Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών
και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών

Χρήση διαφορικών στιγμιοτύπων για την βελτιστοποίηση εκκίνησης Serverless workloads σε ARM επεξεργαστές

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Αλέξανδρος Ασημάκης, Χ. Σαμπάνης

Επιβλέπων : Νεκτάριος Κοζύρης
Καθηγητής ΕΜΠ

Αθήνα, Ιούνιος 2024

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών
και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών

Χρήση διαφορικών στιγμιοτύπων για την βελτιστοποίηση εκκίνησης Serverless workloads σε ARM επεξεργαστές

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Αλέξανδρος Ασημάκης, Χ. Σαμπάνης

Επιβλέπων : Νεκτάριος Κοζύρης
Καθηγητής ΕΜΠ

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 20η Ιουνίου 2024.

....	....	....
Νεκτάριος Κοζύρης	Διονύσιος Πνευματικάτος	Γεώργιος Γκούμας
Καθηγητής ΕΜΠ	Καθηγητής ΕΜΠ	Αναπληρωτής Καθηγητής ΕΜΠ

Αθήνα, Ιούνιος 2024

.................................…
Αλέξανδρος Ασημάκης, Χ. Σαμπάνης

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

Copyright © Αλέξανδρος Ασημάκης, Σαμπάνης, 2024
Με επιφύλαξη παντός δικαιώματος. All rights reserved.

Απαγορεύεται η αντιγραφή, αποθήκευση και διανομή της παρούσας εργασίας, εξ ολοκλήρου ή τμήματος αυτής, για εμπορικό σκοπό. Επιτρέπεται η ανατύπωση, αποθήκευση και διανομή για σκοπό μη κερδοσκοπικό, εκπαιδευτικής ή ερευνητικής φύσης, υπό την προϋπόθεση να αναφέρεται η πηγή προέλευσης και να διατηρείται το παρόν μήνυμα. Ερωτήματα που αφορούν τη χρήση της εργασίας για κερδοσκοπικό σκοπό πρέπει να απευθύνονται προς τον συγγραφέα.

Οι απόψεις και τα συμπεράσματα που περιέχονται σε αυτό το έγγραφο εκφράζουν τον συγγραφέα και δεν πρέπει να ερμηνευθεί ότι αντιπροσωπεύουν τις επίσημες θέσεις του Εθνικού Μετσόβιου Πολυτεχνείου.

Περίληψη

Η αρχιτεκτονική χωρίς διακομιστή (Serverless) είναι ένα σύγχρονο cloud-computing paradigm. Σε συνεργασία με τις Function-as-a-Service εφαρμογές υπόσχεται υψηλή αξιοποίηση των πόρων που παρέχουν τα κέντρα δεδομένων μέσω της κατανομής και διάθεσης πόρων κατά απαίτηση του εκάστοτε αιτήματος κάποιας εφαρμογής. Μια ενδιαφέρουσα εφαρμογή της Serverless αυτής αρχιτεκτονικής, είναι η χρησιμοποίηση Internet of Things συσκευών, οι οποίες ολοένα και αυξάνονται.

Ένας μεγάλος περιορισμός της Serverless αρχιτεκτονικής είναι η καθυστέρηση που προκύπτει από την "ενεργοποίηση" της υποκείμενης υποδομής (microVM, container, unikernel) της εκάστοτε Function-as-a-Service εφαρμογής ώστε να εξυπηρετήσει κάποιο αίτημα προς αυτήν. Η "κρύα" εκκίνηση όπως αποκαλείται, αναφέρεται σε περιπτώσεις που η εφαρμογή μας στο Serverless περιβάλλον δεν έχει κάποια πρόσφατη κλήση, οπότε και η εικονική μηχανή μέσα στην οποία εκτελείται δεν βρίσκεται σε κατάσταση αδράνειας. Όταν πραγματοποιηθεί κάποια κλήση προς την εφαρμογή λοιπόν, πρέπει να στηθεί εξ'ολοκλήρου η μικρο-εικονική μηχανή, γεγονός που προσδίδει καθυστέρηση στην εξυπηρέτηση κάποιου αιτήματος.

Μια ενδεχόμενη λύση που έχει μελετηθεί σε μεγάλο βαθμό από την επιστημονική κοινότητα είναι η χρήση στιγμιότυπων, μέσω των οποίων μπορούμε να αποθηκεύσουμε την τρέχουσα κατάσταση της υποκείμενης υποδομής και να την επαναφέρουμε (όταν χρειαστεί) με μικρό χρονικό κόστος. Προηγούμενες δουλειές στο τομέα της έρευνας έχουν δείξει ότι μπορούμε να καταφέρουμε μείωση του χρόνου της "κρύας" εκκίνησης μέσω διαφορετικών τεχνικών υλοποίησης στιγμιότυπων. Σε αυτήν την εργασία μελετήσαμε ενδελεχώς το SnapFaas εργαλείο που χρησιμοποιεί καινούργιες τεχνικές για την υλοποίηση των στιγμιότυπων και το οποίο υπόσχεται γρήγορες, από άκρη σε άκρη, εξυπηρετήσεις αιτημάτων.

Με βάση την αυξανόμενη τάση των IoT συσκευών οι οποίες χρησιμοποιούν κατά κύριο λόγο ARM επεξεργαστές, εξετάσαμε την δυνατότητα μεταφοράς του SnapFaas σε υπολογιστικά συστήματα βασισμένα σε ARM. Θα αναλύσουμε τις αλλαγές στις οποίες προβήκαμε για να το καταφέρουμε αυτό παραθέτοντας και κάποια κομμάτια κώδικα απο την υλοποίηση. Επιπλέον, θα μελετήσουμε τον τρόπο που λαμβάνονται τα στιγμιότυπα απο το SnapFaas "αιχμαλωτίζοντας" όσο το δυνατόν πιο αποδοτικά την μνήμη και τις διάφορες συσκευές που είναι συνδεδεμένες στο microVM. Αντίστοιχα, θα δούμε το πως τα στιγμιότυπα αυτά χρησιμοποιούνται για την επαναφορά της μικρο-εικονικής μηχανής, τόσο στο κομμάτι της μνήμης όσο και στο κομμάτι των συνδεδεμένων συσκευών.

Στο τελευταίο κομμάτι της διπλωματικής, παρουσιάζουμε έναν client που δημιουργήσαμε για το Firecracker (γνωστό εργαλείο εκκίνησης microVMs), με σκοπό να προσομοιώσουμε την λειτουργία του SnapFaas. Με αυτόν τον τρόπο αποκτήσαμε ένα μέτρο σύγκρισης για το SnapFaas εργαλείο σχετικά με τους χρόνους που μετρήσαμε στα διαφορετικά σκέλη της End-to-End εξυπηρέτησης των αιτημάτων για τις συναρτήσεις που εκτελέσαμε. Το SnapFaas λοιπόν αποδείχτηκε γρηγορότερο για τις "κρύες" αλλά και για τις "χλιαρές" (όταν υπάρχει αποτύπωμα του microVM στην μνήμη) εκκινήσεις των microVMs.

Λέξεις Κλειδιά

Μικρο-εικονική μηχανή, Αρχιτεκτονική χωρίς διακομιστή, ARM αρχιτεκτονική, IoT, Στιγμιότυπα, "Κρύα" εκκίνηση, Συνάρτηση-σαν-Υπηρεσία

Abstract

Serverless architecture is a modern cloud-computing paradigm. In collaboration with Function-as-a-Service applications, it promises high utilization of the resources provided by data centers through the allocation of resources on demand. An interesting application of the Serverless architecture is the use of Internet of Things devices, which are constantly increasing.

A major limitation of the Serverless architecture is the delay resulting from the activation of the underlying infrastructure (microVM, container, unikernel) of each Function-as-a-Service application, in order to serve a request. The cold-start process, as it is called, refers to cases where our application in the Serverless environment has not been invoked recently and thus, the virtual machine in which it is running is not in an idle state, nor activated. So when an invocation to that function is made, the microVM must be set up entirely from scratch, which adds a delay to the service of the request.

A potential solution that has been studied thoroughly by the scientific community is the use of snapshots, through which we can store the current state of the underlying infrastructure and restore it (when needed) with negligible overhead. Previous research work has shown that we can achieve a reduction in cold-start latency through different snapshotting techniques. In this paper we studied the SnapFaas protocol which uses new techniques to implement snapshots and which promises fast End-to-End request servings.

Based on the growing trend of IoT devices that primarily use ARM processors, we inspected the porting of SnapFaas system to ARM-based computing systems. We will analyze the changes needed to achieve this and also quote some code snippets from the implementation. Additionally, we will study how snapshots are taken by SnapFaas, capturing as efficiently as possible the memory and the various devices attached to the microVM. Accordingly, we will examine how these snapshots are used to restore the microVM, both in the memory part, as well as the part of the connected devices.

In the last part of the thesis, a client for the Firecracker tool (a well-known microVMs startup tool) is presented, in order to simulate the operation of SnapFaas. Through this new system we were able to obtain a comparison tool for SnapFaas, throughout all the experiments that were showcased. Through them we proved that SnapFaas is faster for cold-start, as well as for lukewarm-start (when there is an imprint of the microVM inside the page cache of the memory) regarding the End-to-End latency.

KeyWords

MicroVM, Serverless architecture, ARM-based architecture, Internet of Things, Snapshots, Cold-start, Function-as-a-Service

Ευχαριστίες

Θα ήθελα να ευχαριστήσω τον επιβλέποντα Κο Κοζύρη για την επίβλεψη της παρούσας διπλωματικής εργασίας και για την δυνατότητα που μου έδωσε να ασχοληθώ με αυτό το θέμα. Επίσης θα ήθελα να ευχαριστήσω ιδιαίτερα τους Κωσταντίνο Νίκα και Χρήστο Κατσακιώρη για την διαρκή καθοδήγηση και βοήθεια που μου παρείχαν όποτε την χρειάστηκα.

Σε μια πιο προσωπική νότα, θα ήθελα να εκφράσω την ευγνωμοσύνη μου στην οικογένεια μου, τον Χάρη, την Χρυσάνθη, την Σοφιάννα και τον Φίλιππο, για την στήριξη, τα εφόδια και τον τρόπο που μεγάλωσα. Τέλος, να ευχαριστήσω θερμά όλους τους φίλους μου και ιδιαίτερα τον συγκάτοικο μου Ευστάθιο Ανδριανόπουλο και την κοπέλα μου Ευφροσύνη Αθανάσουλα, για την αμέριστη υποστήριξη τους, καθώς δεν θα τα κατάφερνα χωρίς αυτούς.

Περιεχόμενα

Περίληψη.....5
Abstract.....7
Ευχαριστίες.....8
Περιεχόμενα.....9
Κατάλογος Εικόνων.....11
Κατάλογος Πινάκων.....13
1 Εισαγωγή.....14
1.1 Serverless.....14
1.2 Συνάρτηση σαν υπηρεσία (FaaS - Function as a Service).....14
1.3 Μεταφορά Serverless σε IoT devices.....15
1.4 Αρχιτεκτονική των IoT συσκευών:.....16
1.5 Πρόβλημα "κρύας" εκκίνησης στο Serverless:.....16
1.6 Σύγκριση των αρχιτεκτονικών ARM & x86 στην εκκίνηση μέσω στιγμιότυπου.....17
2 Θεωρητικό Υπόβαθρο - Εργαλεία.....20
2.1 Ορισμοί.....20
2.2 Μηχανισμοί Στιγμιοτύπων.....22
2.3 SnapFaas.....25
3 Τεχνική Περιγραφή και Υλοποίηση του SnapFass.....28
3.1 Τεχνικές Λεπτομέρειες.....28
3.1.1 Virtual Sockets - Virtio.....30
3.1.2 Λήψη στιγμιοτύπων.....31
3.2 Τεχνική υλοποίηση.....34
3.2.1 Αποθήκευση μνήμης.....36
3.2.2 Επαναφορά μνήμης.....38
3.2.3 Αποθήκευση κατάστασης Virtio συσκευών και Εικονικού Επεξεργαστή.....41
3.2.4 Επαναφορά κατάστασης virtio συσκευών και Εικονικού Επεξεργαστή.....41
3.2.5 General Interrupt Controller.....42
3.2.5.1 Αποθήκευση και επαναφορά της κατάστασης του GIC.....43
3.2.6 Flattened Device Tree – FDT.....44
3.3 Μεταφορα σε ARM αρχιτεκτονική.....45
3.3.1 Ζητήματα που προέκυψαν.....46
4 Μεθοδολογία.....49
4.1 Στιγμιότυπα στο Firecracker.....49
4.2 Τεχνική υλοποίηση Firecracker client.....50
4.3 Χαρακτηριστικά Περιβάλλοντος Εκτέλεσης.....57
4.4 Benchmarking functions.....58
4.5 Ανάλυση Φάσεων Εκτέλεσης.....60
4.5.1 1ο Μέρος: Αποκατάσταση Μνήμης (Memory restoration).....60
4.5.2 2ο Μέρος: Εκκίνηση μικρο-εικονικής μηχανής (Booting).....62
4.5.3 3ο Μέρος: Ετοιμότητα μικρο-εικονικής μηχανής (VM Ready).....63
4.5.4 4ο Μέρος: Ολοκλήρωση αιτήματος (Request Returned).....64
4.5.5 5ο Μέρος: Διάρκεια εκτέλεσης (Function).....64
5 Πειραματικό Σκέλος.....67
5.1 Cold-Start.....67
5.1.1 Cold-Start Memory restoration phase 128MB/256MB/512MB.....67
5.1.2 Cold-Start Boot phase 128MB/256MB/512MB.....68
5.1.3 Cold-Start VM Ready phase 128MB/256MB/512MB.....69
5.1.4 Cold-Start Request Returned phase 128MB/256MB/512MB.....70
5.1.5 Cold-Start Duration phase 128MB/256MB/512 MB.....70
5.1.6 Cold-Start End-to-End 128MB/256 MB/512 MB.....71

5.2 Lukewarm-Start....72
5.2.1 Lukewarm-Start Memory restoration phase 128MB/256MB/512MB....72
5.2.2 Lukewarm-Start Boot phase 128MB/256MB/512MB....73
5.2.3 Lukewarm-Start VM Ready phase 128MB/256MB/512MB....74
5.2.4 Lukewarm-Start Request Returned phase 128MB/256MB/512MB....74
5.2.5 Lukewarm-Start Duration phase 128MB/256MB/512MB....75
5.2.6 Lukewarm-Start End-to-End 128MB/256MB/512MB....76
6 Συμπεράσματα....77
6.1 Συζήτηση - Σύνοψη....77
6.2 Περιορισμοί και Επόμενα βήματα....78
Βιβλιογραφία....80

