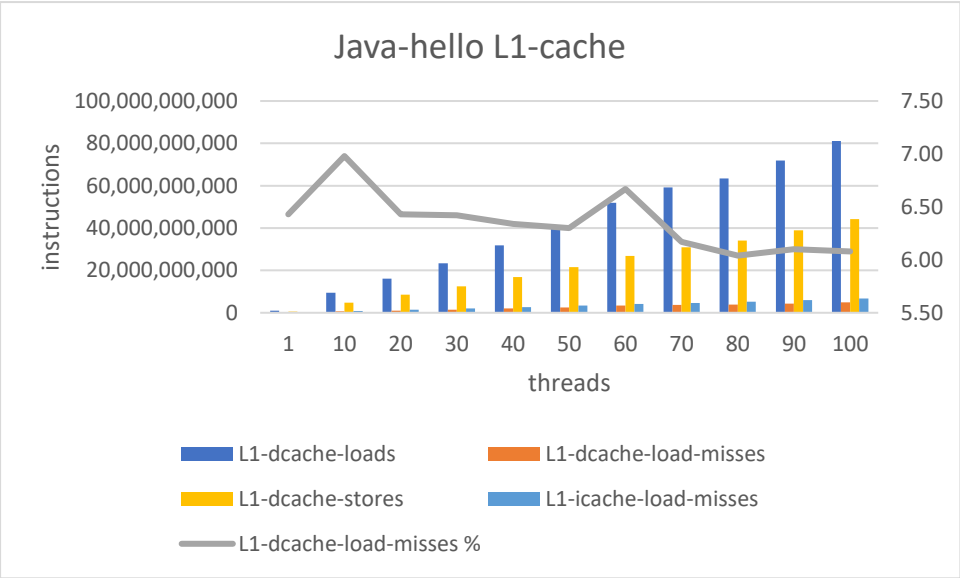
Σχήμα 4.3 Αριθμός εντολών, αριθμός κύκλων και αριθμός εντολών ανά αριθμό κύκλων του προγράμματος Java-hello για 1 μέχρι 100 νήματα

Στο Σχήμα 4.3 φαίνονται ο αριθμός των εντολών (instructions), ο αριθμός των κύκλων (cycles) και ο αριθμός των εντολών ως προς τον αριθμό των κύκλων (ipc) στο πρόγραμμα java hello από 1 μέχρι 100 νήματα. Ο αριθμός των εντολών για 1 νήμα είναι περίπου 3,2 δισεκατομμύρια και φτάνει για 100 νήματα περίπου στα 289 δισεκατομμύρια. Ο αριθμός των κύκλων για 1 νήμα είναι περίπου στα 4 δισεκατομμύρια και φτάνει περίπου στα 322 δισεκατομμύρια. Τόσο ο αριθμός των εντολών όσο και ο αριθμός των κύκλων είναι ανάλογος του αριθμού των νημάτων. Αυτό είναι λογικό αφού το ίδιο πρόγραμμα εκτελείτε περισσότερες φορές από πολλά νήματα. Το ipc αυξάνετε απότομα από το 1 νήμα στα 20 νήματα από το 0.80 στο 0.88 και μετά σταθεροποιείται. Η αύξηση του ipc πιθανό να οφείλεται στο ότι όταν έχουμε περισσότερα νήματα έχουμε και περισσότερες εντολές οι οποίες μπορούν να μπουν στο επεξεργαστή και έτσι να αυξήσουν το ipc. Επίσης ο αριθμός τόσο των εντολών όσο και των κύκλων είναι μεγάλος για ένα τόσο μικρό πρόγραμμα και οφείλεται στο ότι προσμετράται και η δημιουργία των κοντέινερς.



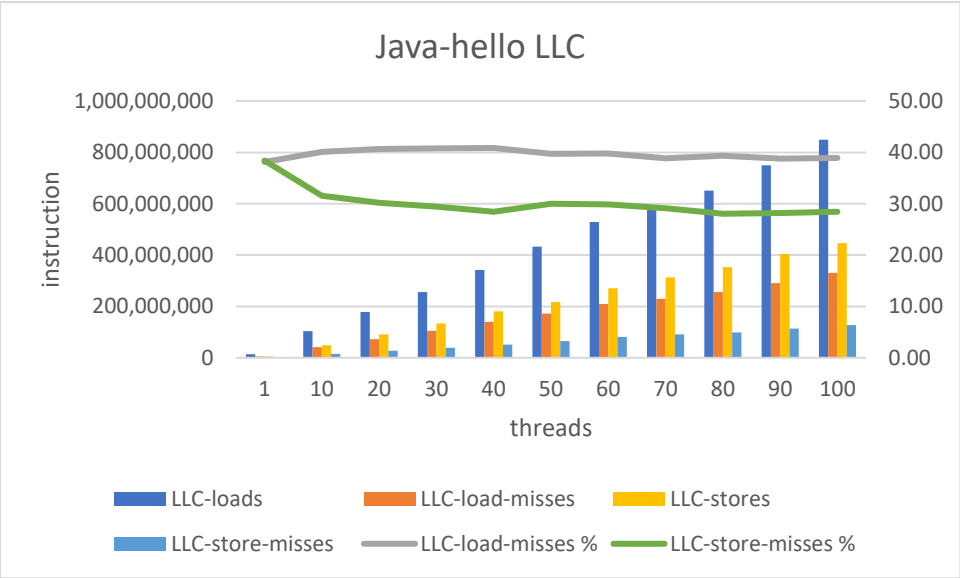
Σχήμα 4.4 Αριθμός εντολών διακλάδωσης, αριθμός αστοχιών εντολών διακλάδωσης και ποσοστό αστοχιών εντολών διακλάδωσης του προγράμματος Java-hello για 1 μέχρι 10 νήματα

Στο σχήμα 4.4 φαίνεται ο αριθμός των εντολών διακλάδωσης (branch-instuctions), ο αριθμός των αστοχιών για εντολές διακλάδωσης (branch-misses) και το ποσοστό αστοχιών εντολών διακλάδωσης (branch-misses %) για 1 μέχρι 100 νήματα. Ο αριθμός των εντολών διακλάδωσης αυξάνονται από περίπου 635 εκατομμύρια για 1 νήμα μέχρι περίπου 58 δισεκατομμύρια για 100 νήματα. Ο αριθμός αστοχιών για εντολές διακλάδωσης αυξάνονται από περίπου 17 εκατομμύρια για 1 νήμα σε περίπου 1 δισεκατομμύριο για 100 νήματα. Τόσο ο εντολές διακλάδωσης, όσο και οι αστοχίες εντολών διακλάδωσης είναι ανάλογες του αριθμού των νημάτων, πράγμα λογικό αφού το ίδιο πρόγραμμα τρέχει πολλές φορές αυξάνονται οι εντολές που εκτελούνται και επακόλουθα και οι εντολές διακλάδωσης. Το ποσοστό αποτυχίας για τις εντολές διακλάδωσης πέφτει από 2.81 για 1 νήμα σε 1.81 για 100 νήματα. Αυτή η μείωση πιθανό να σχετίζεται με το ότι περισσότερες ίδιες εντολές εκτελούνται ξανά και ξανά και έτσι ο branch predictor μπορεί να προβλέπει καλύτερα τις επόμενες εντολές. Σύμφωνα με το άρθρο “A Workload Characterization of the SPEC CPU2017 Benchmark Suite” [12] το ο μέσος όρο του ποσοστού αστοχίας των εντολών διακλάδωσης είναι λίγο πάνω από 3. Άρα το πρόγραμμα έχει λίγο μικρότερο ποσοστό, πράγμα που είναι λογικό αφού το πρόγραμμα έχει μόνο μία εντολή διακλάδωσης.



Σχήμα 4.5 Αριθμός φορτωμάτων L1-dcach, αριθμός αστοχιών φορτωμάτων L1-dcache, αριθμός αποθηκεύσεων L1-dcache, αριθμός αστοχιών φορτωμάτων L1-icache και ποσοστό ατοχέών φορτωμάτων L1-dcache του προγράμματος Java-hello για 1 μέχρι 100 νήματα

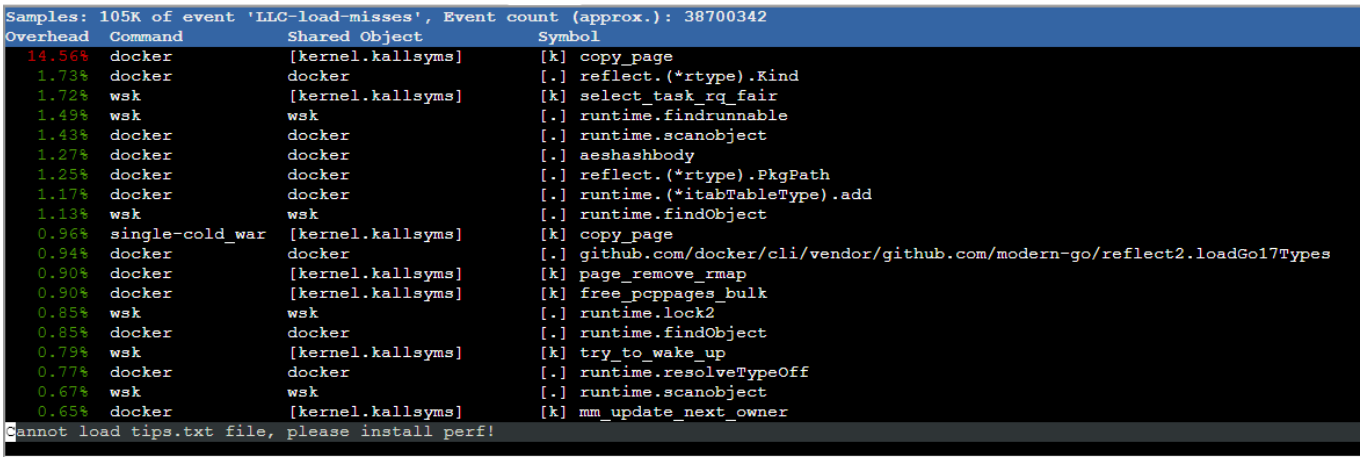
Στο Σχήμα 4.5 έχουμε τον αριθμό των φορτωμάτων (L1-dcache-loads) και αποθηκεύσεων (L1-dcache-stores) της L1-dcache κρυφής μνήμης, τον αριθμό των αστοχιών της L1-dcache των φορτωμάτων της L1-dcache και της L1-icache (L1-dcache-load-misses και L1-icache-load-misses) καθώς και το ποσοστό αστοχιών φορτώματος της L1-dcache (L1-dcache-load-misses %) για το πρόγραμμα Java hello για 1 μέχρι 100 νήματα. Τα L1-dcache-loads αυξάνονται από περίπου 922 εκατομμύρια για 1 νήμα σε περίπου 81 δισεκατομμύρια για 100 νήματα. Τα L1-dcache-stores αυξάνονται από περίπου 508 εκατομμύρια για 1 νήμα σε περίπου 44 δισεκατομμύρια για 100 νήματα. Τα L1-dcache-load-misses αυξάνονται από περίπου 59 εκατομμύρια για 1 νήμα σε περίπου 4.9 δισεκατομμύρια για 100 νήματα. Τα L1-icache-load-misses αυξάνονται από περίπου 100 εκατομμύρια για 1 νήμα σε περίπου 6.7 δισεκατομμύρια για 100 νήματα. Όλα αυτά είναι ανάλογα του αριθμού των νημάτων. Το ποσοστό L1-dcache-load-misses κυμαίνεται από 6.08 σε 6.67 και παραμένει περίπου σταθερό για όλες τις τιμές των νημάτων. Σύμφωνα με το άρθρο “A Workload Characterization of the SPEC CPU2017 Benchmark Suite” [12] το ποσοστό αυτό είναι λίγο πάνω από το μέσο όρο του ποσοστού L1-cache-misses.