Κατάλογος Εικόνων

Σχήμα 1: Snapshot speedups compared to normal boot on x86 & ARM architectures.........19
Σχήμα 2: Σύκγριση cold/warm-start...21
Σχήμα 3: Η εκκίνηση χωρίς αρχικοποίηση του Catalyzer [9]..22
Σχήμα 4: Συνοπτική παρουσίαση του Catalyzer μηχανισμού [9]..23
Σχήμα 5: Η φάση εύρεσης του *working set* της συνάρτησης στο REAP [29]......................23
Σχήμα 6: Η ανάκτηση των σελίδων και η εγκατάσταση τους στην μνήμη της εικονικής μηχανής στο εργαλείο REAP [29]..24
Σχήμα 7: Σύγκριση του SnapFaaS με τα υπάρχοντα στιγμιότυπα για "κρύες" εκκινήσεις. Το REAP καταγράφει την κατάσταση εκτέλεσης ως ένα ολοκληρωμένο σύνολο και φορτώνει μέσω σελίδωσης-απαίτησης τις σελίδες του *working set*. Το SEUSS επαναφέρει την κοινή κατάσταση που είναι αποθηκευμένη στην cache κατά απαίτηση και στη συνέχεια εισάγει τις συναρτήσεις από την πηγή. Το SnapFaaS αποθηκεύει στην cache την κοινή κατάσταση και αποθηκεύει τις καταστάσεις των συναρτήσεων στο δίσκο και φορτώνει άμεσα (eagerly) μόνο το σύνολο εργασίας. [30]..26
Σχήμα 8: Επισκόπηση των επιμέρους κομματιών του Firecrakcer......................................28
Σχήμα 9: Επισκόπηση των επιμέρους κομματιών του SnapFaas [31].................................29
Σχήμα 10: Παράδειγμα εκτέλεσης της singlevm εντολής...29
Σχήμα 11: Διαδικασία λήψης base snaphot...32
Σχήμα 12: Η tarball δομή που περιέχει το workload της εφαρμογής..................................33
Σχήμα 13: Τα πεδία του Vmm struct...35
Σχήμα 14: Το περιεχόμενο ενός pagemap αρχείου μιας τυχαίας διεργασίας........................37
Σχήμα 15: Ο τρόπος που στέλνονται τα αιτήματα και η πυροδότηση της εύρεσης του working set, απο το singlevm εκτελέσιμο του SnapFaas...38
Σχήμα 16: Η συνάρτηση copy_load_memory() του VMM, υπεύθυνη για το eager restoration είτε της μνήμης του στιγμιότυπου συνάρτησης είτε του working set.................40
Σχήμα 17: Η συνάρτηση dump_vcpu_state()..41
Σχήμα 18: Το πεδίο γνωρισμάτων (attribute field) του register (οποιουδήποτε τύπου) του irqchip [23]..43
Σχήμα 19: Η συνάρτηση create_fdt()..45
Σχήμα 20: Η διάταξη μνήμης σε ένα ARM υπολογιστικό σύστημα [2]..............................48
Σχήμα 21: Ο Python κώδικας για την δημιουργία και την αναμονή του virtual socket........52
Σχήμα 22: Connection establishment between Unix socket (client) and Virtual Socket (guest)..55
Σχήμα 23: Ο Python κώδικας στην πλευρά του host για την επικοινωνία με το VM...........57
Σχήμα 24: Τα βήματα του memory restoration & του booting phase για το SnapFaas.........61
Σχήμα 25: Τα βήματα του memory restoration & του booting phase για το Firecracker......62
Σχήμα 26: Τα βήματα του VM Ready phase για το SnapFaas...63
Σχήμα 27: Τα βήματα του VM Ready phase για το Firecracker..64
Σχήμα 28: Τα βήματα του Request Returned phase για το SnapFaas..................................65
Σχήμα 29: Τα βήματα του Request Returned phase για το Firecracker...............................65
Σχήμα 30: Memory restoration Latency, *SnapFaas slowdowns* for both language and function snapshots restored lazily and working set eagerly...67
Σχήμα 31: Boot Latency, *SnapFaas slowdowns* for both language and function snapshots restored lazily and working set eagerly...68
Σχήμα 32: VMReady Latency, SnapFaas *slowdowns* for both language and function snapshots restored lazily and working set eagerly...69
Σχήμα 33: Request Returned Latency, SnapFaas *slowdowns* for both language and function snapshots restored lazily and working set eagerly...70
Σχήμα 34: Duration Latency, SnapFaas *slowdowns* for both language and function snapshots restored lazily and working set eagerly...70

Σχήμα 35: End-to-End Latency, SnapFaas *slowdowns* for both language and function snapshots restored lazily and working set eagerly......71
Σχήμα 36: Memory Restoration Latency, SnapFaas slowdowns for both language and function snapshots restored lazily and working set eagerly......72
Σχήμα 37: Boot Latency, SnapFaas slowdowns for both language and function snapshots restored lazily and working set eagerly......73
Σχήμα 38: VMReady Latency, SnapFaas *slowdowns* for both language and function snapshots restored lazily and working set eagerly......74
Σχήμα 39: Request Returned Latency, SnapFaas slowdowns for both language and function snapshots restored lazily and working set eagerly......74
Σχήμα 40: Duration Latency, SnapFaas slowdowns for both language and function snapshots restored lazily and working set eagerly......75
Σχήμα 41: End-to-End Latency, SnapFaas *slowdowns* for both language and function snapshots restored lazily and working set eagerly......76
Σχήμα 42: Firecracker vs SnapFaas optimization on ARM......78

Κατάλογος Πινάκων

Πίνακας 1: Πίνακας με τους χρόνους εκκίνησης σε x86 και *ARM* συστήματα......................18
Πίνακας 2: Περιγραφή και χαρακτηριστικά των συναρτήσεων που εφαρμόστηκαν............60

1 Εισαγωγή

1.1 Serverless

Το Serverless είναι ένα μοντέλο εκτέλεσης υπολογισμών στο cloud, όπου ο πάροχος cloud διαχειρίζεται δυναμικά την κατανομή πόρων (resources) στους διακομιστές. Ουσιαστικά, οι προγραμματιστές μπορούν να επικεντρωθούν αποκλειστικά στην εγγραφή κώδικα χωρίς να ανησυχούν για την υποκείμενη υποδομή. Θεωρείται το επόμενο στάδιο στην υπολογιστική νέφους. Διαβάζοντας τον όρο Serverless, μπορεί κανείς να θεωρήσει ότι μπαίνουμε σε μια νέα εποχή, όπου δεν χρειαζόμαστε servers για να εκτελέσουμε εφαρμογές. Το συγκεκριμένο μοντέλο εκτέλεσης στην υπολογιστική νέφους δεν εξαλείφει την ανάγκη για χρήση φυσικών διακομιστών, αλλά παρέχει μια ολοκληρωμένη υποδομή για την εκτέλεση εφαρμογών χωρίς την πολύπλοκη διαχείριση των servers.

Ένα από τα πιο σημαντικά πλεονεκτήματα του Serverless είναι η κλιμακωσιμότητά του. Αυτό σημαίνει ότι οι υπολογιστικοί πόροι (resources) παρέχονται ή εξαλείφονται με βάση την ζήτηση. Αντίστοιχα, οι εφαρμογές που χρησιμοποιούν τους συγκεκριμένους πόρους κλιμακώνονται αυτόματα αναλόγως τη ζήτηση, με αποτέλεσμα να μην μας απασχολούν οι απότομες αυξήσεις στην επισκεψιμότητα της εφαρμογής μας. Αυτή η κλιμακωσιμότητα οδηγεί σε εξοικονόμηση κόστους, αφού το hosting εφαρμογών σε cloud servers κοστολογείται με βάση τους υπολογιστικούς πόρους που πραγματικά απαιτούνται (pay-per-usage model).

Επιπλέον, η serverless αρχιτεκτονική επιταχύνει την ανάπτυξη εφαρμογών, αφού ο προγραμματιστής μπορεί να επικεντρωθεί αποκλειστικά στην ανάπτυξη του κώδικα για την εφαρμογή του αντί να διαχειρίζεται διακομιστές. Είναι ιδιαιτέρως κατάλληλη για αρχιτεκτονικές εφαρμογών που "ενεργοποιούνται" από συγκεκριμένα γεγονότα όπως αιτήσεις HTTP, αλλαγές στη βάση δεδομένων ή μεταφορτώσεις αρχείων (file uploads). Η αρχιτεκτονική αυτή αποκαλείται "Συνάρτηση σαν υπηρεσία" (Function as a Service)

1.2 Συνάρτηση σαν υπηρεσία (FaaS - Function as a Service)

Είναι επίσης μια cloud computing υπηρεσία που επιτρέπει στους πελάτες να εκτελούν κώδικα ως απάντηση σε συμβάντα (events), χωρίς την ανάγκη διαχείρισης της πολύπλοκης υποδομής. Τέτοιου είδους συναρτήσεις μπορούν να εξυπηρετηθούν μέσω της Serverless αρχιτεκτονικής.

Το hosting μιας εφαρμογής συνήθως απαιτεί την πρόβλεψη και διαχείριση ενός εικονικού ή φυσικού διακομιστή και τη διαχείριση ενός λειτουργικού συστήματος και των διαδικασιών φιλοξενίας ενός διακομιστή ιστού. Με την Συνάρτηση ως Υπηρεσία (FaaS), το υλικό, το λειτουργικό σύστημα της εικονικής μηχανής και η διαχείριση του λογισμικού του διακομιστή χειρίζονται αυτόματα από τον παροχέα υπηρεσιών νέφους. Επιπλέον, οι εικονικοί ή φυσικοί διακομιστές δεσμεύουν συγκεκριμένους πόρους επεξεργαστών, μνήμης, δικτύου και δίσκου για κάθε συνάρτηση καθώς εκτελείται ή παραμένει αδρανής (περιμένοντας κάποιο συμβάν για να ενεργοποιηθεί). Όπως αναφέραμε και προηγουμένως αυτό είναι ιδιαίτερα σημαντικό για τον χρήστη της υπηρεσίας, καθώς η χρέωση των εικονικών μηχανών/containers στον Severless κόσμο συμβαίνει με βάση την χρήση της επεξεργαστικής ισχύς - μνήμης.

Οι AWS Lambda [32] και Azure Functions [33] είναι μερικές από τις πιο δημοφιλείς πλατφόρμες Serverless computing που προσφέρονται από τους μεγάλους παρόχους cloud για υποστήριξη Function as a Service εφαρμογών. Οι πλατφόρμες αυτές υποστηρίζουν διάφορες γλώσσες προγραμματισμού, επιτρέποντας στον χρήστη να επιλέξει την γλώσσα με την οποία έχει μεγαλύτερη εξοικείωση.

1.3 Μεταφορά Serverless σε IoT devices

Ο όρος IoT (Internet of Things) αναφέρεται στο συλλογικό δίκτυο συνδεδεμένων embedded συσκευών και στην τεχνολογία μέσω της οποίας επιτυγχάνεται η επικοινωνία μεταξύ των συσκευών και του cloud, καθώς και των συσκευών μεταξύ τους. Οι IoT συσκευές είναι φυσικά αντικείμενα ενσωματωμένα με σένσορες και λογισμικό που τους επιτρέπει να συνδεθούν στο διαδίκτυο και να ανταλλάξουν δεδομενα με άλλες συσκευές ή συστήματα.

Για την επεξεργασία δεδομένων, οι IoT συσκευές έχουν περιορισμένη υπολογιστική ισχύ και μνήμη σε σύγκριση με τα πιο συνηθισμένα συστήματα επεξεργασίας. Αυτό τις καθιστά ιδανικές για hosting FaaS εφαρμογών, οι οποίες στις περισσότερες των περιπτώσεων έχουν μικρό χρόνο εκτέλεσης και αρα δεν απαιτούν μεγάλη υπολογιστική ισχύ.

Τα τελευταία χρόνια παρατηρείται μια αυξανόμενη τάση μεταφοράς της serverless αρχιτεκτονικής στις IoT συσκευές. Η τάση αυτή πηγάζει από το γεγονός ότι συγκεκριμένα χαρακτηριστικά των συσκευών αυτών μπορούν να επωφεληθούν από τις δυνατότητες της Serverless αρχιτεκτονικής και τα οποία θα αναλύσουμε παρακάτω.

Τρέχοντας Function as a Service εφαρμογές σε IoT συσκευές μπορούμε να επωφεληθούμε αναλόγως του use case και τις απαιτήσεις:

Όπως αναφέραμε και προηγουμένως, οι συσκευές αυτές έχουν περιορισμένους υπολογιστικούς πόρους, όπως επεξεργαστική ισχύ και μνήμη. Η FaaS αρχιτεκτονική επιτρέπει αποτελεσματική αξιοποίηση αυτών των πόρων εφόσον εκτελείται κώδικας μόνο όταν χρειαστεί, ως απάντηση συγκεκριμένων γεγονότων ή μηχανισμών ενεργοποίησης (triggering mechanisms). Αυτό το μοντέλο εκτέλεσης ελαχιστοποιεί την σπατάλη πόρων και διασφαλίζει την βέλτιστη απόδοση των IoT συσκευών.

Η FaaS υποδομή διαχειρίζεται αυτόματα την κλιμακωσιμότητα των εφαρμογών. Η αυτοματοποίηση αυτή είναι ιδιαίτερα χρήσιμη σε IoT περιβάλλοντα, όπου το πλήθος των συσκευών και των δεδομένων που παράγονται μπορούν να μεταβάλλονται, καθώς δεν χρειάζεται χειροκίνητη παρέμβαση, εξασφαλίζοντας σταθερή απόδοση και ανταπόκριση.

Επιπλέον, οι IoT εφαρμογές είναι εξ'ορισμού event-driven, με τις συσκευές να παράγουν δεδομένα και να πυροδοτούν εκτελέσεις βασισμένες σε γεγονότα. Το FaaS συντονίζεται με αυτήν την event-driven αρχιτεκτονική, επιτρέποντας σε εφαρμογές να πάρουν το έναυσμα για να ξεκινήσουν την εκτέλεση τους απο διάφορα γεγονότα, όπως διάβασμα αισθητήρων, ενεργοποίηση συσκευών και αλληλεπιδράσεις του χρήστη.

Όπως αναφέραμε και προηγουμένως, η Serverless αρχιτεκτονική ακολουθεί μοντέλο πληρωμής-ανα-χρήση (pay-per-usage model), όπου ο χρήστης πληρώνει αποκλειστικά για τους υπολογιστικούς πόρους που καταναλώνουν οι συναρτήσεις που "τρέχουν" εκ μέρους του. Το μοντέλο πληρωμης αυτό είναι ιδανικό για ανάπτυξη IoT εφαρμογών όπου οι χρησιμοποιούμενοι πόροι είναι δύσκολο να προβλεφθούν. Με την αξιοποίηση της FaaS

αρχιτεκτονικής, οι IoT εφαρμογές βελτιστοποιουν την κοστολόγηση με την οποία ο χρήστης βαραίνεται.

1.4 Αρχιτεκτονική των IoT συσκευών:

Στην πλειοψηφία των περιπτώσεων, οι IoT συσκευές χρησιμοποιούν ARM επεξεργαστές, καθώς είναι ενεργειακά αποδοτικοί, χαμηλού κόστους και κατάλληλοι για ενσωματωμένα συστήματα. Η ARM αρχιτεκτονική μπορεί να είναι είτε 32, είτε 64 δυφίων (bits). Στο πειραματικό σκέλος της διπλωματικής χρησιμοποιήσαμε υπολογιστικό σύστημα ARM αρχιτεκτονικής 64 bit, η οποία αποκαλείται aarch64. Είναι μια ευρέως διαδεδομένη επιλογή είτε για χαμηλής ενέργειας ενσωματωμένα συστήματα είτε για servers υψηλής απόδοσης.

Παρακάτω θα εξηγήσουμε πιο αναλυτικά τους λόγους που οι IoT συσκευές χρησιμοποιούν κατα κόρον επεξεργαστές βασισμένους σε ARM αρχιτεκτονικές:

Οι ARM επεξεργαστές είναι γνωστοί για την ενεργειακή τους αποδοτικότητα, κάτι που τους καθιστά κατάλληλους για συσκευές IoT που λειτουργούν με μπαταρίες. Η χαμηλή κατανάλωση ενέργειας είναι καίρια για συσκευές που ενδέχεται να λειτουργούν για μεγάλα χρονικά διαστήματα χωρίς επαναφόρτιση ή πρόσβαση σε εξωτερικές πηγές ενέργειας. Επιπλέον, οι επεξεργαστές ARM είναι συνήθως πιο οικονομικοί σε σύγκριση με άλλες αρχιτεκτονικές επεξεργαστών, ένας παραπάνω λόγος που τους καθιστά ελκυστικούς για τη μαζική παραγωγή συσκευών IoT.