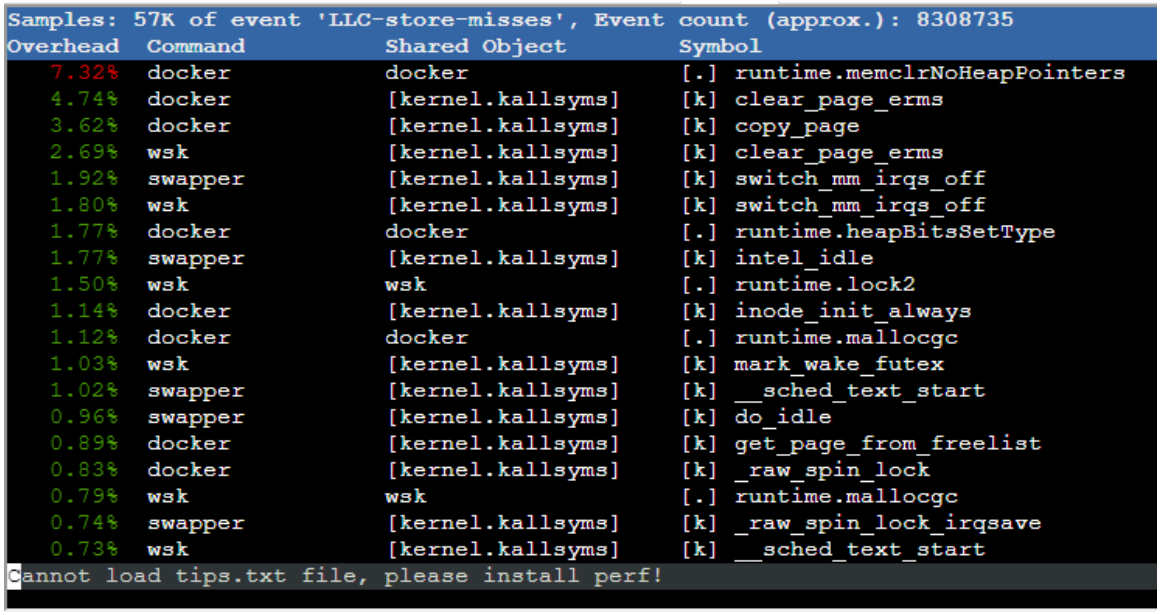
Σχήμα 4.6 ***Αριθμός φορτωμάτων, αριθμός αστοχίας φορτωμάτων, αριθμός αποθηκεύσεων, αριθμός αστοχίας αποθηκεύσεων, ποσοστό αστοχίας φορτωμάτων και ποσοστό αστοχίας αποθηκεύσεων τελευταίου επιπέδου κρυφής μνήμης του προγράμματος Java-hello για 1 μέχρι 100 νήματα***

Στο Σχήμα 4.6 φαίνεται ο αριθμός των φορτωμάτων (LLC-loads), ο αριθμός αστοχιών φορτωμάτων (LLC-load-misses), ο αριθμός αποθηκεύσεων (LLC-stores), ο αριθμός αστοχιών αποθηκεύσεων (LLC-store-misses), το ποσοστό αστοχίας φορτωμάτων (LLC-load-misses %) και το ποσοστό αστοχίας αποθηκεύσεων (LLC-store-misses %) του τελευταίου επιπέδου κρυφής μνήμης. Ο αριθμός των φορτωμάτων κρυφής μνήμης αυξάνονται από περίπου 14 εκατομμύρια για 1 νήμα σε περίπου 849 εκατομμύρια για 100 νήματα. Ο αριθμός των αποθηκεύσεων κρυφής μνήμης αυξάνονται από περίπου 5 εκατομμύρια για 1 νήμα σε περίπου 446 εκατομμύρια για 100 νήματα. Ο αριθμός των αστοχιών φορτωμάτων κρυφής μνήμης αυξάνονται από περίπου 5 εκατομμύρια για 1 νήμα σε περίπου 330 εκατομμύρια για 100 νήματα. Ο αριθμός των αστοχιών αποθηκεύσεων κρυφής μνήμης αυξάνονται από περίπου 2 εκατομμύρια για 1 νήμα σε περίπου 127 εκατομμύρια για 100 νήματα. Τα τέσσερα αυτά μετρικά είναι ανάλογα του αριθμού των νημάτων, πράγμα λογικό αφού και ο αριθμός των εντολών που εκτελούνται αυξάνεται ανάλογα. Το ποσοστό αστοχίας στο τελευταίο επίπεδο κρυφής μνήμης για φορτώματα είναι περίπου 40% σε όλο το φάσμα των τιμών των νημάτων. Το ποσοστό αστοχίας στο τελευταίο επίπεδο κρυφής μνήμη για αποθηκεύσεις είναι περίπου 40% για 1 νήμα και μειώνεται στο 30% για 10 νήματα και παραμένει σταθερό μέχρι και τα 100 νήματα. Η μείωση από 1 μέχρι 10 νήματα πιθανό να οφείλεται στο το ίδιο πρόγραμμα εκτελείτε πολλές φορές. Σύμφωνα με το άρθρο “A Workload

Characterization of the SPEC CPU2017 Benchmark Suite" [12] ο μέσος όρος για αυτές τις αστοχίες είναι κοντά στο 15%. Τα ποσοστά αστοχίας του προγράμματος είναι αρκετά μεγαλύτερα. Αυτό με έκανε να ψάξω περεταίρω την αιτία αυτού και έτσι χρησιμοποίησα το perf record και το perf report. Όπως φαίνεται στο Σχήμα 4.7 και Σχήμα 4.8 τα υψηλά ποσοστά οφείλονται σε μεγάλο βαθμό στο docker. Ο αριθμός των αστοχιών φορτώματος κυμαίνεται από 1.1 μέχρι 1.6 ανά 1000 εντολές και το ποσοστό αστοχίας για τις αποθηκεύσεις κυμαίνεται από 0.4 μέχρι 0.6 ανά 1000. Αυτά τα ποσοστά είναι χαμηλά και καταδεικνύουν ότι οι αστοχίες αυτές δεν έχουν μεγάλο αντίκτυπο στην επίδοση.



Σχήμα 4.7 Συναρτήσεις που προκαλούν αστοχίες φορτώματος στο τελευταίο επίπεδο την κρυφής μνήμης για το πρόγραμμα Java-hello

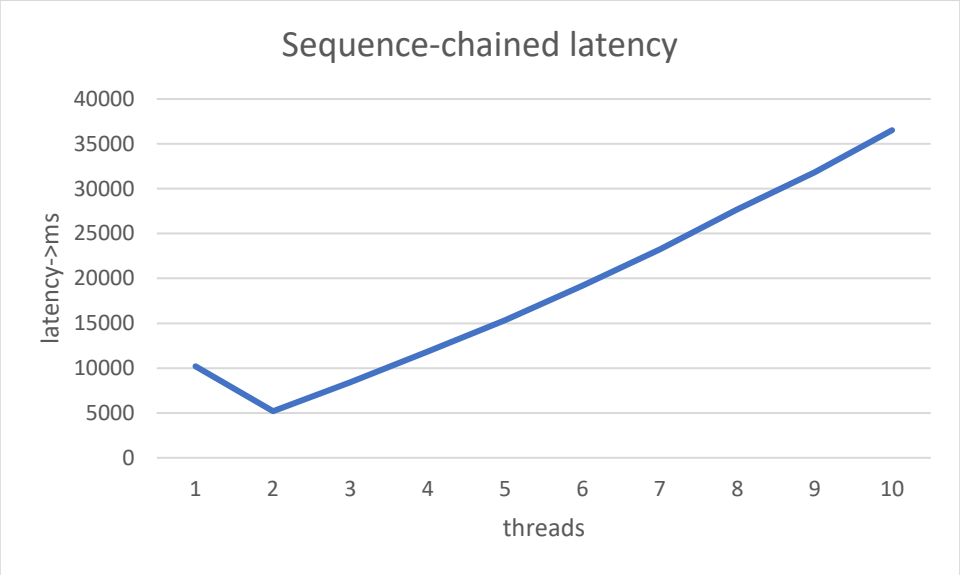


Σχήμα 4.8 Συναρτήσεις που προκαλούν αστοχίες αποθήκευσης στο τελευταίο επίπεδο την κρυφής μνήμης για το πρόγραμμα Java-hello

4.3 Εκτέλεση προγράμματος Sequence-chained

Το επόμενο πρόγραμμα που εξέτασα είναι το Sequence-chained που βρίσκεται κάτω από το φάκελο Testcase3-Long-function-chain του ServerlessBench. Το πρόγραμμα αυτό χρησιμοποιεί την δυνατότητα που δίνεται από το OpenWhisk να συνδέει με αλυσίδα actions χωρίς να χρειάζεται να γραφεί περεταίρω κώδικας, περνώντας την έξοδο του ενός action ως είσοδο του άλλου. Συγκεκριμένα υπάρχει μια αλυσίδα 100 actions, όπου κάθε action επιστρέφει το n που είναι ο αριθμός του action στην αλυσίδα, ένα πίνακα startTimes στον οποίο προστίθεται σε κάθε κλήση action ο χρόνος έναρξης του και ένας πίνακας retTimes στον οποίο προστίθεται σε κάθε κλήση action ο χρόνος που τελειώνει την εκτέλεση του. Έτσι κάθε πίνακας startTimes των action περιέχει όλους του χρόνους έναρξης των actions που προηγήθηκαν στην αλυσίδα και το retimes όλους του χρόνους που τελειώνουν αυτά.