Επιπροσθέτως, η ARM προσφέρει μια ευρεία γκάμα επεξεργαστών, από μικροελεγκτές χαμηλής κατανάλωσης έως πιο ισχυρούς επεξεργαστές εφαρμογών. Αυτή η επεκτασιμότητα επιτρέπει στους προγραμματιστές να επιλέξουν τον κατάλληλο επεξεργαστή για τη συγκεκριμένη εφαρμογή τους, ανεξάρτητα από το αν χρειάζεται ελάχιστη επεξεργαστική ισχύ ή προηγμένες υπολογιστικές δυνατότητες.

1.5 Πρόβλημα “κρύας” εκκίνησης στο Serverless:

Όπως αναφέραμε και προηγουμένως, η Serverless αρχιτεκτονική είναι η πιο δημοφιλής λύση για την ανάπτυξη λογισμικού των FaaS εφαρμογών στην υποδομή “νέφους”. Παρόλα αυτά, υπάρχουν και προβλήματα που “ψάχνουν” για λύσεις. Το πιο βασικό απο αυτά είναι η “κρύα” εκκίνηση (cold-start), που αναφέρεται στην κατάσταση της υποκείμενης υποδομης στην οποία τρέχει η συνάρτηση μας όταν θέλουμε να εξυπηρετήσει ένα αίτημά μας. Η υποδομή αυτή, μπορεί να είναι είτε κάποιο container, είτε κάποια εικονική μηχανή.

Όταν λοιπόν ένα αίτημα έρχεται προς εξυπηρέτηση, το υποκείμενο Serverless σύστημα θα ελέγξει εάν το container/εικονική μηχανή στο οποίο τρέχει η συνάρτηση μας είναι ήδη ενεργό και “τρέχει”. Όταν η υποδομή είναι διαθέσιμη, αναφερόμαστε στην διαδικασία εξυπηρέτησης ως “ζεστή” εκκίνηση (warm-start), αφού το αίτημα μπορεί να εξυπηρετηθεί άμεσα. Σε αντίθετη περίπτωση, υπάρχει καθυστέρηση η οποία οφείλεται στον χρόνο που απαιτείται για να στηθεί το περιβάλλον μέσα στο οποίο θα εκτελεστεί η συνάρτηση μας. Τα αιτήματα προς τις εικονικές μηχανές, είναι συνήθως “κρύα”, γεγονός που σημαίνει ότι η εκάστοτε εικονική μηχανή δεν είναι διαθέσιμη για να εξυπηρετήσει το αίτημα μας.

Αποκαλούμε την εικονική μηχανή που τρέχει η συνάρτηση μας εργατικό κόμβο (worker node) και η διαχείριση του είναι ευθύνη του παρόχου της εκάστοτε πλατφόρμας. Το πρόβλημα που εξετάζουμε σχετίζεται με την καθυστέρηση που προκαλείται στην ενεργοποίηση του κάθε εργατικού κόμβου ώστε να είναι έτοιμος να εξυπηρετήσει την πρώτη κλήση προς κάποια συνάρτηση. Η διαδικασία αυτή ονομάζεται cold-start. Δεν πρέπει να ξεχνάμε άλλωστε ότι βασική αρχή της Serverless αρχιτεκτονικής είναι ότι χρησιμοποιεί τους διαθέσιμους πόρους μόνο όταν απαιτούνται.

Μια απο τις λύσεις που έχουν εξεταστεί ενδελεχώς στον τομέα της έρευνας είναι η χρήση στιγμιότυπων (snapshots). Στο πλαίσιο του Serverless, τα στιγμιότυπα αναφέρονται στην δυνατότητα να αποθηκεύεται η κατάσταση της εικονικής μηχανής, με στόχο την επαναφορά της (απο το σημείο εκτέλεσης που έγινε η λήψη του στιγμιότυπου) και εν τέλει την μείωση του χρόνου εκκίνησης. Θα αναλύσουμε στο επόμενο κεφάλαιο τις διαφορετικές τεχνικές στιγμιότυπου που έχουν μελετηθεί από την τεχνολογική κοινότητα. Στην παρούσα διπλωματική εξετάσαμε ενδελεχώς την σχεδίαση ονόματι SnapFaas από το πανεπιστήμιο του Princeton [30].

Η συγκεκριμένη τεχνική υποστηριζόταν αποκλειστικά σε υπολογιστικά συστήματα x86 αρχιτεκτονικής. Λόγω της αυξανόμενης τάσης στην μεταφορά της Serverless αρχιτεκτονικής σε IoT συσκευές, όπως εξηγήσαμε και προηγουμένως, πραγματοποιήσαμε και μελετήσαμε την μεταφορά του SnapFaas εργαλείου σε ARM αρχιτεκτονικές.

1.6 Σύγκριση των αρχιτεκτονικών ARM & x86 στην εκκίνηση μέσω στιγμιότυπου

Η χρήση του snapshotting μηχανισμού για την βελτίωση της εκκίνησης μιας μικρο-εικονικής μηχανής (microVM) είναι ευρέως διαδεδομένη στα υπολογιστικά συστήματα που χρησιμοποιούν x86 επεξεργαστές. Για να καταλάβουμε αν αξίζει η μεταφορά του συγκεκριμένου εργαλείου σε ARM αρχιτεκτονικές και κατ' επέκταση σε IoT συσκευές, εκτελέσαμε το εξής πείραμα: Για την εκκίνηση του microVM, την λήψη του στιγμιότυπου και την επαναφορά της μηχανής χρησιμοποιήσαμε την τεχνολογία εικονικοποίησης Firecracker της Amazon [13], που είναι και η βάση για το εργαλείο SnapFaas που μελετήσαμε. Το Firecracker είναι μια open source τεχνολογία εικονικοποίησης σχεδιασμένη ειδικά για Serverless υπολογιστικά περιβάλλοντα. Η βασική του λειτουργία είναι η ελαφριά, γρήγορη και ασφαλής εκτέλεση μικρο-εικονικών μηχανών (microVMs), οι οποίες με την σειρά τους παρέχουν ένα lightweight, container-like περιβάλλον εκτέλεσης για συναρτήσεις-σαν-υπηρεσία (FaaS) εφαρμογών.

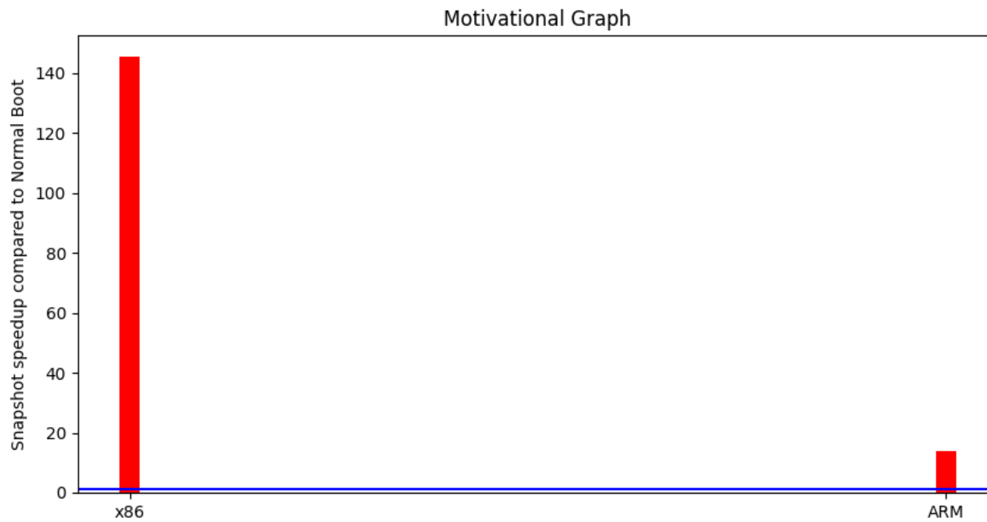
Εκκινήσαμε λοιπόν ενα microVM με την κανονική διαδικασία εκκίνησης (normal boot process) και μετρήσαμε την χρονική διάρκεια μέχρι η εικονική μηχανή να είναι έτοιμη να δεχθεί κάποιο αίτημα για επεξεργασία. Έπειτα, πραγματοποιήσαμε λήψη ενός στιγμιότυπου του microVM στο σημείο που είναι ήδη έτοιμο να εξυπηρετήσει ένα αίτημα. Στην συνέχεια, έγινε επαναφορά του microVM μέσω του στιγμιότυπου αυτού στην κατάσταση έτοιμη να δεχθεί κάποιο request και μετρήσαμε την χρονική διάρκεια αυτή. Αυτή η διαδικασία εφαρμόστηκε και για τις δύο αρχιτεκτονικές, χρησιμοποιώντας τα ίδια χαρακτηριστικά στα microVM που "σηκώθηκαν" για την εκάστοτε περίπτωση. Στον Πίνακα 1 παρουσιάζουμε τους χρόνους για x86 και ARM αρχιτεκτονική. Στις γραμμές έχουμε τους τέσσερις διαφορετικούς τρόπους εκκίνησης (κανονική εκκίνηση και αποκατάσταση για x86 και ARM αντίστοιχα), ενώ στις στήλες είναι ο μέσος όρος και η τυπική απόκλιση για δέκα εκτελέσεις:

τρόπος εκκίνησης / τύπος μέτρησης	μέσος όρος (us)	τυπική απόκλιση (us)
normal boot x86	2308617.7	111190.36
snap x86	15889.9	6136.28
normal boot ARM	1111989.4	165976.17
snap ARM	80202.1	9623.75

Πίνακας 1: Πίνακας με τους χρόνους εκκίνησης σε x86 και ARM συστήματα

Σε αυτό το σημείο αξίζει να σημειώσουμε ότι για το x86 σύστημα χρησιμοποιήσαμε ένα MSI laptop με τα εξής χαρακτηριστικά: 8 GB RAM, 6 CPUs, Thread(s) per core: 1, Model name: Intel Core Processor (Skylake, IBRS) και CPU MHz: 2808.006. Αντίστοιχα, για το ARM σύστημα χρησιμοποιήσαμε ένα RaspberryPi 4 Model B με τα εξής χαρακτηριστικά: RAM: 8GB, CPUs: 8, Thread(s) per core: 2 και CPU MHz: 2800.000.

Παρατηρούμε ότι μέσω της αποκατάστασης της μηχανής και εξαλείφοντας όλες τις χρονοβόρες διαδικασίες εκκίνησης, η εκκίνηση της μηχανής συμβαίνει γύρω στις 145 φορές γρηγορότερα. Όπως είναι φανερό, η εφαρμογή στιγμιότυπων σε x86 αρχιτεκτονικές μπορεί να μας ωφελήσει σε μεγάλο βαθμό στις FaaS εφαρμογές, καθώς εξοικονομούμε σημαντικό χρόνο ώστε η εικονική μηχανή να είναι έτοιμη να εξυπηρετήσει κάποιο αίτημα.
Για την aarch64 αρχιτεκτονική εξακολουθούμε να βλέπουμε βελτίωση στην διαδικασία εκκίνησης, αλλά με μικρότερο speedup (x13 φορές γρηγορότερο αυτή τη φορά), γεγονός που είναι σημαντικό αν αναλογιστούμε τον περιορισμό των πόρων που έχει μια συσκευή σαν το Raspberry Pi. Παραδείγματος χάριν στην περίπτωση του storage, η microSD που χρησιμοποιούμε μας εξασφαλίζει μια ταχύτητα της τάξεως των 10MB/s η οποία όμως θεωρείται πολύ αργή σε σύγκριση με άλλες διαθέσιμες που συναντάμε σε μεγαλύτερα υπολογιστικά συστήματα (π.χ. SSD). Η σχηματική αναπαράσταση των speedups που προσφέρει η αποκατάσταση μέσω snapshot, παρουσιάζεται στο Σχήμα 1.



Σχήμα 1: Snapshot speedups compared to normal boot on x86 & ARM architectures

Συμπερασματικά, η εφαρμογή στιγμιότυπων είναι έγκυρη σε ARM αρχιτεκτονικές αφού κερδίζουμε χρονικά στην αποκατάσταση του microVM και θα προσπαθήσουμε να βελτιώσουμε ακόμα περαιτέρω την "κρύα" εκκίνηση μέσω του SnapFaaS εργαλείου.

2 Θεωρητικό Υπόβαθρο - Εργαλεία

2.1 Ορισμοί

Πριν προβούμε στην ανάλυση των προϋπάρχοντων εργαλείων που κάνουν χρήση στιγμιοτύπων για την μείωση του χρόνου εκκίνησης μιας υποκείμενης υποδομής (είτε αυτή πρόκειται για microVM, container είτε ακόμα και unikernel), πρέπει να αναλύσουμε κάποιες έννοιες που θα φανούν βοηθητικές για την κατανόηση των μηχανισμών των εργαλείων αυτών.

Το **snapshot**, αναφέρεται σε μια αποθηκευμένη κατάσταση ή εικόνα (image) ενός περιβάλλοντος εκτέλεσης μιας συνάρτησης σε ένα συγκεκριμένο σημείο εκτέλεσης. Το snapshot αυτό, περιλαμβάνει όλες τις απαραίτητες εξαρτήσεις (dependencies), διαμορφώσεις (configurations) και πόρους που απαιτούνται για την εκτέλεση της συνάρτησης. Η διαδικασία δημιουργίας και διαχείρισης ενός snapshot περιλαμβάνει:

→ “Αιχμαλώτιση” κατάστασης: Η εκάστοτε πλατφόρμα καταγράφει την κατάσταση ενός περιβάλλοντος εκτέλεσης κατά την διάρκεια κλήσης και εκτέλεσης μιας Function-as-a-Service συνάρτησης. Αυτό περιλαμβάνει dependencies, runtime configurations και τυχόν προσαρμοσμένες ρυθμίσεις ειδικά για την εκάστοτε λειτουργία.

→ Αποθήκευση: Το captured snapshot αποθηκεύεται σε έναν μόνιμο μηχανισμό αποθήκευσης ώστε μετέπειτα να μπορέσει να φορτωθεί στην μνήμη της υποκείμενης υποδομής. Αυτό διασφαλίζει ότι το στιγμιότυπο μπορεί να ανακτηθεί αποτελεσματικά και να χρησιμοποιηθεί ξανά για επόμενες κλήσεις.

→ Επαναφορά: Όταν συμβαίνει μια κλήση συνάρτησης, η πλατφόρμα ανακτά το κατάλληλο στιγμιότυπο απο το storage και το επαναφέρει. Αυτή η διαδικασία είναι ταχύτερη από την δημιουργία ενός νέου περιβάλλοντος εκτέλεσης από την αρχή, οδηγώντας σε μειωμένη χρονική καθυστέρηση για την εξυπηρέτηση μιας κλήσης συνάρτησης.

Το **cold-start** αναφέρεται στην περίπτωση που μία συνάρτηση καλείται στο Serverless περιβάλλον και η υποκείμενη υποδομή δεν είναι ενεργή. Σε αυτή την περίπτωση το Serveless platform πρέπει να δημιουργήσει μια εικονική μηχανή είτε ένα άλλο περιβάλλον χρόνου εκτέλεσης (runtime environment) το οποίο θα χειριστεί το επικείμενο request. Η παραπάνω διαδικασία περιλαμβάνει τα εξής βήματα:

→ Αρχικοποίηση runtime environment: Η πλατφόρμα αρχικοποιεί ένα container ή ένα microVM, μέσω της κατανομής πόρων όπως ο επεξεργαστής, η μνήμη και η δικτύωση.

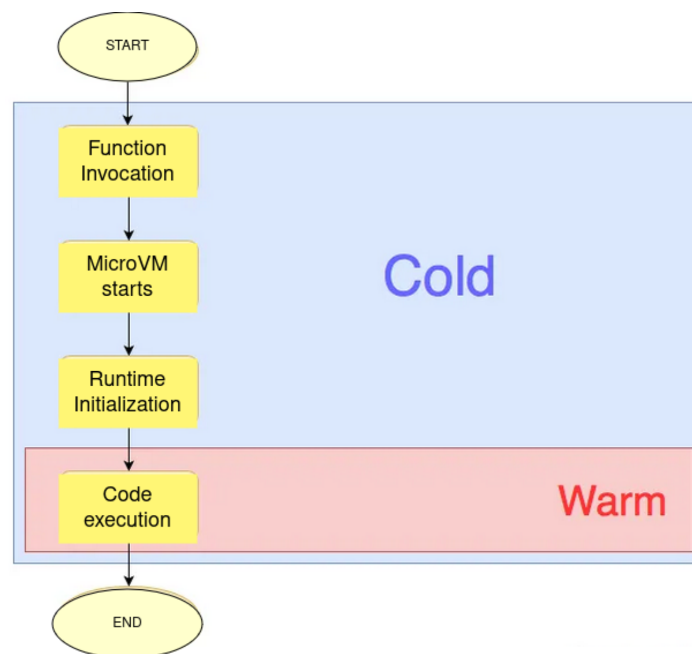
→ Application Bootstrapping: Σε αυτό το στάδιο, ο διαχειριστής της εικονικής μηχανής φορτώνει το κώδικα της εφαρμογής και τυχόν εξαρτήσεις αυτής, απαραίτητες για την

εκτέλεση της. Αυτό μπορεί να περιλαμβάνει την λήψη κώδικα απο κάποιο storage, την αρχικοποίηση βιβλιοθηκών και το σετάρισμα μεταβλητών περιβάλλοντος.

→ Function invocation: Όταν το επικείμενο περιβάλλον είναι έτοιμο, η εκάστοτε πλατφόρμα καλεί την συνάρτηση ώστε να επεξεργαστεί το εισερχόμενο αίτημα.

Η χρονική διάρκεια για να ολοκληρωθούν τα παραπάνω βήματα και να κάνουν διαθέσιμη την συνάρτηση για εξυπηρετήσει ένα request, ονομάζεται χρόνο "κρύας" εκτέλεσης (cold start time)

Το **warm-start** αναφέρεται στην διαδικασία εξυπηρέτησης ενός αιτήματος (request) όταν αυτό έχει εξυπηρετηθεί στο πρόσφατο παρελθόν και έτσι η υποκείμενη υποδομή που κάνει host την Function-as-a-Service εφαρμογή μας έχει ήδη ένα προϋπάρχων περιβάλλον εκτέλεσης έτοιμο να χειριστεί το αίτημα (idle state). Σε αντίθεση με το cold-start, το warm-start αξιοποιεί ένα υπάρχων περιβάλλον που έχει χρησιμοποιηθεί πρόσφατα για την εκτέλεση της ίδιας ή παρόμοιας εφαρμογής και έτσι για την εξυπηρέτηση ενός αιτήματος η καθυστέρηση είναι μειωμένη. Οι διαφορές cold/warm-start παρουσιάζονται στο Σχήμα 2.



Σχήμα 2: Σύκγριση cold/warm-start

Σε αυτό το σημείο θα εισάγουμε και μία ακόμα έννοια που θα συναντήσουμε στο πειραματικό σκέλος της παρούσας διπλωματικής, το **lukewarm-start**. Σε αυτή την περίπτωση, η υποκείμενη υποδομή "χτίζεται" απο το μηδέν, όπως και στην περίπτωση του cold-start, με την διαφορά ότι η page cache του guest δεν αδειάζεται (όπως συμβαίνει στο cold-start). Αυτό έχει σαν αποτέλεσμα, όταν πάμε να εκκινήσουμε μια εικονική μηχανή μέσω της αποκατάστασης ενός στιγμιότυπου, οι σελίδες του snapshot να υπάρχουν ήδη στην page cache και έτσι να γίνονται μόνο minor page faults ώστε να γεμίσει ο πίνακας σελίδων (page table). Για να καταλάβουμε πως ακριβώς διαφοροποιείται η συγκεκριμένη τεχνική εκκίνησης από την "κρύα¨ εκκίνηση, πρέπει να σημειώσουμε ότι στην περίπτωση

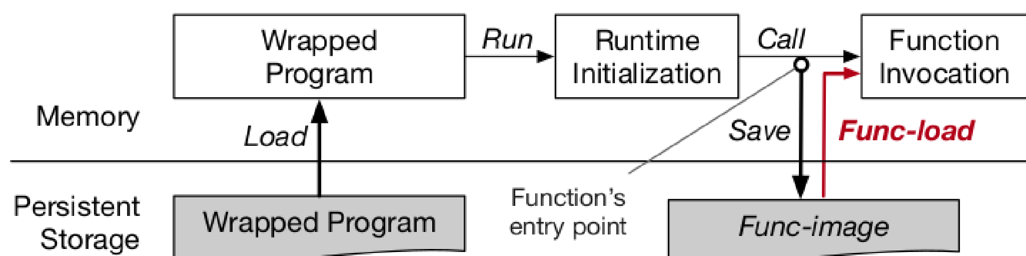
των cold-starts, έχουμε major page faults, το οποίο συμπεριλαμβάνει το allocation των σελίδων και την χρονική διάρκεια για να ολοκληρωθούν οι Input και Output διεργασίες για να γίνουν fetch οι σελίδες αυτές απο τον δίσκο.

2.2 Μηχανισμοί Στιγμιοτύπων

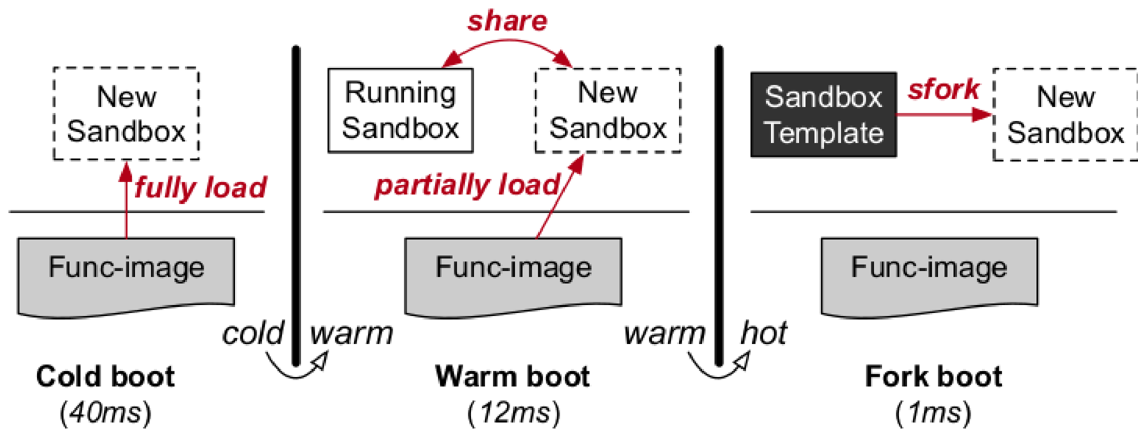
Προηγούμενες δουλειές στον τομέα της έρευνας έχουν προτείνει διάφορες τεχνικές μνημονοποίησης (memoization), κατάλληλες για διαφορετικά περιβάλλοντα εκτέλεσης, με σκοπό την βελτίωση των cold starts. Το Catalyzer [9] έχει σχεδιαστεί πάνω στο gVisor [10] – εργαλείο δημιουργίας Linux compatible sandbox περιβαλλόντων της Google – το SOCK [11] είναι υλοποιημένο για Docker [8], το SEUSS για unikernel [12] εικονικές μηχανές, ενώ το REAP είναι βασισμένο στις μικρο-εικονικές μηχανές (microVMs) του Firecracker – ένας διαχειριστής εικονικής μηχανής (VMM) που χρησιμοποιεί το KVM (Kernel-based Virtual Machine) για να δημιουργήσει και να διαχειριστεί μικρο-εικονικές μηχανές.

Παρά τα διαφορετικά περιβάλλοντα εκτέλεσης και τεχνικές υλοποίησης, οι παραπάνω μηχανισμοί μοιράζονται τις ίδιες ιδέες υψηλού επιπέδου - η μνημονοποίηση μέσα στο στιγμιότυπο θα πρέπει να “αιχμαλωτίζει” όσο το δυνατόν περισσότερο υπολογισμό αρχικοποίησης και η επαναφορά του εκάστοτε περιβάλλοντος εκτέλεσης θα πρέπει να μειώνει την ποσότητα της κατάστασης που αποκαθίσταται απο τον δίσκο.

Ένας ευθύς τρόπος μνημονοποίησης, είναι ένα στιγμιότυπο που λαμβάνεται μετά την αρχικοποίηση της συνάρτησης (πλήρες στιγμιότυπο), το οποίο “αιχμαλωτίζει” την κατάσταση εκτέλεσης της συνάρτησης πριν αυτή κληθεί. Ο μηχανισμός Catalyzer λειτουργεί καθ’ αυτόν τον τρόπο (func-image): μια εικόνα συνάρτησης παράγεται μέσω του Docker μηχανισμού checkpoint – ένας πειραματικός μηχανισμός που επιτρέπει το “πάγωμα” ενός τρέχοντος container (η λειτουργία *Save* στο Σχήμα 3), προσδιορίζοντας ένα σημείο ελέγχου το οποίο μετατρέπει το container σε μια συλλογή αρχείων στο δίσκο. Στην συνέχεια, το container μπορεί να αποκατασταθεί απο το σημείο που βρισκόταν όταν δόθηκε η εντολή checkpoint. Περιέχει την μνήμη και τα μεταδεδομένα του guest (της συνάρτησης και του περιβάλλοντος εκτέλεσης), από την πλευρά του host. Η εκκίνηση από ένα func-image ανακατασκευάζει τον χώρο διευθύνσεων του φιλοξενούμενου (guest address space) μέσω σελίδωσης-απαίτησης (demand paging) χρησιμοποιώντας την τεχνική file-mmap (χαρτογράφηση αρχείων στην μνήμη). Η on-demand επαναφορά αποφεύγει την προ-ανάκτηση ολόκληρης της κατάστασης από τον δίσκο, αλλά έχει το μειονέκτημα της καθυστέρησης των σύγχρονων σφαλμάτων σελίδας (page faults) κατά την διάρκεια της εκτέλεσης. Μια σχηματική παρουσίαση του Catalyzer μηχανισμού παρουσιάζεται στο Σχήμα 4.

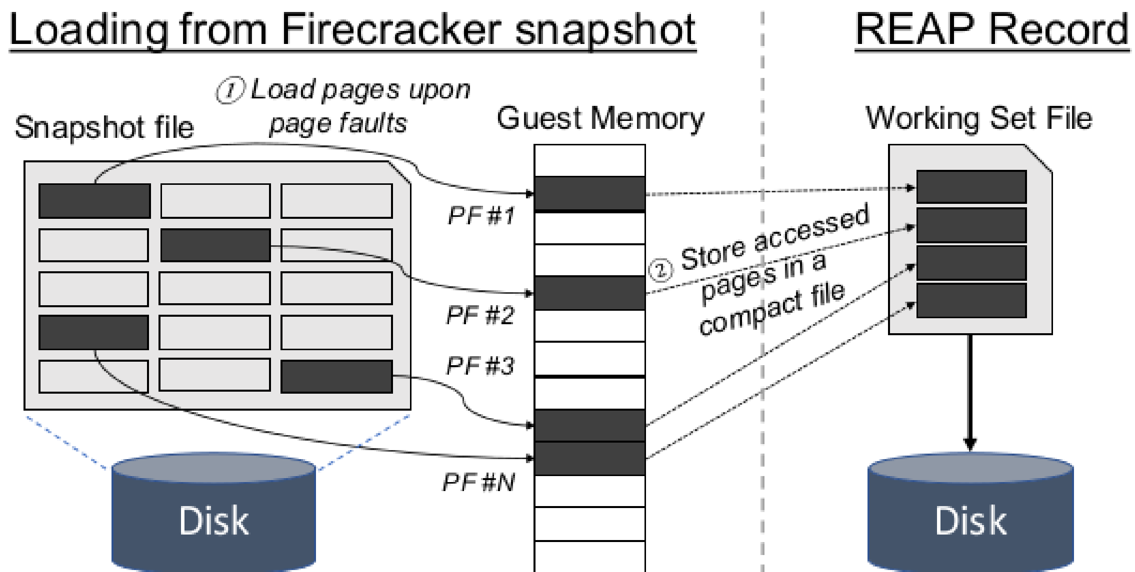


Σχήμα 3: Η εκκίνηση χωρίς αρχικοποίηση του Catalyzer [9]

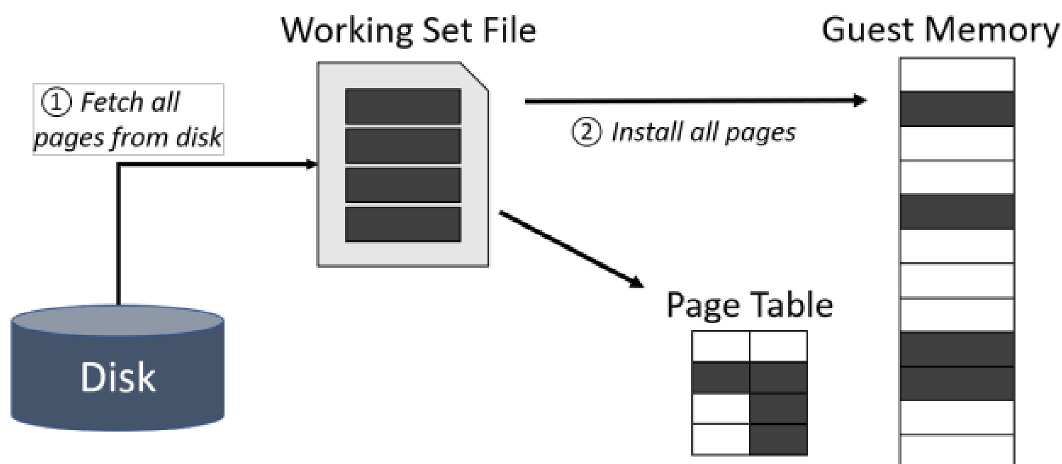


Σχήμα 4: Συνοπτική παρουσίαση του Catalyzer μηχανισμού [9]

Το REAP [29] αναγνωρίζει ότι τα σύγχρονα σφάλματα σελίδων (αυτά που συμβαίνουν κατα την διάρκεια της εκτέλεσης), επιβάλλουν υψηλή καθυστέρηση στην εκτέλεση και προτείνει μια βελτιστοποίηση που προ-ανακτά μόνο ένα ενεργό σύνολο σελίδων (working set). Συγκεκριμένα, μετά την λήψη ενός πλήρους στιγμιότυπου συνάρτησης (όπου εμπεριέχεται ολόκληρη η μνήμη της αρχικοποιημένης συνάρτησης), το REAP εκτελεί μια φορά την συνάρτηση παρατηρώντας ποιές σελίδες από το snapshot προσπελάστηκαν πραγματικά κατά την διάρκεια της εκτέλεσης, όπως φαίνεται και στο Σχήμα 5. Στις μελλοντικές εκτελέσεις, το REAP προ-ανακτά μόνο αυτό το υποσύνολο μνήμης αφήνοντας την υπόλοιπη μνήμη στο δίσκο για demand-paging (ανάκτηση των σελίδων από τον δίσκο μόνο στην περίπτωση που πάνε να προσπελαστούν κατα την διάρκεια της εκτέλεσης), όπως απεικονίζεται και στο Σχήμα 6. Η ανάκτηση κατά δέσμες (in-batch prefetch) μειώνει σημαντικά την συνολική καθυστέρηση μέσω της μείωσης των σύγχρονων σφαλμάτων σελίδας στο δίσκο κατά την διάρκεια της εκτέλεσης.