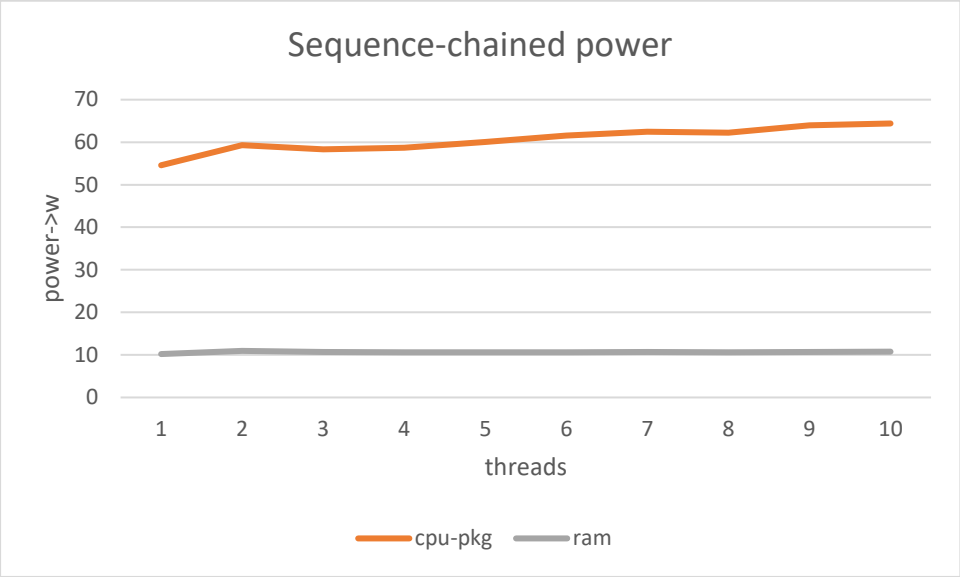
4.3.1 Χρόνος εκτέλεσης Sequence-chained



Σχημα 4.9 Χρόνος εκτέλεσης του προγράμματος Sequence-chained για 1-100 νήματα

Στο Σχήμα 4.9 φαίνεται ο χρόνος εκτέλεσης του προγράμματος Sequence-chained για 1-10 νήματα τα οποία δεν μοιράζονται το φόρτο εργασίας. Αν εκτελούνται N νήματα τότε γίνεται κλήση N actions στο OpenWhisk άρα εκτελείτε N φορές το ίδιο πρόγραμμα. Ο χρόνος εκτέλεσης για 1 νήμα είναι περίπου 10000 ms και μειώνεται στα περίπου 5200 ms για 2 νήματα. Αυξάνεται από περίπου 5200 ms για 2 νήματα σε περίπου 36500 ms για 10 νήματα. Η ύπαρξη μεγαλύτερου χρόνου εκτέλεσης για 1 νήμα σε σχέση με τα 2 νήματα πιθανό να οφείλεται στο cold start, δηλαδή το χρόνο αρχικοποίησης των κοντέινερς. Μετά από τα 2 νήματα αυξάνεται ο χρόνος λόγο της αύξησης του φόρτου εργασίας.

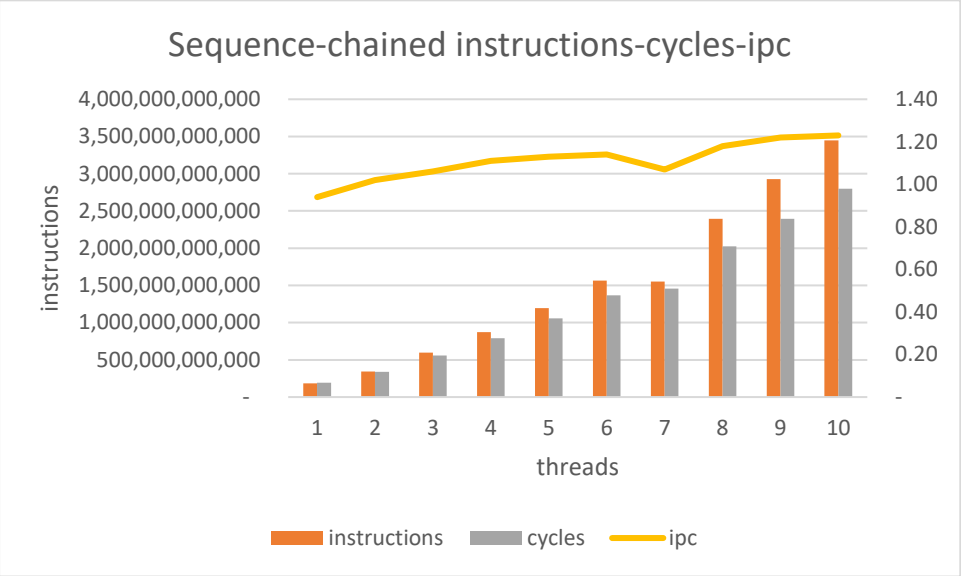
4.3.2 Κατανάλωση ισχύς Sequence-chained



Σχήμα 4.10 Κατανάλωση ισχύς από το πακέτου του επεξεργαστή και από την κύρια μνήμη για 1-100 νήματα

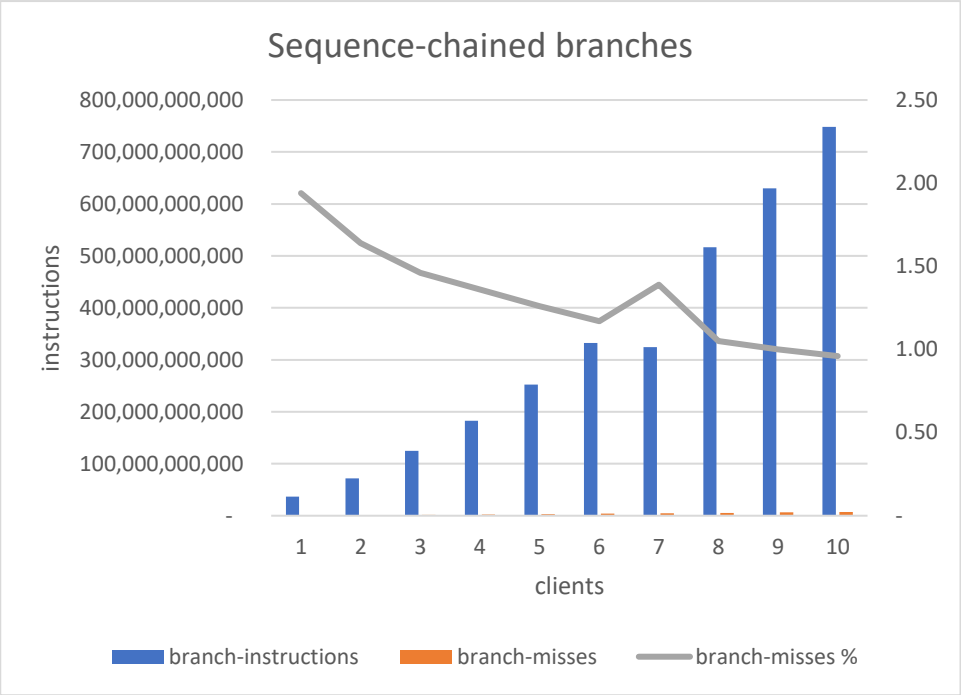
Στο Σχήμα 4.10 φαίνεται η ισχύς που καταναλώνεται στο πακέτο του επεξεργαστή (cpu-pkg) και στην κύρια μνήμη (ram) του προγράμματος Sequence-chained για 1 μέχρι 10 νήματα. Η ισχύς που καταναλώνεται στο πακέτο του επεξεργαστή σε αδράνεια είναι περίπου 44.5 Watts και στην κύρια μνήμη είναι περίπου 6.1 Watts. Η ισχύς που καταναλώνεται για 1 νήμα από το πακέτο του επεξεργαστή είναι περίπου 55 Watts. Για 2 νήματα αυξάνεται στα περίπου 60 Watts όπου και παραμένει περίπου σταθερό μέχρι τα 10 νήματα. Η ισχύς που καταναλώνεται από την κύρια μνήμη για 1 νήμα είναι περίπου 10.2 Watts. Για 2 νήματα αυξάνεται στα περίπου 10.7 Watts όπου παραμένει σταθερή μέχρι τα 10 νήματα. Η μικρότερη κατανάλωση ισχύς στο 1 νήμα πιθανό να οφείλεται στο ότι τρέχει περισσότερο χρόνο για να εκτελέσει την ίδια δουλεία έτσι η ενέργεια που καταναλώνεται στον ίδιο χρόνο είναι μικρότερη. Η ισχύς παραμένει σταθερή για 2 μέχρι 10 νήματα πιθανό διότι κάθε νήμα τρέχει 100 αλυσιδωτά actions, δεν μπορούν να εκτελεστούν παράλληλα και έτσι αφού όλοι οι πόροι χρησιμοποιούνται δεν μπορεί να αυξηθεί περεταίρω η ισχύς.

4.3.3 Performance counters Sequence-chained



Σχήμα 4.11 Αριθμός εντολών, αριθμός κύκλων και αριθμός εντολών ανά αριθμό κύκλων του προγράμματος Sequence-chained για 1 μέχρι 10 νήματα

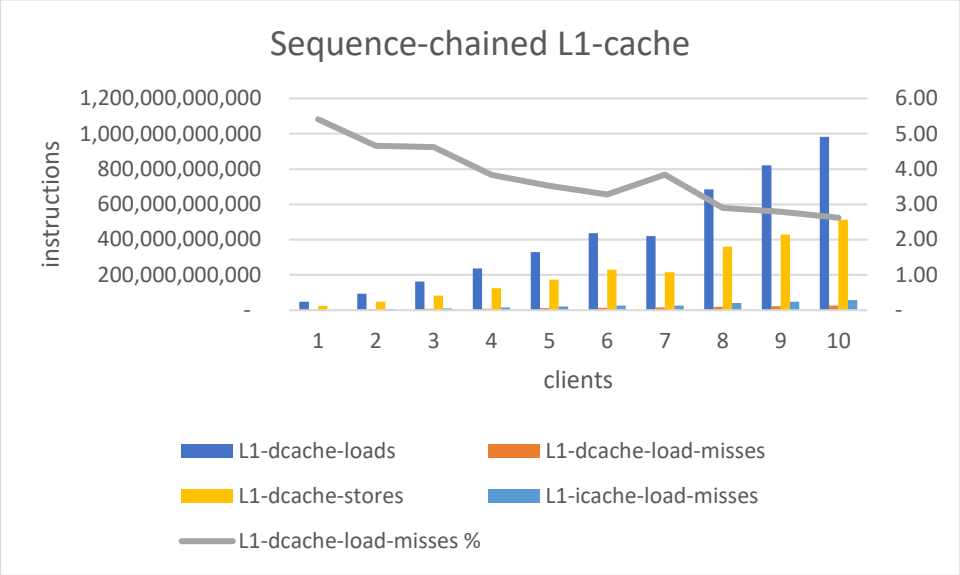
Στο Σχήμα 4.11 φαίνονται ο αριθμός των εντολών (instructions), ο αριθμός των κύκλων (cycles) και ο αριθμός των εντολών ως προς τον αριθμό των κύκλων (ipc) στο πρόγραμμα Sequence-chained από 1 μέχρι 10 νήματα. Ο αριθμός των εντολών για 1 νήμα είναι περίπου 182 δισεκατομμύρια και φτάνει περίπου στα 3.5 τρισεκατομμύρια για 10 νήματα. Ο αριθμός των κύκλων για 1 νήμα είναι περίπου στα 193 δισεκατομμύρια και φτάνει περίπου στα 2.8 τρισεκατομμύρια. Τόσο ο αριθμός των εντολών όσο και ο αριθμός των κύκλων είναι ανάλογος του αριθμού των νημάτων. Αυτό είναι λογικό αφού το ίδιο πρόγραμμα εκτελείτε περισσότερες φορές από πολλά νήματα. Το ipc αυξάνετε από το 0.94 για 1 νήμα, σε 1.23 για 10 νήματα. Η αύξηση του ipc πιθανό να οφείλεται στο ότι με μεγαλύτερο αριθμό νημάτων εκτελούνται περισσότερες εντολές, άρα μπαίνουν στον επεξεργαστή περισσότερες εντολές για εκτέλεση και έτσι αυξάνεται το ipc. Επίσης ο αριθμός τόσο των εντολών όσο και των κύκλων είναι μεγαλύτερος από θα περιμέναμε για αυτό το πρόγραμμα και αυτό οφείλεται στο ότι προσμετράται και η δημιουργία των κοντέινερς.