Σχήμα 5: Η φάση εύρεσης του working set της συνάρτησης στο REAP [29]



Σχήμα 6: Η ανάκτηση των σελίδων και η εγκατάσταση τους στην μνήμη της εικονικής μηχανής στο εργαλείο REAP [29]

Τα υπόλοιπα εργαλεία λήψης στιγμιότυπων ακολουθούν διαφορετική προσέγγιση. Αποθηκεύουν στην cache ένα μέρος της κατάστασης, που είναι κοινή και επομένως μπορεί να μοιραστεί ανάμεσα σε πολλές συναρτήσεις στην μνήμη. Συνήθως αυτή η μερικώς αποθηκευμένη κατάσταση περιλαμβάνει μέχρι και την αρχικοποίηση της γλώσσας εκτέλεσης (η προγραμματιστική γλώσσα στην οποία έχει γραφτεί η εφαρμογή), αλλά όχι την αρχικοποίηση της συνάρτησης.

Σε αυτό το πλαίσιο, το Catalyzer για να βελτιώσει περαιτέρω τα cold starts, προτείνει το template γλώσσας Zygote. Το Zygote είναι ένα container μέσα στο οποίο έχει ολοκληρωθεί μια εν μέρη αρχικοποίηση και από το οποίο μπορεί να παραχθεί και να εξατομικευτει (ως προς την εκάστοτε συνάρτηση) ένα καινούργιο container. Για παράδειγμα, συναρτήσεις της γλώσσας Python μπορούν να δημιουργηθούν από το ίδιο Python Zygote container. Οι συναρτήσεις αρχικοποιούνται από Zygotes χρησιμοποιώντας την κλήση συστήματος fork για τη δημιουργία ενός Zygote κλώνου μέσα στον οποίο θα φορτωθεί ο κώδικας και οι βιβλιοθήκες της συνάρτησης.

Το SEUSS χρησιμοποιεί runtime στιγμιότυπα βασισμένα σε εικονικές μηχανές (virtual machines) για την εξυπηρέτηση "κρύων" αιτημάτων. Ένα runtime στιγμιότυπο περιλαμβάνει την φυσική μνήμη της εικονικής μηχανής από την στιγμή που το initialization του language runtime ολοκληρώνεται. Οποιαδήποτε συνάρτηση γραμμένη στην ίδια γλώσσα μπορεί να κάνει boot απο το ίδιο runtime στιγμιότυπο. Στην συνέχεια η αρχικοποίηση της συνάρτησης ξεκινά από το σημείο που σταμάτησε το runtime μέσα στο στιγμιότυπο. Για την διαχείριση των σελίδων μνήμης, χρησιμοποιείται η κλήση συστήματος mmap.

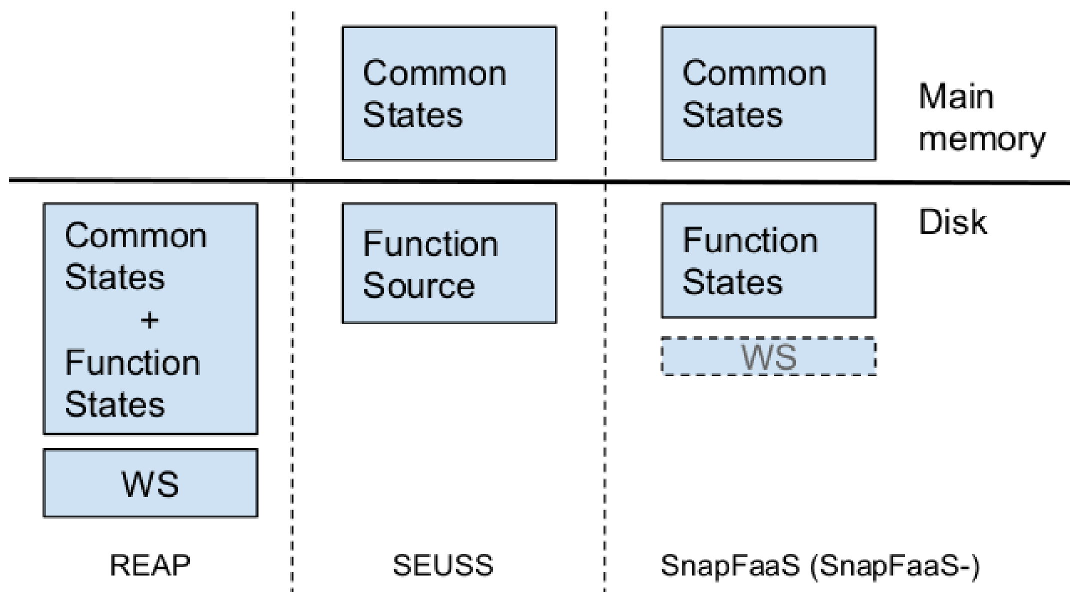
Πρέπει να σημειώσουμε ότι όταν διαχωρίζουμε και αποθηκεύουμε την κοινή runtime κατάσταση (οι βιβλιοθήκες και ο κώδικας πριν εκτελεστεί η συνάρτηση), οι παραπάνω σχεδιασμοί αποτυγχάνουν να κρατήσουν στην μνήμη την αρχικοποίηση που απαιτείται για τις ίδιες τις συναρτήσεις όπως κάνουν τα στιγμιότυπα πλήρους συναρτησης (full-function snapshots). Τα στιγμιότυπα συναρτήσεων (σαν μια βελτιωμένη έκδοση του runtime στιγμιότυπου) "αιχμαλωτίζουν" τις σελίδες στην μνήμη που τροποποιούνται κατα την αρχικοποίηση που απαιτείται για την εκτέλεση της συνάρτησης, ξεκινώντας από το σωστό runtime snapshot. Η αποθήκευση της "πλήρους κατάστασης συνάρτησης" στην μνήμη είναι γρήγορη και μπορεί να βοηθήσει στην βελτίωση της κλιμακωσιμότητας όταν ένα instance

συνάρτησης λειτουργεί ήδη. Ωστόσο κάθε τέτοια αποθηκευμένη κατάσταση καταναλώνει μνήμη ανάλογη του πλήθους των συναρτήσεων, επομένως ο συγκεκριμένος σχεδιασμός δεν είναι κατάλληλος για την επιτάχυνση των "κρύων" εκκινήσεων όταν οι συναρτήσεις που ενδέχεται να κληθούν είναι πολλές. Για την επιλυση του συγκεκριμένου ζητήματος, υλοποιήθηκε το εργαλείο SnapFaas το οποίο μελετήσαμε και επεκτείναμε.

2.3 SnapFaas

Όπως αναφέραμε και προηγουμένως, μελετήσαμε ενδελεχώς το εργαλείο SnapFaas που δημιουργήθηκε για ακριβώς αυτό το πρόβλημα του cold-start που προσθέτει καθυστέρηση στην εκτέλεση Function-as-a-Service εφαρμογών. Είναι ένας μηχανισμός στιγμιότυπου, βασισμένος στο εργαλείο διαχείρισης microVMs Firecracker της Amazon.

Σε μια υψηλότερου επιπέδου ανάλυση, η σχεδίαση του αποτελείται από ένα στιγμιότυπο βάσης ή γλώσσας (base/language snapshot – το στιγμιότυπο που περιλαμβάνει όλες εκείνες τις αρχικοποιήσεις που απαιτούνται για την εκκίνηση της μηχανής και την εγκατάσταση της γλώσσας εκτέλεσης και είναι κοινό για τις εικονικές μηχανές με την ίδια γλώσσα εκτέλεσης) αποθηκευμένο στην μνήμη της εικονική μηχανής καθώς και ένα στιγμιότυπο διαφοράς ή συνάρτησης (diff/function snapshot – το στιγμιότυπο που περιλαμβάνει το workload της κάθε εφαρμογής και είναι ξεχωριστό για κάθε συνάρτηση) αποθηκευμένο στον δίσκο. Η επιλογή τοποθεσίας αποθήκευσης των δυο αυτών στιγμιότυπων είναι άμεσα συνδεδεμένη με τον τρόπο επαναφοράς των αρχείων μνήμης τους που θα αναλύσουμε και στην συνέχεια. Επιπλέον στον αποθηκευτικό χώρο του δίσκου, βρίσκεται και το ενεργό σύνολο σελίδων (working set – WS) της κάθε συνάρτησης στο οποίο αναφερθήκαμε και προηγουμένως και θα εξηγήσουμε περαιτέρω, στο κομμάτι της τεχνικής περιγραφής.



Σχήμα 7: Σύγκριση του SnapFaaS με τα υπάρχοντα στιγμιότυπα για "κρύες" εκκινήσεις. Το REAP καταγράφει την κατάσταση εκτέλεσης ως ένα ολοκληρωμένο σύνολο και φορτώνει μέσω σελίδωσης-απαίτησης τις σελίδες του working set. Το SEUSS επαναφέρει την κοινή κατάσταση που είναι αποθηκευμένη στην cache κατά απαίτηση και στη συνέχεια εισάγει τις συναρτήσεις από την πηγή. Το SnapFaaS αποθηκεύει στην cache την κοινή κατάσταση και αποθηκεύει τις καταστάσεις των συναρτήσεων στο δίσκο και φορτώνει άμεσα (eagerly) μόνο το σύνολο εργασίας. [30]

Ένας βασικός στόχος του SnapFaas, για να μειώσει σημαντικά την καθυστέρηση του cold-start, είναι να μεγιστοποιήσει τις σελίδες που εδρεύουν στην μνήμη και που μοιράζονται μεταξύ των microVMs (και περιλαμβάνονται στο στιγμιότυπο βάσης) και παράλληλα να μειώσει την αναλογία αυτών που όντως χρησιμοποιούνται κατά την διάρκεια της εκτέλεσης. Αυτή είναι και η αιτία πίσω από την δημιουργία των δύο διαφορετικών στιγμιοτύπων, της βάσης, για την εκτέλεση που είναι κοινή και της διαφοράς, που διαφοροποιείται για κάθε συνάρτηση και για τις βιβλιοθήκες που η καθεμία εισαγάγει.

Πιο συγκεκριμένα, το στιγμιότυπο βάσης περιλαμβάνει την μνήμη που τροποποιείται από τον πυρήνα και το λειτουργικό σύστημα κατα την διάρκεια της εκκίνησης (booting phase), από την φόρτωση της γλώσσας εκτέλεσης και των απαραίτητων βιβλιοθηκών που αυτή χρειάζεται καθώς και από το SnapFaas εκτελέσιμο. Επειδή ακριβώς οι συναρτήσεις εκτελούνται μετά από το σημείο που συνεχίζεται η εκτέλεση με την αποκατάσταση του στιγμιότυπου βάσης, οι σελίδες μνήμης που τροποποιούνται, από την εκάστοτε συνάρτηση, είναι λίγες συγκριτικά με το συνολικό πλήθος που είναι αποθηκευμένο στο αρχείο μνήμης (ποσοστό της τάξεως του 5% κατα μέσο όρο). Αυτό έχει ως αποτέλεσμα το πλήθος των πραγματικά διαμοιραζόμενων σελίδων να είναι μικρό και το στιγμιότυπο βάσης να μπορεί να μοιραστεί μεταξύ πολλών συναρτήσεων.

Το diff στιγμιότυπο περιλαμβάνει μνήμη που έχει αρχικοποιηθεί ή τροποποιηθεί από την ίδια την συνάρτηση καθώς και από τις βιβλιοθήκες που εισάγει. Σε κάποιες εκ των περιπτώσεων, οι συγκεκριμένες σελίδες μνήμης μπορεί να επικαλύπτονται με τις σελίδες μνήμης του base snapshot. Τότε, οι σελίδες του diff "πανωγράφουν" τις σελίδες μνήμης του base στιγμιότυπου. Μία από τις σημαντικότερες τεχνικές που έχουν χρησιμοποιηθεί στο SnapFaas και το κάνουν να ξεχωρίζει ως ένα εργαλείο γρήγορης "κρύας" εκκίνησης μικρο-εικονικών μηχανών, είναι η χρησιμοποίηση του συστήματος αρχείων εφαρμογής

(Application File System – AppFS). Για την δημιουργία των εικόνων συστήματος αρχείου (file system images), που χρησιμοποιουνται απο το SnapFaas, ακολουθείται η ίδια διαδικασία που παρέχεται από το Firecracker [15]. Είναι βασισμένες στο Alpine Linux 3.10 [17], μια Linux έκδοση που χρησιμοποιεί το ελαφρύ UNIX utilities BusyBox, το σύστημα έναρξης (init system) OpenRC [16] και τον πυρήνα Linux έκδοσης 4.20 μεταγλωττισμένο με τις ελάχιστες δυνατές διαμορφώσεις. Τα εκτελέσιμα που περιλαμβάνει πρέπει και αυτά να έχουν μεταγλωτιστεί μέσα στο Alpine container για να μπορέσουν μετέπειτα να εκτελεστούν.

Προκειμένου να επιτευχθεί η στρωματοποίηση (layering) των diff στιγμιότυπων πάνω απο το base, είναι απαραίτητη η χρήση του AppFS καθώς είναι ο μοναδικός τρόπος να περιέχονται συγκεντρωτικά οι εξαρτήσεις και ο κώδικας της εφαρμογής σαν ξεχωριστή "οντότητα". Επιπλέον, το SnapFaas χρησιμοποιεί το σύστημα αρχείων ρίζας (RootFS), μέσα στο οποίο αποθηκεύονται λειτουργίες/προγράμματα κρίσιμα για την εκκίνηση της μηχανής και του λειτουργικού συστήματος καθώς και utilities της εκάστοτε γλώσσας. Με αυτόν τον τρόπο μπορούμε να διασφαλίσουμε ότι το στιγμιότυπο βάσης δεν θα περιλαμβάνει κάποια προσανατολισμένη στην συνάρτηση αρχικοποίηση, καθώς όταν φορτώνετε ένα block device (όπως ένα σύστημα αρχείου) στο microVM όλα τα μεταδεδομένα του αποθηκεύονται στην κρυφή μνήμη του συστήματος, μέσω του Linux πυρήνα, με αποτέλεσμα το στιγμιότυπο να "αιχμαλωτίζει" την διάταξη του filesystem. Η χρήση του AppFS λύνει αυτό ακριβώς το πρόβλημα διάταξης, καθώς όταν γίνεται λήψη και δημιουργία του base snapshot το AppFS δεν έχει φορτωθεί ακόμα. Επιπρόσθετα, αυτός ο διαχωρισμός των δύο συστημάτων αρχείου (RootFS – AppFS), μπορεί εύκολα να μας ξεχωρίσει τα διαφορετικά root file systems για κάθε μία υποστηριζόμενη γλώσσα εκτέλεσης.