Σχήμα 4.12 Αριθμός ενολών διακλάδωσης, αριθμός αστοχιών εντολών διακλωδωσης, ποσοστό ασοχίων εντολών διακλάδωσης του προγράμματος Sequence-chained για 1 μέχρι 10 νήματα

Στο Σχήμα 4.12 φαίνεται ο αριθμός των εντολών διακλάδωσης (branch-instuctions), ο αριθμός των αστοχιών για εντολές διακλάδωσης (branch-misses) και το ποσοστό αστοχιών εντολών διακλάδωσης (branch-misses %) για το πρόγραμμα Sequence-chained για 1 μέχρι 10 νήματα. Ο αριθμός των εντολών διακλάδωσης αυξάνονται από περίπου 37 δισεκατομμύρια για 1 νήμα μέχρι περίπου 748 δισεκατομμύρια για 10 νήματα. Ο αριθμός αστοχιών για εντολές διακλάδωσης αυξάνονται από περίπου 716 εκατομμύρια για 1 νήμα σε περίπου 7.1 δισεκατομμύρια για 10 νήματα. Τόσο ο εντολές διακλάδωσης, όσο και οι αστοχίες εντολών διακλάδωσης είναι ανάλογες του αριθμού των νημάτων, πράγμα λογικό αφού το ίδιο πρόγραμμα τρέχει πολλές φορές αυξάνονται οι εντολές που εκτελούνται και επακόλουθα και οι εντολές διακλάδωσης. Το ποσοστό αποτυχίας για τις εντολές διακλάδωσης πέφτει από 1.94 για 1 νήμα σε 0.96 για 10 νήματα. Αυτή η μείωση πιθανό να σχετίζεται με το ότι περισσότερες ίδιες εντολές εκτελούνται ξανά και ξανά και έτσι ο branch predictor μπορεί να προβλέπει καλύτερα τις επόμενες εντολές. Σύμφωνα με το άρθρο “A Workload Characterization of the SPEC CPU2017 Benchmark Suite” [12] ο μέσος όρο του ποσοστού αστοχίας των εντολών διακλάδωσης είναι λίγο πάνω από 3. Άρα το πρόγραμμα έχει λίγο

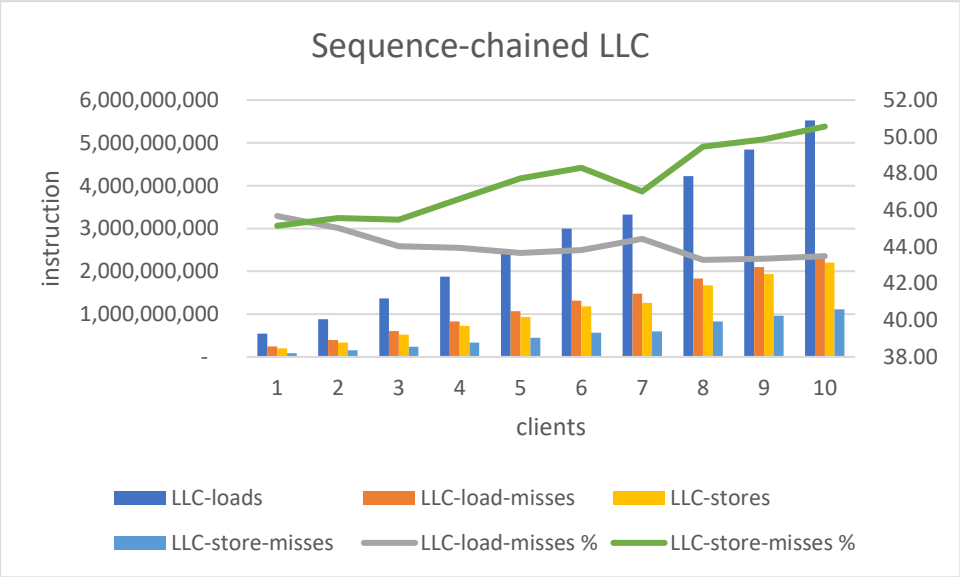
μικρότερο ποσοστό, πράγμα που είναι λογικό αφού το πρόγραμμα έχει μόνο δύο εντολές διακλάδωσης.



Σχήμα 4.13 Αριθμός φορτωμάτων της L1-dcache, αριθμός αστοχίας φορτωμάτων της L1-dcache, αριθμός αποθηκεύσεων L1-dcache, αριθμός αστοχίας φορτωμάτων L1-icache και ποσοστό αστοχίας φορτωμάτων L1-dcache του προγράμματος Sequence-chained για 1 μέχρι 10 νήματα

Στο Σχήμα 4.13 έχουμε τον αριθμό των φορτωμάτων (L1-dcache-loads) και αποθηκεύσεων (L1-dcache-stores) της L1-dcache κρυφής μνήμης, τον αριθμό των αστοχιών της L1-dcache των φορτωμάτων της L1-dcache και της L1-icache (L1-dcache-load-misses και L1-icache-load-misses) καθώς και το ποσοστό αστοχιών φορτώματος της L1-dcache (L1-dcache-load-misses %) για το πρόγραμμα Sequence-chained n για 1 μέχρι 10 νήματα. Τα L1-dcache-loads αυξάνονται από περίπου 48 δισεκατομμύρια για 1 νήμα σε περίπου 482 δισεκατομμύρια για 10 νήματα. Τα L1-dcache-stores αυξάνονται από περίπου 25 δισεκατομμύρια για 1 νήμα σε περίπου 513 δισεκατομμύρια για 10 νήματα. Τα L1-dcache-load-misses αυξάνονται από περίπου 2.6 δισεκατομμύρια για 1 νήμα σε περίπου 26 δισεκατομμύρια για 10 νήματα. Τα L1-icache-load-misses αυξάνονται από περίπου 3.5 δισεκατομμύρια για 1 νήμα σε περίπου 56 δισεκατομμύρια για 10 νήματα. Όλα αυτά είναι ανάλογα του αριθμού των νημάτων, πράγμα λογικό αφού εκτελείτε το ίδιο πρόγραμμα περισσότερες φορές. Το ποσοστό L1-dcache-load-misses μειώνεται από 5.41 για 1 νήμα σε 2.62 για 10 νήματα. Αυτό πιθανό να οφείλεται στο ότι επειδή εκτελείτε το ίδιο πρόγραμμα ενδεχόμενος να υπάρχουν τα δεδομένα ήδη στην L1. Σύμφωνα με το άρθρο “A Workload Characterization of the

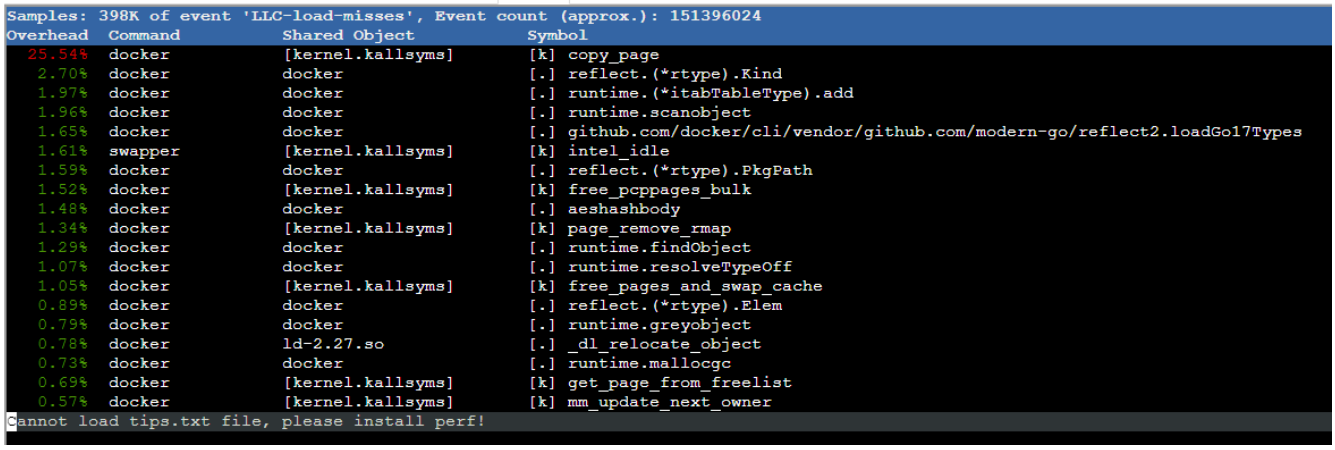
SPEC CPU2017 Benchmark Suite" [12] τα ποσοστά αυτά είναι κοντά στο μέσο όρο του ποσοστού L1-cache-misses.



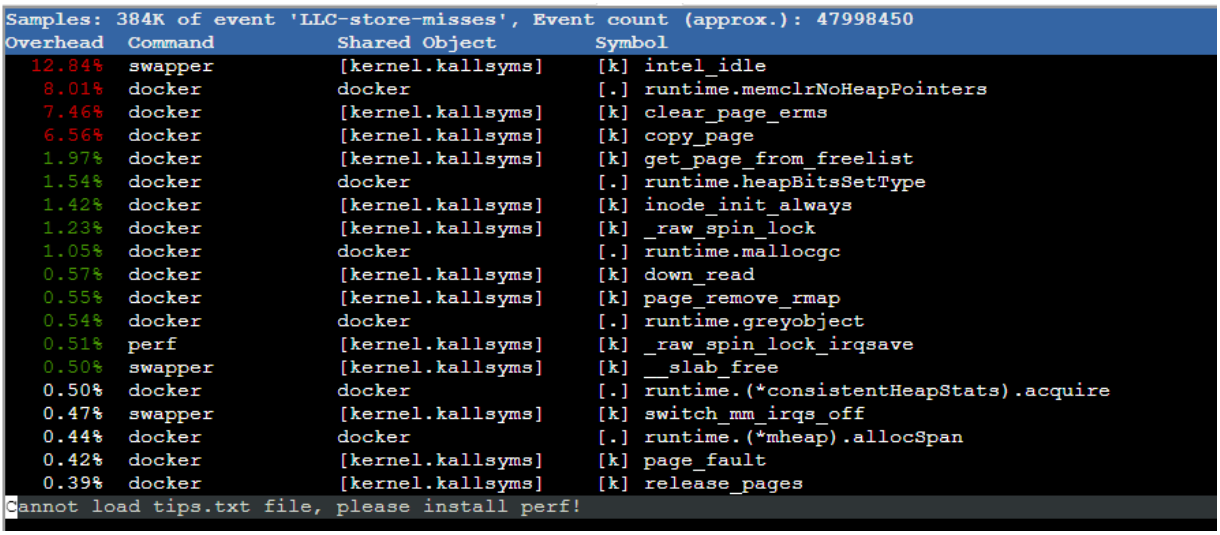
Σχήμα 4.14 Αριθμός φορτωμάτων, αριθμός αστοχίας φορτωμάτων, αριθμός αποθηκεύσεων, αριθμός αστοχίας αποθηκεύσεων, ποσοστό αστοχίας φορτωμάτων και ποσοστό αστοχίας αποθηκεύσεων τελευταίου επιπέδου κρυφής μνήμης για το πρόγραμμα Sequence-chained για 1 μέχρι 10 νήματα

Στο Σχήμα 4.14 φαίνεται ο αριθμός των φορτωμάτων (LLC-loads), ο αριθμός αστοχιών φορτωμάτων (LLC-load-misses), ο αριθμός αποθηκεύσεων (LLC-stores), ο αριθμός αστοχιών αποθηκεύσεων (LLC-store-misses), το ποσοστό αστοχίας φορτωμάτων (LLC-load-misses %) και το ποσοστό αστοχίας αποθηκεύσεων (LLC-store-misses %) του τελευταίου επιπέδου κρυφής μνήμης για το πρόγραμμα Sequence-chained για 1 μέχρι 10 νήματα. Ο αριθμός των φορτωμάτων κρυφής μνήμης αυξάνονται από περίπου 542 εκατομμύρια για 1 νήμα σε περίπου 5.5 δισεκατομμύρια για 10 νήματα. Ο αριθμός των αποθηκεύσεων κρυφής μνήμης αυξάνονται από περίπου 196 εκατομμύρια για 1 νήμα σε περίπου 2.2 δισεκατομμύρια για 10 νήματα. Ο αριθμός των αστοχιών φορτωμάτων κρυφής μνήμης αυξάνονται από περίπου 247 εκατομμύρια για 1 νήμα σε περίπου 2.4 δισεκατομμύρια για 10 νήματα. Ο αριθμός των αστοχιών αποθηκεύσεων κρυφής μνήμης αυξάνονται από περίπου 88 εκατομμύρια για 1 νήμα σε περίπου 1.1 δισεκατομμύρια για 10 νήματα. Τα τέσσερα αυτά μετρικά είναι ανάλογα του αριθμού των νημάτων, πράγμα λογικό αφού και ο αριθμός των εντολών που εκτελούνται

αυξάνεται ανάλογα. Το ποσοστό αστοχίας στο τελευταίο επίπεδο κρυφής μνήμης για φορτώματα μειώνεται από 45.68% για 1 νήμα σε 43.49% για 10 νήματα. Το ποσοστό αστοχίας στο τελευταίο επίπεδο κρυφής μνήμη για αποθηκεύσεις αυξάνεται από 45.15% για 1 νήμα σε 50.56% για 10 νήματα. Σύμφωνα με το άρθρο “A Workload Characterization of the SPEC CPU2017 Benchmark Suite” [12] ο μέσος όρος για αυτές τις αστοχίες είναι κοντά στο 15%. Τα ποσοστά αστοχίας του προγράμματος είναι αρκετά μεγαλύτερα. Αυτό με έκανε να ψάξω περεταίρω την αιτία αυτού και έτσι χρησιμοποίησα το perf record και το perf report. Όπως φαίνεται στα σχήματα 4.15 και 4.16 τα υψηλά ποσοστά οφείλονται σε μεγάλο βαθμό στο docker και στη διεργασία swapper, η οποία είναι μια διεργασία που τρέχει όταν καμία άλλη διεργασία δεν τρέχει, για τις αστοχίες φορτώματος και στο docker για τις αστοχίες αποθήκευσης του τελευταίου επιπέδου κρυφής μνήμης. Ο αριθμός αστοχιών φορτωμάτων κυμαίνεται από 0.7 μέχρι 1.4 ανά 1000 εντολές και ο αριθμός αστοχιών αποθήκευσης κυμαίνεται από 0.3 μέχρι 0.5 ανά 1000 εντολές. Αυτά τα ποσοστά είναι μικρά άρα δεν έχουν μεγάλο αντίκτυπο στην επίδοση.



Σχήμα 4.15 Συναρτήσεις που προκαλούν αστοχίες φορτωμάτων στο τελευταίο επίπεδο κρυφής μνήμης για το πρόγραμμα Sequence-chained

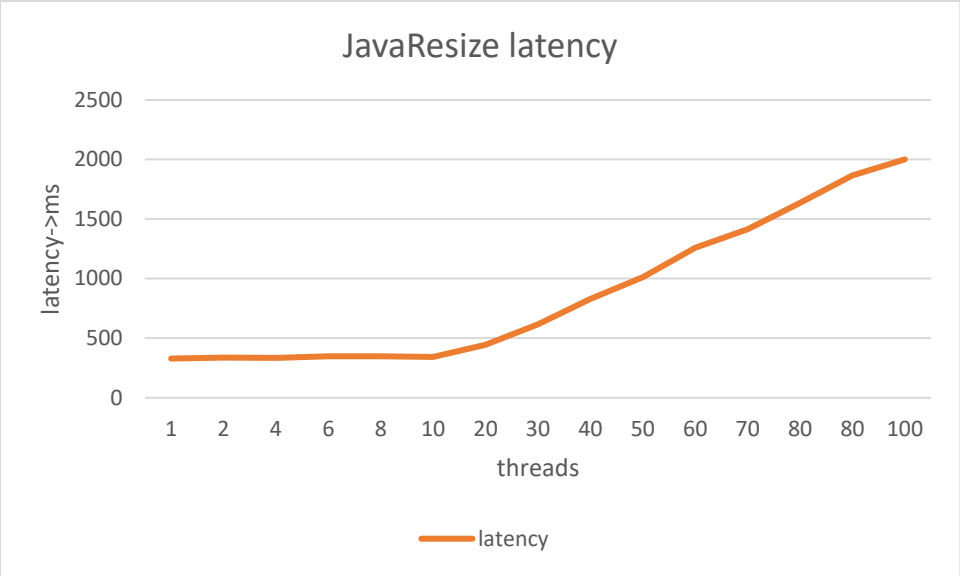


Σχήμα 4.16 Συναρτήσεις που προκαλούν αστοχίες αποθηκεύσεων στο τελευταίο επίπεδο κρυφής μνήμης για το πρόγραμμα Sequence-chained

4.4 Εκτέλεση προγράμματος JavaResize

Το τελευταίο πρόγραμμα που εξέτασα στο OpenWhisk είναι το JavaResize που βρίσκεται κάτω από το φάλεοTastcase10-Stateless-cost του ServelessBench. Το πρόγραμμα αυτό παίρνει μια εικόνα και αλλάζει το μέγεθος της. Συγκεκριμένα παίρνει σαν παραμέτρους από ένα json αρχείο μια εικόνα σε κωδικοποίηση BASE64, το νέο ύψος και το νέο πλάτος της εικόνας, της αλλάζει το μέγεθος και επιστρέφει την νέα εικόνα σε κωδικοποίηση BASE64.

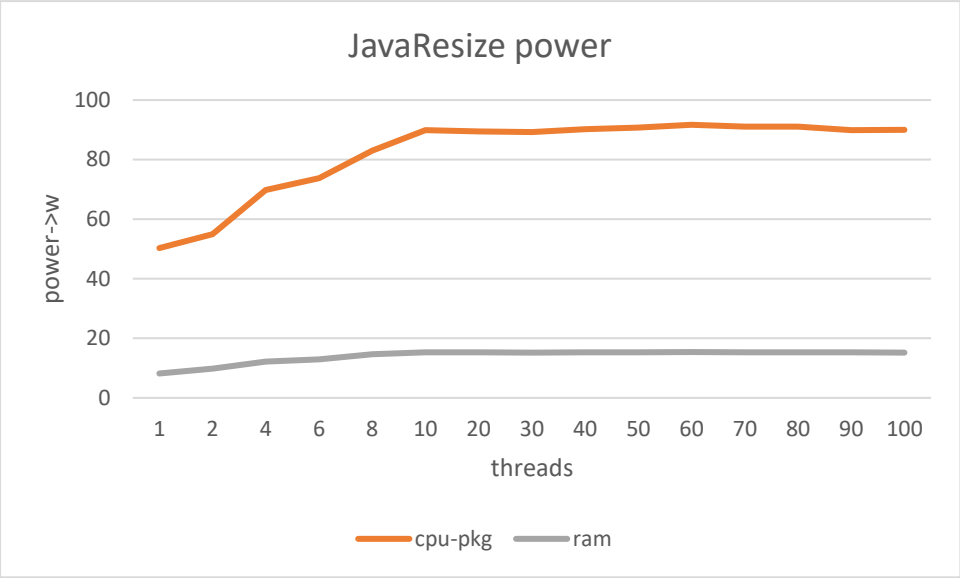
4.4.1 Χρόνος εκτέλεσης JavaResize



Σχήμα 4.16 Χρόνος εκτέλεσης του προγράμματος JavaResize για 1 μέχρι 100 νήματα

Στο σχήμα 4.16 φαίνεται ο χρόνος εκτέλεσης του προγράμματος JavaResize για 1-100 νήματα τα οποία δεν μοιράζονται το φόρτο εργασίας. Αν εκτελούνται N νήματα τότε γίνεται κλήση N actions στο OpenWhisk άρα εκτελείτε N φορές το ίδιο πρόγραμμα. Παρατηρούμε ότι έχουμε δύο μέρη στη γραφική, από 1 μέχρι 10 νήματα από περίπου ο χρόνος εκτέλεσης παραμένει σταθερός στα περίπου 340 ms και από περίπου 340 ms για 10 νήματα ως περίπου 2000 ms για 100 νήματα. Θα περιμέναμε ο χρόνος να παραμένει σταθερός μέχρι τα 40 νήματα δηλαδή να υπήρχε παράλληλη εκτέλεση μέχρι τα 40 νήματα γιατί έχουμε 40 λογικούς πυρήνες. Παρόλα αυτά έχουμε παράλληλη εκτέλεση μέχρι 10 νήματα και μετά σειριοποιήται η εκτέλεση των actions. Αυτό πιθανό να οφείλεται στο ότι ο χρόνος εκτέλεσης του προγράμματος είναι τόσο μικρός που όταν στέλνεται αίτημα για άλλη εκτέλεση νήματος οι προηγούμενες έχουν ήδη τελειώσει, άρα δεν μπορεί να επιτευχθεί μεγαλύτερη παραλληλία.