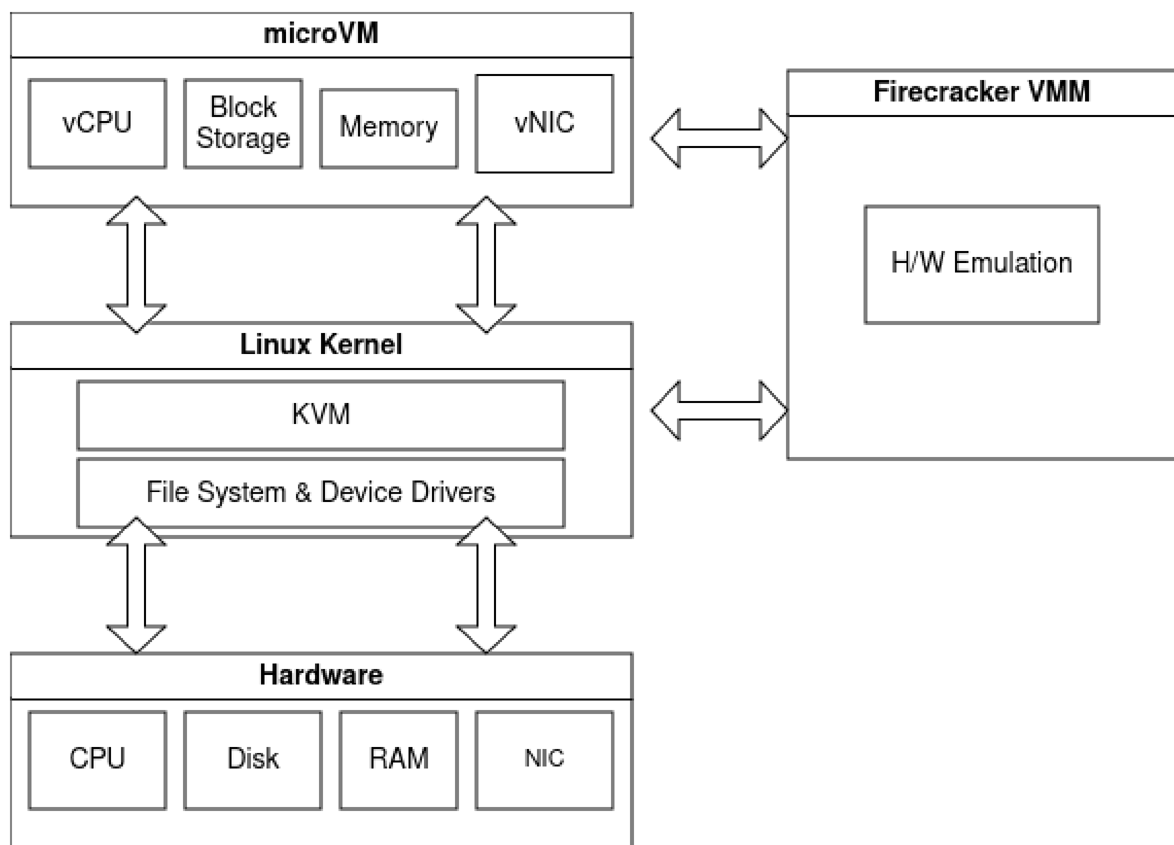
Ένας επιπλέον στόχος του SnapFaas συστήματος είναι να βρεί και να ξεχωρίσει το ελάχιστο σετ μοναδικών σελίδων που πρέπει να εδρεύουν στην μνήμη. Για να το πετύχει, επιστρατεύει την τεχνική του REAP εργαλείου που αναφέραμε και προηγουμένως. Πιο συγκεκριμένα, προσεγγίζει το ενεργό σύνολο σελίδων που χρησιμοποιεί μια συνάρτηση, εκτελώντας την πρώτα μια φορά. Στην συνέχεια το SnapFaas φορτώνει με άμεσο τρόπο (eager restoration – που θα εξηγήσουμε πιο αναλυτικά στην συνέχεια) το σύνολο αυτό των σελίδων στην μνήμη πριν απο την έναρξη της εκτέλεσης, αφού απο την στιγμή που οι πιθανότητες να χρησιμοποιηθούν είναι πολύ μεγάλες δεν χρειάζεται να τις φορτώσει on-demand (lazy restoration). Με αυτόν τον τρόπο μειώνονται οι σελίδες που φορτώνονται από τον δίσκο στην μνήμη (μια αργή διαδικασία στις περισσότερες των περιπτώσεων), καθώς πλέον δεν είναι απαραίτητο να γίνουν fetch όλες οι σελίδες του diff snapshot. Οι διαφορές του SnapFaas με τα εργαλεία στιγμιοτύπων REAP και SEUSS, αναφορικά με την τοποθεσία αποθήκευσης (δίσκος / κύρια μνήμη microVM) των διαφόρων καταστάσεων μνήμης απεικονίζεται στο Σχήμα 7. Ο πηγαίος κώδικας για το Snapfaas, του Princeton University, είναι ανεβασμένος στο github [31]

3 Τεχνική Περιγραφή και Υλοποίηση του SnapFass

3.1 Τεχνικές Λεπτομέρειες

Σε αυτό το κεφάλαιο θα αναλύσουμε σε μεγαλύτερο βάθος τις τεχνικές λεπτομέρειες του SnapFaas για να καταλάβουμε τον λόγο που παρουσιάζει μεγάλη βελτίωση στις “κρύες” εκκινήσεις των μικρο-εικονικών μηχανών. Επίσης, θα παρουσιάσουμε τις αλλαγές στις οποίες προβήκαμε για να καταφέρουμε να μεταφέρουμε το υπό μελέτη εργαλείο σε ARM αρχιτεκτονική καθώς όπως αναφέραμε και προηγουμένως, το SnapFaas είχε σχεδιαστεί αποκλειστικά για x86 υπολογιστικά συστήματα.

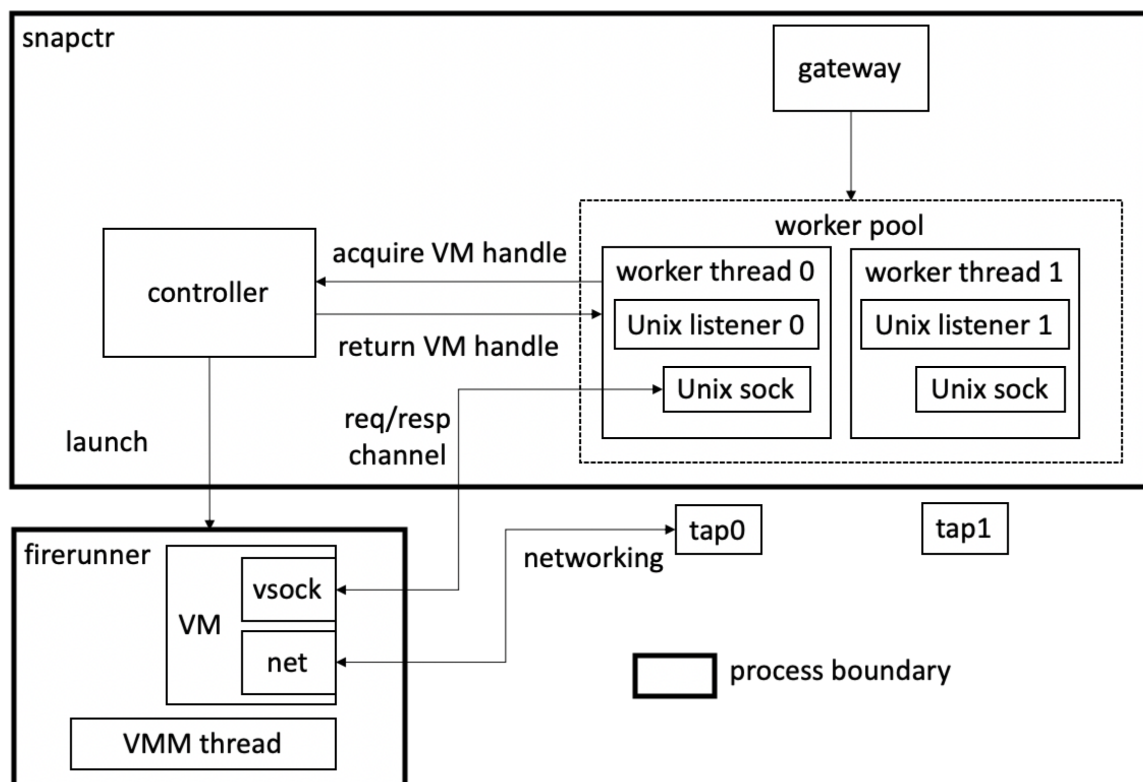
Η σχεδίαση του εργαλείου μας έχει βασιστεί πάνω στο εργαλείο διαχείρισης microVMs, Firecracker. Για να διαχειρίζεται τις εικονικές αυτές μηχανές, το Firecracker έχει τον ρόλο του διαχειριστή εικονικής μηχανής (Virtual Machine Manager) σε συνεργασία με τον KVM (Kernel-based Virtual Machine) hypervisor του Linux. Το KVM εικονικοποιεί τον επεξεργαστή και την μνήμη και η VMM διεργασία διαχειρίζεται το απαραίτητο input και output της εικονικής μηχανής, όπως φαίνεται και στο Σχήμα 8.



Σχήμα 8: Επισκόπηση των επιμέρους κομματιών του Firecrakcer

Το SnapFaas για να μπορέσει να διαχειριστεί τις διεργασίες οι οποίες προαναφέρθηκαν έχει υλοποιήσει μια σειρά απο wrappers εκτελέσιμα που επιτρέπουν στον χρήστη με την εκτέλεση μιας εντολής να εκκινήσει μια εικονική μηχανή είτε κανονικά από τον πυρήνα,

είτε μέσω ενός στιγμιότυπου. Μέσω αυτών των wrapper προγραμμάτων ο χρήστης περνάει την διαμόρφωση που επιθυμεί να έχει το VM, όπως τον πυρήνα του, το πλήθος των emulated επεξεργαστών, το μέγεθος της μνήμης και τα συστήματα αρχείων. Μέσα από τις παραμέτρους του singlevm εκτελέσιμου (το εργαλείο του SnapFaas που εκτελεί μία συνάρτηση μέσω ενος microVM) περνάμε και τα μονοπάτια των φακέλων που περιέχουν τα αντίστοιχα στιγμιότυπα με βάση τα οποία θέλουμε να αποκατασταθεί η εικονική μηχανή. Στο Σχήμα 10 βλέπουμε ένα παράδειγμα εκτέλεσης του singlevm. Στην συνέχεια και αφού περάσει ο έλεγχος της εκτέλεσης στο VMM, η εικονική μηχανή εγγράφεται στο Linux KVM, ζητώντας τους απαιτούμενους πόρους σε επεξεργαστές και μνήμη και την σύνδεση των παρεχόμενων συσκευών.



Σχήμα 9: Επισκόπηση των επιμέρους κομματιών του SnapFaas [31]

```
./singlevm --network --kernel /home/pi/preview_vmlinux.bin --kernel_args 'reboot=k panic=1 pci=off' \
    --rootfs ../../../snapfaas-images/rootfs/snapfaas/pythonfs.ext4 \
    --appfs ../../../snapfaas-images/appfs/python3/"$1"/output.ext2 \
    --load_dir /resources/images/snapshot/base_128/,/resources/images/snapshot/diff_128/lorem/ \
    --load_ws --vcpu_count 1 --mem_size 128 --firerunner ./firerunner < /resources/requests/lorem-python3.json
```

Σχήμα 10: Παράδειγμα εκτέλεσης της singlevm εντολής

Οι επιμέρους οντότητες του SnapFaas είναι γραμμένες στην γλώσσα προγραμματισμού Rust για μεγαλύτερη συμβατότητα με τον Διαχειριστή Εικονικής Μηχανής Firerunner με τον οποίο συνεργάζεται για την δημιουργία των εικονικών μηχανών. Ένα από τα μεγαλύτερα πλεονεκτήματα της Rust είναι η ασφάλεια μνήμης που παρέχει, εξαλείφωντας πολλά συνήθη σφάλματα κατά την διάρκεια της μεταγλώττισης όπως αναφορες σε μηδενικούς δείκτες (null pointer dereferences) και υπερχειλίσεις σε buffers (buffer overflows). Αυτός είναι και ο βασικός λόγος που το Firerunner χρησιμοποιεί την Rust καθώς μία από τις βασικές του λειτουργίες είναι η διαχείριση μνήμης. Επιπλέον, προσφέρει

μεγάλη απόδοση συγκρίσιμη με χαμηλού επιπέδου γλώσσες (όπως η C και η C++), παρέχοντας παράλληλα υψηλού επιπέδου δομές (higher level abstractions) καθώς και εγγυήσεις για την ασφάλεια της εφαρμογής.

Σε αυτό το σημείο αξίζει να σημειώσουμε ότι ο πυρήνας που παρέχεται στις μικρο εικονικές μηχανές που δημιουργούνται είναι "απογυμνωμένος" με στόχο να είναι όσο πιο ελαφρύς και αποδοτικός γίνεται, δίχως λειτουργίες που δεν θα χρησιμοποιηθούν εν τέλει απο την εικονική μηχανή που θα δημιουργηθεί, καθώς είναι πολύ συγκεκριμένη η λειτουργία που θα επιτελέσει (η εκτέλεση μιας εφαρμογής). Πέρα από τα δύο block devices, RootFS και AppFS στα οποία έχουμε αναφερθεί και μια IP συσκευή δικτύου για την επικοινωνία με τον έξω κόσμο, η εικονική μηχανή χρειάζεται και μια συσκευή εικονικής υποδοχής (virtual socket – vsock) μέσω της οποίας γίνεται εγκατάσταση της επικοινωνίας μεταξύ του guest/εικονικής μηχανής (οντότητα VM από Σχήμα 9) και του host (οντότητες worker thread 0/1 από Σχήμα 9).

3.1.1 Virtual Sockets - Virtio

Ένα σημαντικό device που χρησιμοποιεί το SnapFaas για την επικοινωνία μεταξύ του host και του microVM που εδρεύει στον guest, είναι το virtual socket. Τα virtual sockets λειτουργούν με παρόμοιο τρόπο με τα παραδοσιακά network sockets αλλά είναι βελτιστοποιημένα για την επικοινωνία μέσα σε ένα σύστημα. Ορίζουν μια καινούργια socket οικογένεια διευθύνσεων (AF_VSOCK) και χρησιμοποιούν την πρωτότυπη διεπαφή socket (socket(), connect(), bind(), listen(), accept()) για την εγκατάσταση επικοινωνίας και για την λήψη και μετάδοση δεδομένων. Επιπλέον, χρησιμοποιούν ένα ζεύγος ακέραιων [conext id, port] για την αναγνώριση των διεργασιών. Το host σύστημα έχει πάντα τον ακέραιο '2' για τιμή στην μεταβλητή context id, ενώ σε κάθε guest VM εκχωρείται ένα μοναδικό context id κατά την εκκίνηση του.

Τα Virtual sockets χρησιμοποιούνται ευρέως απο πλατφόρμες εικονικοποίησης (π.χ. QEMU/KVM) για να επιτρέψουν την επικοινωνία μεταξύ του οικοδεσπότη (host) και των εικονικών μηχανών ή containers που τρέχουν σε αυτόν – ή και των εικονικών μηχανών μεταξύ τους. Με αυτόν τον τρόπο γίνεται πιο εύκολο το χτίσιμο και η ανάπτυξη παράλληλων συστημάτων μέσα σε ένα virtualized περιβάλλον.