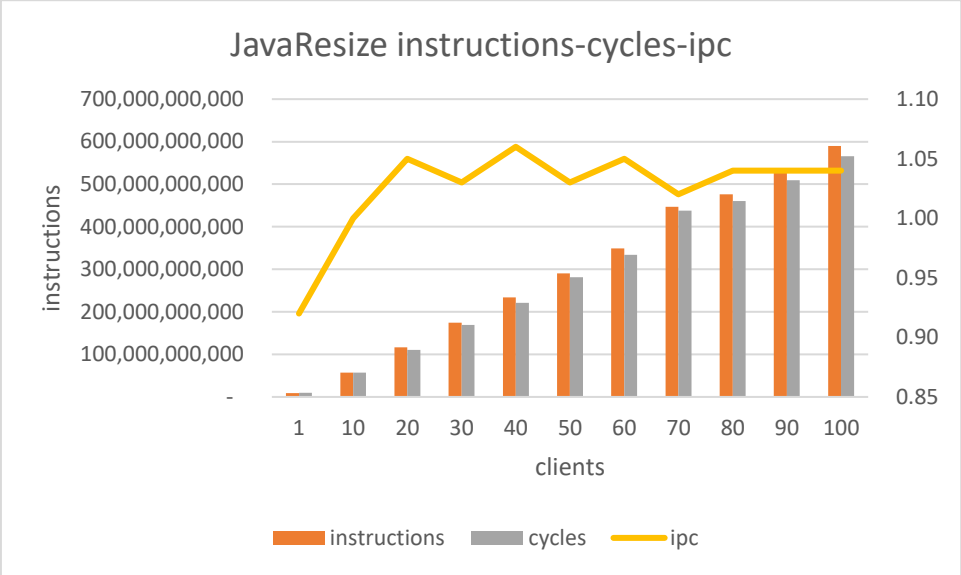
4.4.2 Κατανάλωση ισχύς JavaResize



Σχήμα 4.17 κατανάλωση ισχύς από το πακέτο του επεξεργαστή και πό την κύρια μνήμη για 1-100 νήματα

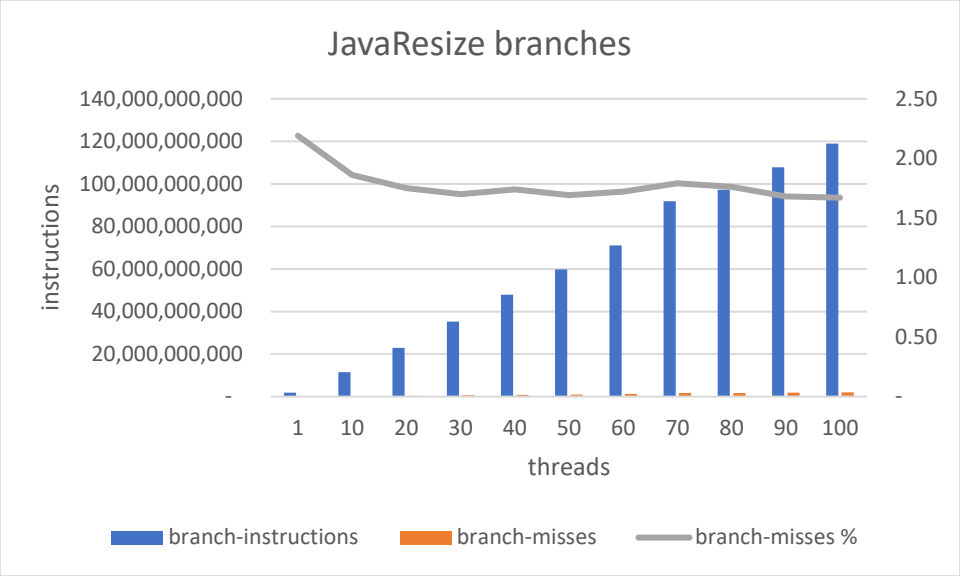
Στο Σχήμα 4.17 φαίνεται η ισχύς που καταναλώνεται από το πακέτο του επεξεργαστή (cpu-pkg) και από την κύρια μνήμη (ram) για 1 μέχρι 100 νήματα για το πρόγραμμα ImageResize. Η ισχύς που καταναλώνεται στο πακέτο του επεξεργαστή σε αδράνεια είναι περίπου 44.5 Watts και στην κύρια μνήμη είναι περίπου 6.1 Watts. Έχουμε δύο μέρη στη γραμμή του cpu-pkg, από 1 μέχρι 10 νήματα και από 10 μέχρι 100 νήματα. Στο πρώτο μέρος η ισχύς στο πακέτο του επεξεργαστή αυξάνεται από περίπου 50 Watts μέχρι περίπου 90 Watts και στο δεύτερο από παραμένει σταθερή στα περίπου 90 Watts. Η ισχύς που καταναλώνεται από την κύρια μνήμη αυξάνεται από περίπου 8.1 Watts για 1 νήμα μέχρι περίπου 15.3 Watts για 10 νήματα και παραμένει σταθερή στα περίπου 15.3 Watts από 10 μέχρι 100 νήματα. Από 1 μέχρι 10 νήματα λόγο έχουμε αύξηση της ισχύς τόσο στον επεξεργαστή όσο και στην κύρια μνήμη και αυτό καταδεικνύει την παραλληλία μέχρι 10 νήματα. Μετά τα 10 νήματα σειριοποιήται η εκτέλεση των προγραμμάτων. Θα περιμέναμε η ισχύς να αυξάνεται μέχρι τα 40 νήματα, επειδή υπάρχουν 40 λογικοί πυρήνες, πράγμα που δεν συμβαίνει λόγο του μικρού χρόνου εκτέλεσης του προγράμματος.

4.4.3 Performance counters JavaResize



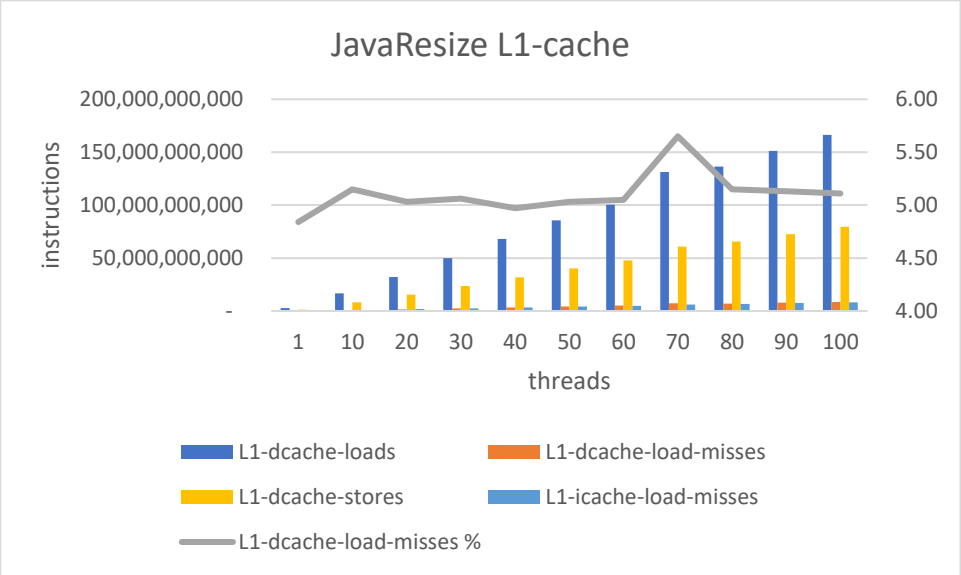
Σχήμα 4.18 Αριθμός εντολών, αριθμός κύκλων και αριθμός εντολών ανά αριθμό κύκλων του προγράμματος JavaResize για 1 μέχρι 100 νήματα

Στο Σχήμα 4.18 φαίνονται ο αριθμός των εντολών (instructions), ο αριθμός των κύκλων (cycles) και ο αριθμός των εντολών ως προς τον αριθμό των κύκλων (ipc) στο πρόγραμμα ImageResize από 1 μέχρι 100 νήματα. Ο αριθμός των εντολών για 1 νήμα είναι περίπου 8.7 δισεκατομμύρια και φτάνει για 100 νήματα περίπου στα 590 δισεκατομμύρια. Ο αριθμός των κύκλων για 1 νήμα είναι περίπου στα 9.4 δισεκατομμύρια και φτάνει περίπου στα 566 δισεκατομμύρια για 100 νήματα. Τόσο ο αριθμός των εντολών όσο και ο αριθμός των κύκλων είναι ανάλογος του αριθμού των νημάτων. Αυτό είναι λογικό αφού το ίδιο πρόγραμμα εκτελείτε περισσότερες φορές από πολλά νήματα. Το ipc αυξάνετε απότομα από το 1 νήμα στα 20 νήματα από το 0.92 στο 1.05 και μετά σταθεροποιείται Η αύξηση του ipc πιθανό να οφείλεται στο ότι όταν έχουμε περισσότερα νήματα έχουμε και περισσότερες εντολές οι οποίες μπορούν να μπουν στο επεξεργαστή και έτσι να αυξήσουν το ipc. Επίσης ο αριθμός τόσο των εντολών όσο και των κύκλων είναι μεγάλος για ένα τόσο μικρό πρόγραμμα και οφείλεται στο ότι προσμετράται και η δημιουργία των κοντέινερς.