Τα virtual sockets, όπως και οι υπόλοιπες συνδεδεμένες συσκευές στο microVM, είναι virtio devices. Το virtio είναι μια προδιαγραφή paravirtualized συσκευών για την αποδοτική χρήση των πόρων του host. Προσφέρει έναν αποδοτικό και επεκτάσιμο μηχανισμό, ενώ βασίζεται σε ορολογία και μηχανισμούς φυσικών συσκευών, διευκολύνοντας την υποστήριξη και επιτρέποντας την επαναχρησιμοποίηση από υπάρχουσες υλοποιήσεις οδηγών στους guests.
Το πρότυπο virtio έχει δύο βασικούς άξονες:

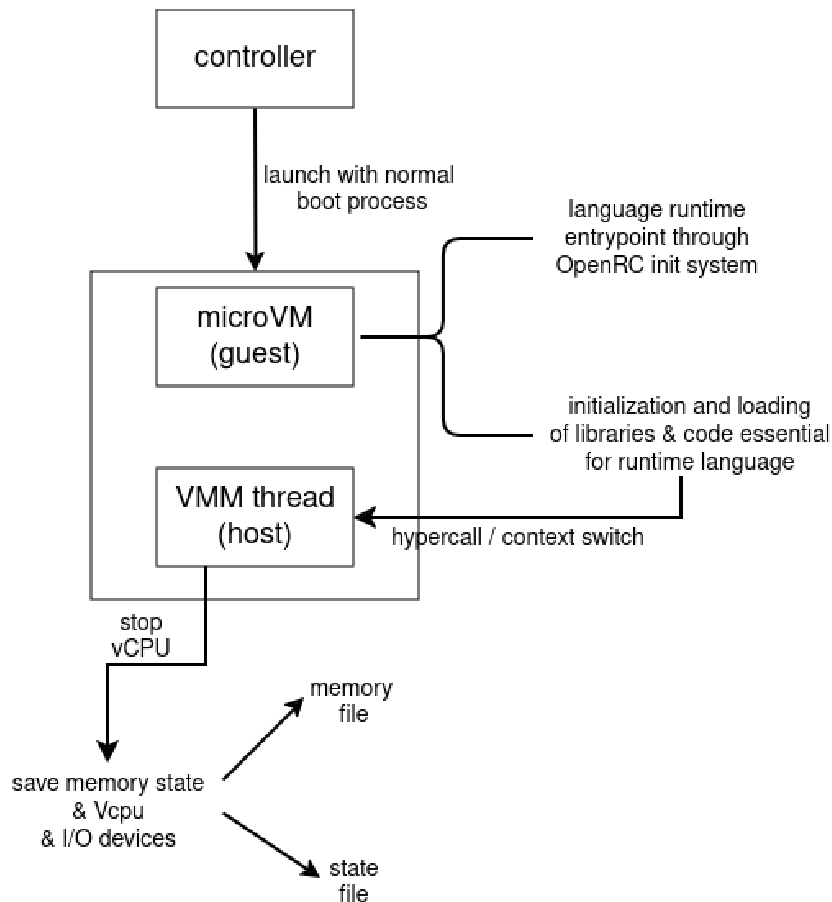
Το επίπεδο μεταφοράς το οποίο ορίζει έναν γενικό τρόπο για την ανακάλυψη (discovery), διαμόρφωση (configuration) και κανονική λειτουργία (αμφίδρομη μεταφορά δεδομένων) των συσκευών. Η τελευταία γίνεται μέσω των λεγόμενων virtqueues, βασιζόμενη στην κοινή μνήμη μεταξύ guest και hypervisor. Το επίπεδο μεταφοράς, το οποίο κατ΄ αρχήν ορίζεται αφηρημένα, εξειδικεύεται με τρεις υλοποιήσεις: δίαυλο (bus) PCI, memory-mapped I/O (MMIO) και I/O channel.

Το σύνολο των συσκευών που χτίζεται πάνω από το επίπεδο μεταφοράς, καθορίζοντας για κάθε συσκευή τα διαθέσιμα πεδία διαμόρφωσης (configuration fields) της, το πλήθος των virtqueues που χρησιμοποιεί και τα μηνύματα που μεταφέρονται μέσω αυτών για τη λειτουργία της.

3.1.2 Λήψη στιγμιοτύπων

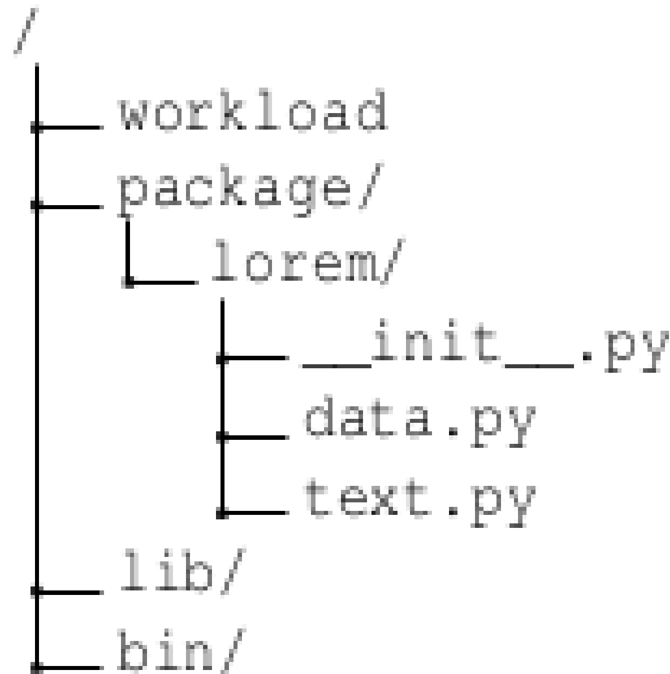
Ιδιαίτερη αναφορά θα πρέπει να γίνει στον τρόπο με τον οποίο υλοποιείται η λήψη των εκάστοτε στιγμιότυπων, εφόσον είναι και ένα από τα σημεία που κάνουν το SnapFaas να ξεχωρίζει ανάμεσα από τα υπόλοιπα εργαλεία βελτιστοποίησης των cold-starts. Ο Διαχειριστής Εικονικής Μηχανής, σε συνεργασία με το προσαρμοσμένο στην γλώσσα σημείο εισόδου του κώδικα (custom language specific runtime entry point) που εκτελείται μέσα στην εικονική μηχανή, “αιχμαλωτίζει” την κατάσταση εκτέλεσης του VM. Το custom language specific runtime entry point είναι ένα script που όταν φτάσει στο σημείο να εκτελεστεί (μέσα από το OpenRC init system), αρχικοποιεί και φορτώνει τις βασικές βιβλιοθήκες και τον ίδιο τον κώδικα που απαιτούνται για την λειτουργία της εκάστοτε γλώσσας εκτέλεσης. Επιπλέον, φορτώνει το workload της εφαρμογής (το οποίο περιλαμβάνει, μεταξύ άλλων, και το AppFS) και το πιο σημαντικό, στέλνει τις κατάλληλες εντολές (hypercalls) ώστε να σταματήσει η λειτουργία της εικονικής μηχανής και να προκληθεί μια αλλαγή περιβάλλοντος (context switch) από τον κώδικα που εκτελείται μέσα στην εικονική μηχανή στον φιλοξενούμενο (guest), στον Διαχειριστή Εικονικής Μηχανής που τρέχει στον οικοδεσπότη (host). Με αυτόν τον τρόπο ξεκινάει η διαδικασία λήψης ενός στιγμιότυπου.
Στο πλαίσιο των λειτουργικών συστημάτων, ένα hypercall αναφέρεται στον μηχανισμό virtualized περιβάλλοντων, όπου το λειτουργικό σύστημα του guest, μέσα σε μια εικονική μηχανή (στην προκειμένη περίπτωση μέσα στο microVM), επικοινωνεί με τον hypervisor (στην προκειμένη περίπτωση το KVM), για να εκτελέσει προνομιακές λειτουργίες (privileged operations) ή να αιτηθεί λειτουργίες που δεν εμπίπτουν στο πεδίο εφαρμογής του Guest OS. Σε αντίθεση με μια κανονική κλήση συστήματος, η οποία περιλαμβάνει την μετάβαση απο το user mode σε kernel mode μέσα στο ίδιο λειτουργικό σύστημα, ένα hypercall περιλαμβάνει την μετάβαση από το context του λειτουργικού συστήματος του guest, στο context του hypervisor που εδρεύει στον host. Τα hypercalls επιτρέπουν την διαχείριση της μνήμης, τον προγραμματισμό της CPU και την επικοινωνία μεταξύ VMs στο ίδιο σύστημα.

Για την δημιουργία ενός στιγμιότυπου βάσης, η εικονική μηχανή εκκινείται κανονικά από τον πυρήνα χρησιμοποιώντας το RootFS σαν παρεχόμενο σύστημα αρχείου σε συνδυασμό με ένα placeholder σύστημα αρχείου εφαρμογής (AppFS) που θα χρησιμοποιηθεί μετέπειτα. Μόλις ολοκληρωθεί η αρχικοποίηση όλων των απαραίτητων εξαρτήσεων και βιβλιοθηκών για την λειτουργία της γλώσσας εκτέλεσης, το language runtime entry point εκτελεί ένα hypercall ώστε να προκληθεί το context switch που αναφέραμε. Με αυτόν τον τρόπο προκαλείται το σταμάτημα του εικονικού επεξεργαστή (vCPU) της εικονικής μηχανής και πυροδοτείται η εκκίνηση της διαδικασίας για την “αιχμαλώτιση” της κατάστασης εκτέλεσης. Αυτή περιλαμβάνει, μεταξύ άλλων, την κατάσταση της μνήμης, του επεξεργαστή και των συσκευών εισόδου/εξόδου. Σαν αποτέλεσμα της παραπάνω διαδικασίας, παράγεται το στιγμιότυπο βάσης το οποίο αποτελείται από ένα αρχείο μνήμης που περιέχει ολόκληρη την διάταξη των σελίδων μνήμης του microVM, καθώς και ένα JSON αρχείο που περιέχει την κατάσταση του επεξεργαστή και των Input/Output συσκευών. Στο Σχήμα 11 παρουσιάζεται η διαδικασία λήψης ενός base snapshot.



Σχήμα 11: Διαδικασία λήψης base snaphot

Για την δημιουργία ενός diff snapshot για κάποια συγκεκριμένη συνάρτηση, εκκινούμε την εικονική μηχανή μέσω της αποκατάστασης του σωστού στιγμιότυπου βάσης, με το σύστημα αρχείου της εφαρμογής (AppFS) φορτωμένο σε αυτήν. Ως αποτέλεσμα, το VM συνεχίζει την εκτέλεση του αμέσως μετά την εντολή που προκάλεσε το context switch και οδήγησε στην παύση της εικονικής μηχανής. Η αμέσως επόμενη εντολή στο κώδικα εκτέλεσης της γλώσσας (language runtime) είναι η φόρτωση των απαραίτητων για την εφαρμογή βιβλιοθηκών. Σε μια γλώσσα σαν την Python (που είναι και η γλώσσα εκτέλεσης που έχουμε χρησιμοποιήσει στα πειράματά μας), αυτό επιτυγχάνεται μέσω της εισαγωγής του module workload που περιέχει τον κώδικα εκτέλεσης της εφαρμογής καθώς και όλες τις εξαρτώμενες βιβλιοθήκες και εκτελέσιμα που απαιτούνται. Η δομή του application module παρουσιάζεται στο Σχήμα 12.



Σχήμα 12: Η tarball δομή που περιέχει το workload της εφαρμογής

Όταν η συνάρτηση αρχικοποιηθεί – και πριν εκτελεστεί ο κώδικας της εφαρμογής – η γλώσσα εκτέλεσης πραγματοποιεί και πάλι ένα hypercall όπως και στην περίπτωση του στιγμιότυπου βάσης, δίνοντας έτσι την εντολή στο VMM να παράξει και πάλι στιγμιότυπο, το snapshot της συνάρτησης αυτή τη φορά. Η συνέχιση της εκτέλεσης μετά την αποκατάσταση του στιγμιότυπου βάσης, προϋποθέτει την ενεργοποίηση της ανίχνευσης των "βρώμικων" σελίδων μνήμης (dirty page tracking), εφόσον τα στιγμιότυπα διαφοράς αποτελούνται αποκλειστικά από τις σελίδες αυτές οι οποίες έχουν τροποποιηθεί από την στιγμή της συνέχισης εκτέλεσης απο το base snapshot μέχρι και την στιγμή λήψης του diff snapshot. Το diff στιγμιότυπο περιέχει ακόμη τις σελίδες μνήμης που έχουν τροποποιηθεί σε σχέση με τις σελίδες μνήμης του στιγμιότυπου βάσης. Όταν θέλουμε να επαναφέρουμε την εικονική μηχανή χρησιμοποιώντας και τα δύο snapshots (base & diff), αυτή η πληροφορία θα αξιοποιηθεί. Για την υπόλοιπη κατάσταση (πλην της μνήμης) που θα πρέπει να αποθηκευτεί χρησιμοποιείται και πάλι ένα JSON αρχείο.

Για την εύρεση του ενεργού συνόλου σελίδων (working set), ξεκινάμε την διαδικασία εκκίνησης της εικονικής μηχανής μέσω του restoration των base και diff στιγμιότυπων. Στο σημείο αυτό η εικονική μηχανή είναι έτοιμη να εκτελέσει τον κώδικα της εφαρμογής/συνάρτησης. Αφού έχει ολοκληρωθεί η εκτέλεση, καταγράφουμε το σετ των σελίδων στις οποίες γίνεται πρόσβαση και έχουν αποκατασταθεί απο το στιγμιότυπο διαφοράς, φτιάχνοντας με αυτόν τον τρόπο το working set.

3.2 Τεχνική υλοποίηση

Σε μια πιο τεχνική ματιά, όταν ο client του SnapFaas (ο wrapper μέσω του οποίου διαχειριζόμαστε το εργαλείο Firecracker) καλέσει το εκτελέσιμο Firecracker και αφού γίνουν οι κατάλληλες αρχικοποιήσεις, καλείται η συνάρτηση start_vmm_thread(), μέσω της οποίας ξεκινάμε ένα καινούργιο νήμα (thread) VMM που μπορεί να εξυπηρετήσει API αιτήματα. Αρχικά, φτιάχνουμε ένα καινούργιο Vmm struct με βάση το configuration που έχουμε περάσει μέσω του singlevm εκτελέσιμου του SnapFaas. Το struct αυτό αποτελείται από διάφορα πεδία, όπως το περιβάλλον KVM (KVM context – ένας KVM pointer που συνδέεται στο στιγμιότυπο του microVM). Μέσω αυτού αποκτούμε πρόσβαση στις λειτουργίες του KVM, εφόσον υπάρχει η εκάστοτε KVM λειτουργία στο host μηχάνημα. Ένα επιπλέον πεδίο είναι η δομή Vm που περιέχει τον file descriptor που προκύπτει απο την KVM_CREATE_VM κλήση, τον χειριστή διακοπων και την κύρια μνήμη της εικονικής μηχανής. Άλλα πεδία είναι η μνήμη της εικονικής μηχανής, το configuration του πυρήνα (τα οποία δεν έχουν αρχικοποιηθεί ακόμα), διάφοροι χειριστές συσκευών (π.χ. Legacy Device Manager), καθώς και τα διάφορα στιγμιότυπα εάν έχουμε δώσει εντολή για αποκατάσταση της εικονικής μηχανής μέσω αυτών. Στο Σχήμα 13 παραθέτουμε την δομή όπως αυτή υπάρχει στον πηγαίο κώδικα του τροποποιημένου Firecracker.

```
struct Vmm {
    kvm: KvmContext,
    vm_config: VmConfig,
    shared_info: Arc<RwLock<InstanceInfo>>,
    stdin_handle: io::Stdin,
    // Guest VM core resources.
    guest_memory: Option<GuestMemory>,
    kernel_config: Option<KernelConfig>,
    vcpus_handles: Vec<thread::JoinHandle<()>>,
    exit_evt: Option<EpollEvent<EventFd>>,
    vm: Vm,
    // Guest VM devices.
    mmio_device_manager: Option<MMIODeviceManager>,
    legacy_device_manager: LegacyDeviceManager,
    drive_handler_id_map: HashMap<String, usize>,
    net_handler_id_map: HashMap<String, usize>,
    // Device configurations.
    // If there is a Root Block Device, this should be added as the first element of the list.
    // This is necessary because we want the root to always be mounted on /dev/vda.
    block_device_configs: BlockDeviceConfigs,
    network_interface_configs: NetworkInterfaceConfigs,
    // #[cfg(feature = "vsock")]
    vsock_device_configs: VsockDeviceConfigs,
    epoll_context: EpollContext,
    // API resources.
    api_event: EpollEvent<EventFd>,
    from_api: Receiver<Box<VmmAction>>,
    write_metrics_event: EpollEvent<TimerFd>,
    // The level of seccomp filtering used. Seccomp filters are loaded before executing guest code.
    seccomp_level: u32,
    // Snapshot
    // load
    load_dir: Vec<PathBuf>,
    snap_to_load: Option<Snapshot>,
    // dump
    dump_dir: Option<PathBuf>,
    snap_to_dump: Option<Snapshot>,
    snap_receiver: Option<Receiver<VcpuInfo>>,
    snap_sender: Option<Sender<VcpuInfo>>,
    snap_evt: EventFd,
    // restore memory by copying
    base: MemoryFileOption,
    huge_page: bool,
    //diff_dirs: Vec<PathBuf>,
    diff: MemoryFileOption,
    load_ws: bool,
}
```