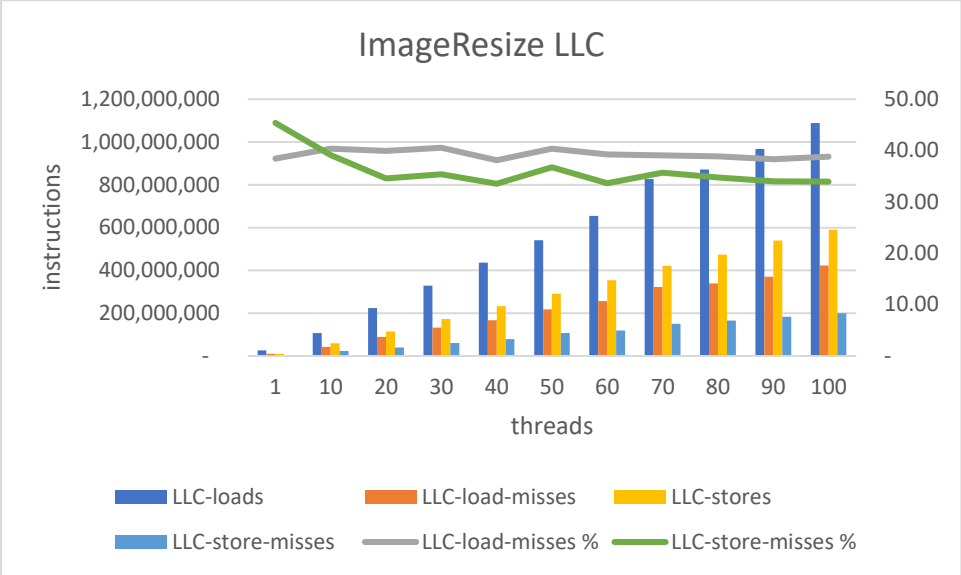
Σχήμα 4.19 Αριθμός ενολών διακλάδωσης, αριθμός αστοχιών εντολών διακλωδωσης, ποσοστό αστοχιών εντολών διακλάδωσης του προγράμματος JavaResize για 1 μέχρι 100 νήματα

Στο Σχήμα 4.19 φαίνεται ο αριθμός των εντολών διακλάδωσης (branch-instuctions), ο αριθμός των αστοχιών για εντολές διακλάδωσης (branch-misses) και το ποσοστό αστοχιών εντολών διακλάδωσης (branch-misses %) για 1 μέχρι 100 νήματα για το πρόγραμμα ImageResize. Ο αριθμός των εντολών διακλάδωσης αυξάνονται από περίπου 1.8 δισεκατομμύρια για 1 νήμα μέχρι περίπου 119 δισεκατομμύρια για 100 νήματα. Ο αριθμός αστοχιών για εντολές διακλάδωσης αυξάνονται από περίπου 40 δισεκατομμύρια για 1 νήμα σε περίπου 2 δισεκατομμύρια για 100 νήματα. Τόσο ο εντολές διακλάδωσης, όσο και οι αστοχίες εντολών διακλάδωσης είναι ανάλογες του αριθμού των νημάτων, πράγμα λογικό αφού το ίδιο πρόγραμμα τρέχει πολλές φορές αυξάνονται οι εντολές που εκτελούνται και επακόλουθα και οι εντολές διακλάδωσης. Το ποσοστό αποτυχίας για τις εντολές διακλάδωσης πέφτει από 2.19% για 1 νήμα σε 1.75% για 20 νήματα και μετά σταθεροποιείται. Αυτή η μείωση πιθανό να σχετίζεται με το ότι περισσότερες ίδιες εντολές εκτελούνται ξανά και ξανά και έτσι ο branch predictor μπορεί να προβλέπει καλύτερα τις επόμενες εντολές. Σύμφωνα με το άρθρο “A Workload Characterization of the SPEC CPU2017 Benchmark Suite” [12] ο μέσος όρος του ποσοστού αστοχίας των εντολών διακλάδωσης είναι λίγο πάνω από 3. Άρα το πρόγραμμα έχει λίγο μικρότερο ποσοστό, πράγμα που είναι λογικό αφού το πρόγραμμα έχει μικρό αριθμό διακλαδώσεων.



Σχήμα 4.20 Αριθμός φορτωμάτων της L1-dcache, αριθμός αστοχίας φορτωμάτων της L1-dcache, αριθμός αποθηκεύσεων L1-dcache, αριθμός αστοχίας φορτωμάτων L1-icache και ποσοστό αστοχίας φορτωμάτων L1-dcache του προγράμματος JavaReize για 1 μέχρι 100 νήματα

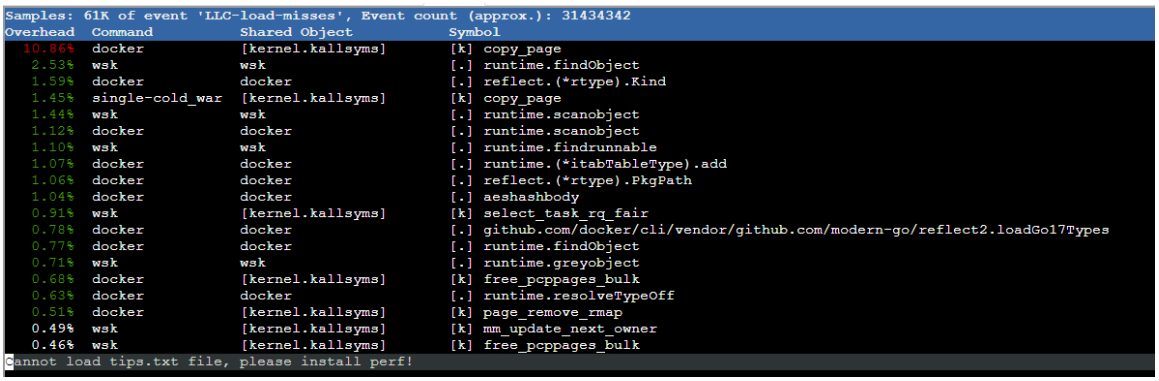
Στο Σχήμα 4.20 έχουμε τον αριθμό των φορτωμάτων (L1-dcache-loads) και αποθηκεύσεων (L1-dcache-stores) της L1-dcache κρυφής μνήμης, τον αριθμό των αστοχιών της L1-dcache των φορτωμάτων της L1-dcache και της L1-icache (L1-dcache-load-misses και L1-icache-load-misses) καθώς και το ποσοστό αστοχιών φορτώματος της L1-dcache (L1-dcache-load-misses %) για το πρόγραμμα JavaResize για 1 μέχρι 100 νήματα. Τα L1-dcache-loads αυξάνονται από περίπου 2.6 δισεκατομμύρια για 1 νήμα σε περίπου 166 δισεκατομμύρια για 100 νήματα. Τα L1-dcache-stores αυξάνονται από περίπου 1.2 δισεκατομμύρια για 1 νήμα σε περίπου 79 δισεκατομμύρια για 100 νήματα. Τα L1-dcache-load-misses αυξάνονται από περίπου 126 εκατομμύρια για 1 νήμα σε περίπου 8.5 δισεκατομμύρια για 100 νήματα. Τα L1-icache-load-misses αυξάνονται από περίπου 180 εκατομμύρια για 1 νήμα σε περίπου 8.3 δισεκατομμύρια για 10 νήματα. Όλα αυτά είναι ανάλογα του αριθμού των νημάτων, πράγμα λογικό αφού εκτελείτε το ίδιο πρόγραμμα περισσότερες φορές. Το ποσοστό L1-dcache-load-misses παραμένει σταθερό κοντά στο 5%. Σύμφωνα με το “A Workload Characterization of the SPEC CPU2017 Benchmark Suite” [12] τα ποσοστά αυτά είναι κοντά στο μέσο όρο του ποσοστού L1-cache-misses.



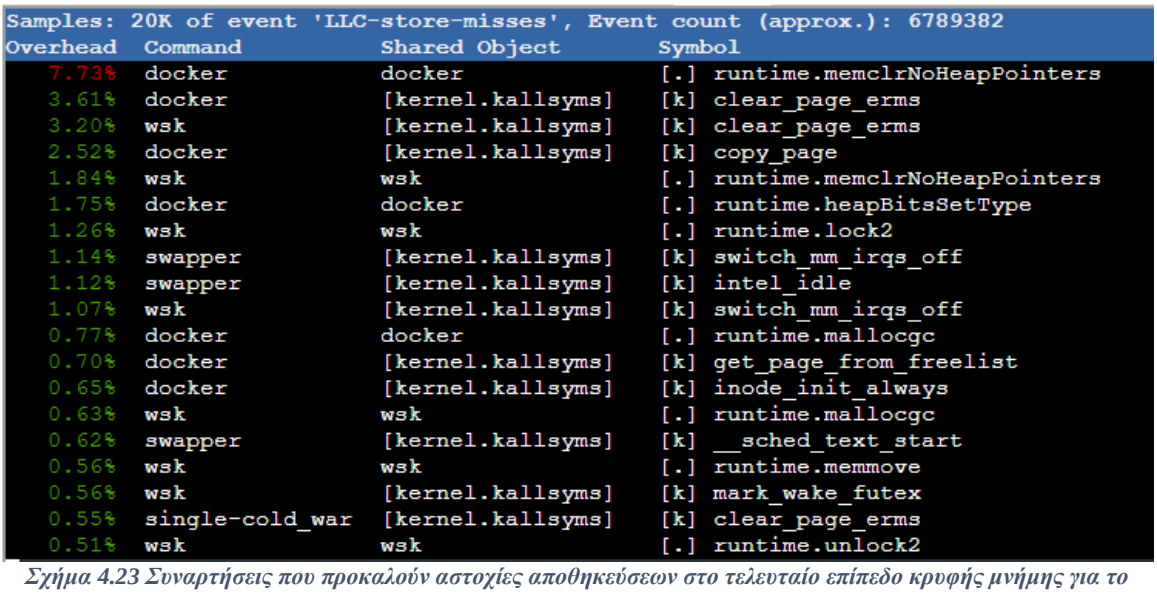
Σχήμα 4.21 Αριθμός φορτωμάτων, αριθμός αστοχίας φορτωμάτων, αριθμός αποθηκεύσεων, αριθμός αστοχίας αποθηκεύσεων, ποσοστό αστοχίας φορτωμάτων και ποσοστό αστοχίας αποθηκεύσεων τελευταίου επιπέδου κρυφής μνήμης του προγράμματος JavaResize για 1 μέχρι 100 νήματα

Στο Σχήμα 4.21 φαίνεται ο αριθμός των φορτωμάτων (LLC-loads), ο αριθμός αστοχιών φορτωμάτων (LLC-load-misses), ο αριθμός αποθηκεύσεων (LLC-stores), ο αριθμός αστοχιών αποθηκεύσεων (LLC-store-misses), το ποσοστό αστοχίας φορτωμάτων (LLC-load-misses %) και το ποσοστό αστοχίας αποθηκεύσεων (LLC-store-misses %) του τελευταίου επιπέδου κρυφής μνήμης για το πρόγραμμα ImageResize για 1 μέχρι 100 νήματα. Ο αριθμός των φορτωμάτων κρυφής μνήμης αυξάνονται από περίπου 26 εκατομμύρια για 1 νήμα σε περίπου 1 δισεκατομμύριο για 100 νήματα. Ο αριθμός των αποθηκεύσεων κρυφής μνήμης αυξάνονται από περίπου 9.5 εκατομμύρια για 1 νήμα σε περίπου 589 εκατομμύρια για 100 νήματα. Ο αριθμός των αστοχιών φορτωμάτων κρυφής μνήμης αυξάνονται από περίπου 10 εκατομμύρια για 1 νήμα σε περίπου 422 εκατομμύρια για 10 νήματα. Ο αριθμός των αστοχιών αποθηκεύσεων κρυφής μνήμης αυξάνονται από περίπου 9.5 εκατομμύρια για 1 νήμα σε περίπου 590 εκατομμύρια για 100 νήματα. Τα τέσσερα αυτά μετρικά είναι ανάλογα του αριθμού των νημάτων, πράγμα λογικό αφού και ο αριθμός των εντολών που εκτελούνται αυξάνεται ανάλογα. Το ποσοστό αστοχίας στο τελευταίο επίπεδο κρυφής μνήμης για φορτώματα μειώνεται από 45.38% για 1 νήμα σε 34.55% για 20 νήματα και μετά παραμένει σταθερό. Το ποσοστό αστοχίας στο τελευταίο επίπεδο κρυφής μνήμη για αποθηκεύσεις κυμαίνεται κοντα στο39%. Σύμφωνα με το άρθρο “A Workload Characterization of the SPEC

CPU2017 Benchmark Suite" [12] ο μέσος όρος για αυτές τις αστοχίες είναι κοντά στο 15%. Τα ποσοστά αστοχίας του προγράμματος είναι αρκετά μεγαλύτερα. Αυτό με έκανε να ψάξω περεταίρω την αιτία αυτού και έτσι χρησιμοποίησα το perf record και το perf report. Όπως φαίνεται στο Σχήμα 4.22 και Σχήμα 4.23 τα υψηλά ποσοστά οφείλονται σε μεγάλο βαθμό στο docker. Ο αριθμός αστοχιών φορτώματος κυμαίνεται από 0.8 μέχρι 1.2 ανά 1000 εντολές και ο αριθμός αστοχιών αποθήκευσης κυμαίνεται από 0.3 μέχρι 0.5 ανά 1000 εντολές. Τα ποσοστά αυτά είναι μικρά και έτσι δεν επηρεάζουν την επίδοση.



Σχήμα 4.22 Συναρτήσεις που προκαλούν αστοχίες φορτωμάτων στο τελευταίο επίπεδο κρυφής μνήμης για το πρόγραμμα JavaResize



Σχήμα 4.23 Συναρτήσεις που προκαλούν αστοχίες αποθηκεύσεων στο τελευταίο επίπεδο κρυφής μνήμης για το πρόγραμμα JavaResize

Κεφάλαιο 5

Συμπεράσματα και μελλοντική εργασία

5.1 Συμπεράσματα

Σε αυτή την εργασία έγινε χρήση performance counters για να μελετηθεί η συμπεριφορά προγραμμάτων του ServerlessBench στην πλατφόρμα OpenWhisk χρησιμοποιώντας το εργαλείο perf. Παρατήρησα ότι είναι δύσκολο στο να πάρεις σωστές μετρήσεις με το εργαλείο αυτό στο OpenWhisk λόγο του ότι δεν μπορείς να προσδέσεις το perf με κάποιο action και πρέπει να το βάλεις να τρέχει στο παρασκήνιο πράγμα που σημαίνει ότι πιθανό να μετράς συμπεριφορά και άλλο συναρτήσεων που τρέχουν εκείνη την ώρα στο σύστημα. Η χρήση perf έχει απαγορευτεί για λόγους ασφαλείας για τη χρήση του σε κοντέινερς docker [13]. Έτσι θα ήταν καλό να αναπτυχθούν άλλα εργαλεία για την λήψη μετρικών που αλλιώς θα παίρνονταν με το perf.

Επίσης ένα πράγμα που παρατηρήθηκε είναι ότι και στα τρία προγράμματα που πήρα μετρικές παρόλο που ήταν σχετικά μικρά προγράμματα χωρίς ιδιαίτερες απαιτήσεις σε πόρους και υπολογιστική ισχύ είχαν μεγάλο ποσοστό αστοχιών στο τελευταίο επίπεδο κρυφής μνήμης τόσο για το φορτώματα, όσο και για τις αποθηκεύσεις. Αυτό οφειλόταν σε μεγάλο βαθμό από το docker.

Ένα άλλο πράγμα που παρατηρήθηκε είναι ότι τα προκαθορισμένα όρια για της πλατφόρμας OpenWhisk είναι χαμηλά. Για να μπορέσεις να τρέξεις προγράμματα με

πολλά νήματα ή με μεγάλη ανάγκη πόρων ή οι να τρέξεις πολλά actions πρέπει να αυξήσεις κάποια όρια μέσο μεταβλητών περιβάλλοντος.

Τέλος η τα προγράμματα του ServerlessBench ήταν αρκετά μικρά και δεν χρειάζονταν μεγάλη χρησιμοποίηση πόρων και υπολογιστικής ισχύς. Έτσι δεν επιτεύχθηκε να πιέσουμε την πλατφόρμα OpenWhisk ώστε να μπορούμε να βγάλουμε περισσότερες παρατηρήσεις για την πλατφόρμα. Στα πλαίσια της διπλωματικής προσπαθήσαμε να εισάξουμε μεγαλύτερο μέγεθος εικόνας για να πιέσουμε το σύστημα, αλλά αυτό δεν επιτευχθεί λόγο του ότι το μέγεθος του αρχείου που μπορούσε να εισαχθεί στο OpenWhisk σαν παράμετρο ήταν 1MB.

5.2 Μελλοντική δουλειά

Αυτό που θα μπορούσε να γίνει είναι η ανάπτυξη προγραμμάτων τα οποία να έχουν μεγάλες ανάγκες σε πόρους και υπολογιστική ισχύ. Αυτό θα έχει ως αποτέλεσμα την περεταίρω μελέτη της πλατφόρμα OpenWhisk. Επίσης θα μπορούσε να γίνει χρήση άλλων εργαλείων για να εξαχθούν μετρικά τα οποία δεν επηρεάζονται από άλλες διεργασίες που πιθανόν να τρέχουν. Ακόμα θα μπορούσε να γίνει ανάπτυξη και άλλων τέτοιων εργαλείων που να παίρνουν μετρικά. Τέλος θα μπορούσαν να μελετηθούν και άλλες πλατφόρμες serverless και να γίνει σύγκριση μεταξύ τους.

Βιβλιογραφία

[1] T. a. L. Q. a. D. D. a. X. Y. a. Z. B. a. L. Z. a. Y. P. a. Q. C. a. C. H. Yu, Characterizing Serverless Platforms with ServerlessBench, Association for Computing Machinery, 2020.

[2] «AWS Lambda,» [Ηλεκτρονικό]. Available: https://aws.amazon.com/lambda/.

[3] «Apache OpenWhisk,» [Ηλεκτρονικό]. Available: https://openwhisk.apache.org/.

[4] «Fn Project,» [Ηλεκτρονικό]. Available: https://fnproject.io/.

[5] «Cloud Functions,» [Ηλεκτρονικό]. Available: https://cloud.google.com/functions.

[6] «Azure Functions,» [Ηλεκτρονικό]. Available: https://docs.microsoft.com/en-us/azure/azure-functions/.

[7] D. U. a. P. P. a. M. K. a. E. B. a. B. Grot, «Benchmarking, Analysis, and Optimization of Serverless Function Snapshots,» σε *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'21)*, ACM, 2021.

[8] «CloudLab,» [Ηλεκτρονικό]. Available: https://www.cloudlab.us/.

[9] A. K. P. M. Hossein Shafiei, «Serverless Computing: A Survey of Opportunities,» ACM Comput. Surv., 2021.

[10] V. I. V. M. A. S. Paul Castro, «The server is dead, long live the server: Riseof Serverless Computing, Overview of Current State and Future Trends in Research and Industry».

[11] «OpenWhisk,» [Ηλεκτρονικό]. Available: https://github.com/apache/openwhisk/.

[12] A. L. a. T. Adegbija, «A Workload Characterization of the SPEC CPU2017 Benchmark Suite».

[13] «running `perf` in docker & kubernetes,» [Ηλεκτρονικό]. Available: https://medium.com/@geekidea_81313/running-perf-in-docker-kubernetes-7eb878afcd42.

Παράρτημα Α

OpenWhiskSetup.sh:

```
sudo apt-get update
sudo apt-get install \
   apt-transport-https \
   ca-certificates \
   curl \
   gnupg \
   lsb-release -y
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor
-o /usr/share/keyrings/docker-archive-keyring.gpg
echo \
  "deb [arch=amd64 signed-by=/usr/share/keyrings/docker-archive-keyring.gpg]
https://download.docker.com/linux/ubuntu \
  $(lsb_release -cs) stable" | sudo tee /etc/apt/sources.list.d/docker.list >
/dev/null
sudo apt-get update
sudo apt-get install docker-ce docker-ce-cli containerd.io -y
sudo groupadd docker
sudo usermod -aG docker $USER
sudo apt install openjdk-8-jdk -y
sudo apt-get install jq -y
sudo apt install nodejs npm -y
sudo apt install xdg-utils -y
git clone https://github.com/apache/openwhisk.git
wget                                        https://github.com/apache/openwhisk-
cli/releases/download/1.2.0/OpenWhisk_CLI-1.2.0-linux-amd64.tgz
tar zxvf OpenWhisk_CLI-1.2.0-linux-amd64.tgz
sudo cp wsk /usr/local/bin
git clone https://github.com/SJTU-IPADS/ServerlessBench.git
cd openwhisk
export LIMITS_ACTIONS_INVOKES_PERMINUTE=60000
export LIMITS_ACTIONS_INVOKES_CONCURRENT=5000
export LIMITS_TRIGGERS_FIRES_PERMINUTE=60000
export LIMITS_ACTIONS_INVOKES_CONCURRENTINSYSTEM=5000
```

```
export LIMITS_ACTIONS_SEQUENCE_MAXLENGTH=200
sudo ./gradlew core:standalone:bootRun

wsk property set --apihost 'http://172.17.0.1:3233' --auth '23bc46b1-71f6-4ed5-8c54-816aa4f8c502:123zO3xZCLrMN6v2BKK1dXYFpXlPkccOFqm12CdAsMgRU4VrNZ9lyGVCGuMDGIwP'
```