Σχήμα 13: Τα πεδία του Vmm struct

Στην συνέχεια (μέσα στην συνάρτηση start_vmm_thread()), καλούμε την συνάρτηση run_control(), στην οποία υπάρχει βρόχος (loop) στον οποίο ελέγχουμε διαρκώς για εισερχόμενα συμβάντα (events), όπως το VmmActionRequest που περιλαμβάνει τις ενέργειες που μπορούμε να στείλουμε απο τον SnapFaas client σχετικά με την διαχείριση της εικονικής μηχανής. Ένα από αυτά τα events είναι το Snap event, το οποίο στέλνεται μέσα από την εξομοίωση (emulation) του εικονικού επεξεργαστή όταν λάβει το κατάλληλο σήμα από τον κώδικα εκτέλεσης, είτε της γλώσσας (όταν πρόκειται για στιγμιότυπο βάσης) είτε της συνάρτησης (όταν πρόκειται για στιγμιότυπο διαφοράς). Σε αυτό το σημείο είμαστε έτοιμοι να αποθηκεύσουμε, αρχικά, την κατάσταση του επεξεργαστή (Vcpu state) και στην συνέχεια να “αιχμαλωτισουμε” τις εκάστοτε καταστάσεις των συσκευών καθώς και την μνήμη.

3.2.1 Αποθήκευση μνήμης

Διαβάζοντας το αρχείο /proc/<pid_of_firecracker>/pagemap του host συστήματος, παίρνουμε μια αντιστοίχιση μεταξύ των φυσικών σελίδων του host και του guest καθώς και την λίστα από την αρίθμηση των φυσικών σελίδων του guest που έχουν τροποποιηθεί (dirty guest physical page numbers) [Σχήμα 14 – στο συγκεκριμένο παράδειγμα η εικονική διεύθυνση 0x400000 του guest, αντιστοιχίζεται στην φυσική διεύθυνση 0x12845d000 του host]. Στην περίπτωση μας, αιτούμαστε αντιστοίχιση των σελίδων απο τον guest στον host. Ως αποτέλεσμα, έχουμε το πλήθος των σελίδων που έχουν προσπελαστεί (accessed), αυτών που έχουν τροποποιηθεί (dirty) καθώς και αυτών που έχουν μόνο διαβαστεί (read-only), μέχρι αυτό το σημείο που η εφαρμογη μας αιτήθηκε την λήψη του στιγμιότυπου. Με την βοήθεια των παραπάνω σετ σελίδων παράγονται τα εξής αρχεία:

→ **memory_dump:** το εύρος των σελίδων οι οποίες έχουν τροποποιηθεί κατά την εκτέλεση του VM μέχρι και την λήψη του στιγμιότυπου (dirty regions). "Μεταφέρονται" από την μνήμη στο συγκεκριμένο αρχείο μέσω της συνάρτησης write_from_memory() (writes data from memory to a writable object).

→ **memory_dump_sparse:** το μέγεθος του αρχείου αυτού είναι το ίδιο με το μέγεθος της μνήμης του VM, καθώς ολόκληρη η RAM (Random access memory) γράφεται στο συγκεκριμένο αρχείο. Στην συνέχεια, είναι απαραίτητη η δημιουργία τρυπών (punching holes), μέσω της συνάρτησης fallocate() η οποία αποδεσμεύει χώρο σε ένα αρχείο. Μέσα στο εύρος μνήμης που αποφασίζουμε ότι δεν μας είναι χρήσιμο (είτε ότι πρέπει να είναι δεσμευμένο για διαφορετικό σκοπό), το συγκεκριμένο μπλοκ του αρχείου γεμίζει με μηδενικά. Το μέγεθος του αρχείου δεν μικραίνει, καθώς περνάμε το flag FALLOC_FL_KEEP_SIZE στην συνάρτηση fallocate.

Η προαναφερόμενη διαδικασία δημιουργίας των αρχείων μνήμης ενός στιγμιότυπου εκτελείται αυτούσια, είτε πρόκειται για base snapshot είτε για diff. Η διαφορά τους έγκειται στο πότε καλείται η πυροδότηση του εκάστοτε snapshot με αποτέλεσμα την αποθήκευση μνήμης που περιλαμβάνει τις εκάστοτε αρχικοποιήσεις που απαιτούνται από τα δύο είδη στιγμιότυπων.

Αφού ολοκληρωθεί η εγγραφή των αρχείων μνήμης, είναι απαραίτητο να κρατήσουμε κάποιες πληροφορίες, σχετικά με τις θέσεις των σελίδων μνήμης και το εύρος για το οποίο μας ενδιαφέρει σε κάθε περίπτωση, σαν μεταδεδομένα. Για το στιγμιότυπο γλώσσας (base) ένα νέο στρώμα μεταδεδομένων μνήμης (struct memory snapshot layer – μια δομή που περιγράφει ένα μονό στρώμα στιγμιότυπου μνήμης) δημιουργείται, ενώ για το στιγμιότυπο συνάρτησης (diff) επικαιροποιείται το υπάρχον στρώμα μνήμης. Η δομή MemorySnapshotLayer αποτελείται από το πεδίο dirty_regions που περιλαμβάνει τις τροποποιημένες σελίδες της μνήμης και από αυτό το σύνολο, αυτές στις οποίες έχει γίνει πρόσβαση αποτελούν το πεδίο working_set (το οποίο είναι κενό στην περίπτωση του στιγμιότυπου γλώσσας). Για την επικαιροποίηση (update), που εκτελείται όταν δημιουργούμε ένα στιγμιότυπο συνάρτησης (diff snapshot), όλες οι σελίδες από το εύρος μνήμης που έχουν τροποποιηθεί (dirty regions), εισάγονται σε ένα ταξινομημένο σύνολο που βασίζεται σε ένα B-Tree [19]. Με βάση το καινούργιο αυτό δέντρο βρίσκονται οι διαφορές σε σχέση με το υπάρχων (που προκύπτει από το B-Tree του υπάρχοντος base memory layer, δηλαδή οι διαφορές μεταξύ των δύο layers – base και diff) και προστίθεται σαν καινούργιο στρώμα στον πίνακα MemorySnapshotLayer. Τα μεταδεδομένα για το κάθε στρώμα μνήμης αποθηκεύονται στο JSON αρχείο που περιλαμβάνει την κατάσταση του

VM και χρησιμοποιουνται για την εύρεση του working set καθώς και για την αποκατάσταση της μνήμης που θα εξετάσουμε στην συνέχεια.

```
addr pfn soft-dirty file/shared swapped present library
400000 12845d 0 1 0 1 /bin/bash
401000 12845e 0 1 0 1 /bin/bash
402000 12845f 0 1 0 1 /bin/bash
```

Σχήμα 14: Το περιεχόμενο ενός pagemap αρχείου μιας τυχαίας διεργασίας

Η εύρεση του working set πυροδοτείται. σε αντίθεση με τα δύο στιγμιότυπα που η λήψη τους γίνεται trigger μέσα από την προσομοίωση του Vcpu στον VMM του Firecracker, από τον wrapper του SnapFaas. Πιο συγκεκριμένα, η ενέργεια dump_working_set του Διαχειριστή Εικονικής Μνήμης (VmmAction) πυροδοτείται από τον client του SnapFaas – απο το εκτελέσιμο singlevm μέσα από το οποίο αποστέλλονται όλες οι ενέργειες για την εκκίνηση μιας εικονικής μηχανής και είναι υπεύθυνο για να σταλούν τα αιτήματα (requests) σε αυτή. Αφού πρώτα εξυπηρετηθεί ένα request από μία συνάρτηση, το singlevm αναλαμβάνει να στείλει μέσω μιας καινούργιας σύνδεσης unix socket (host) - virtual socket (guest), το VmmAction που θα πυροδοτήσει την εύρεση του working set (dump_working_set). Ο πηγαίος κώδικας παρουσιάζεται στο Σχήμα 15. Για να βρούμε το ενεργό σύνολο του στιγμιότυπου διαφοράς (diff snapshot), πρέπει να έχουμε επαναφέρει το VM με το στιγμιότυπο βάσης σε συνδυασμό με το στιγμιότυπο διαφοράς της συνάρτησης για την οποία θέλουμε να βρούμε το working set. Εκμεταλλευόμαστε και πάλι την συνάρτηση get_pagemap() και παίρνουμε την αντιστοίχιση μεταξύ των φυσικών σελίδων του guest και του host, που έχουν προσπελαστεί απο την διεργασία του Virtual Machine Manager. Η τομή των σελίδων αυτών και των σελίδων που έχουν τροποποιηθεί και περιλαμβάνονται στο στιγμιότυπο βάσης, αποτελούν το ws_page_set. Το σετ αυτό αποθηκεύεται στα μεταδεδομένα του MemorySnapshotLayer και αποτελείται από regions μνήμης. Κάθε region μνήμης που περιλαμβάνεται στο ws_page_set αλλά και στο αρχείο memory_dump_sparse (η μνήμη του στιγμιότυπου συνάρτησης) προστίθεται στο τελικό ενεργό σύνολο σελίδων της επικείμενης συνάρτησης.

```rust
// Synchronously send the request to vm and wait for a response
let dump_working_set = true && cmd_arguments.is_present("dump working set");
for req in requests {
    let t1 = Instant::now();
    log::debug!("request: {:?}", req);
    match vm.process_req(req) {
        Ok(rsp) => {
            let t2 = Instant::now();
            println!("request returned in: {} us", t2.duration_since(t1).as_micros());
            log::debug!("response: {:?}", rsp);
            println!("response: {:?}", rsp);
            num_rsp+=1;
        }
        Err(e) => {
            eprintln!("Request failed due to: {:?}", e);
        }
    }
    if dump_working_set {
        let listener_port = format!("dump_ws-{}.sock", id);
        UnixStream::connect(listener_port).expect("Failed to connect to VMM UNIX listener");
        let port = format!("dump_ws-{}.sock.back", id);
        let li = UnixListener::bind(port).expect("Failed to listen at the port");
        li.accept().expect("Failed to accept a connection");
        break;
    }
}
```

Σχήμα 15: Ο τρόπος που στέλνονται τα αιτήματα και η πυροδότηση της εύρεσης του working set, απο το singlevm εκτελέσιμο του SnapFaas

3.2.2 Επαναφορά μνήμης

Η χρήση στιγμιότυπων βελτιστοποιεί το cold-start μέσω της αρχικοποίησης της αποθηκευμένης κατάστασης της μνήμης από προηγούμενες εκτελέσεις, με αποτέλεσμα η εικονική μηχανή να συνεχίζει την εκτέλεση της απο το σημείο που είχε σταματήσει και έτσι να παραλείπονται χρονοβόρες διαδικασίες όπως το booting phase, τα init scripts του λειτουργικού συστήματος καθώς και οι αρχικοποιήσεις της γλώσσας εκτέλεσης και των συναρτήσεων. Παρόλα αυτά η επαναφορά μιας απομνημονευμένης κατάστασης δεν μπορεί να συμβεί στιγμιαία καθώς η αντιγραφή των αποθηκευμένων δεδομένων απαιτεί επίσης χρόνο. Για την αποκατάσταση ενός στιγμιότυπου γλώσσας ή και συνάρτησης, στην δημιουργία μνήμης της εικονικής μηχανής (guest memory), ακολουθείται διαφορετική διαδικασία σε σχέση με την περίπτωση της κανονικής εκκίνησης και διαφοροποιείται ανάλογα του τρόπου αποκατάστασης που επιλέγεται στην εκάστοτε περίπτωση.

Η πρώτη υλοποίηση που έχει υιοθετηθεί απο το SnapFaas είναι η αποκατάσταση “κατά απαίτηση” (on-demand/lazy) όπως αποκαλείται. Σε αυτήν την περίπτωση η εφαρμογή μας όταν προσπαθεί να αποκτήσει πρόσβαση σε κάποια σελίδα μνήμης η οποία δεν υπάρχει στην μνήμη της διεργασίας, ο hypervisor (KVM) φορτώνει την σελίδα, σύγχρονα απο το file-mmaped αρχείο μνήμης *memory_dump_sparse*, στην κύρια μνήμη του microVM. Σαν αποτέλεσμα, μόνο χρήσιμες σελίδες φορτώνονται και η εφαρμογή μπορεί να ξεκινήσει να εκτελείται απευθείας. Ο συγκεκριμένος τρόπος αποκατάστασης αποφεύγει την προ-ανάκτηση ολόκληρης της κατάστασης της εικονικής μηχανής αλλά “πληρώνει” τα σύγχρονα page fault traps κατα την διάρκεια της εκτέλεσης, αφού τότε πρέπει να κάνει fetch την αντίστοιχη σελίδα από τον δίσκο. Ο σύγχρονος τρόπος που φορτώνονται οι σελίδες έχει σαν απόρροια το μπλοκάρισμα της συνάρτησης κάθε φορά που μία σελίδα μνήμης ανακτάται στην κύρια μνήμη, με αποτέλεσμα η καθυστέρηση της “κρύας”

εκτέλεσης να μετατίθεται στην εκτέλεση της συνάρτησης αυτής καθεαυτής, όταν επιλέγεται καθολικά ο on-demand τρόπος αποκατάστασης της μνήμης.

Για να επιτευχθεί η κατά απαίτηση φόρτωση της μνήμης στην μικρο-εικονική μηχανή, ο Διαχειριστής Εικονικής Μηχανής (VMM) χαρτογραφεί στην ιδιωτική μνήμη του VM το αρχείο μνήμης του στιγμιότυπου, μέσω της συνάρτησης mmap. Ένα memory mapped file όπως αποκαλείται το αποτέλεσμα της παραπάνω διαδικασίας, είναι ένα τμήμα εικονικής μνήμης στο οποίο έχει ανατεθεί μια απευθείας byte προς byte αντιστοίχιση με μια πηγή αρχείου. Η πηγή αυτή μπορεί να είναι ένα αρχείο φυσικά παρόν στον δίσκο, ένα αντικείμενο κοινής μνήμης ή οποιαδήποτε άλλη πηγή στην οποία το λειτουργικό σύστημα μπορεί να αναφερθεί μέσω ενός περιγραφέα αρχείου (file descriptor). Απο την στιγμή που το αρχείο χαρτογραφηθεί στην μνήμη, η εφαρμογή μας μέσα στην εικονική μηχανή χρησιμοποιεί το χαρτογραφημένο αυτό τμήμα σαν την κύρια μνήμη του. Με την χρήση του mmap, γίνονται page-in αποκλειστικά οι σελίδες μνήμης στις οποίες γίνεται πρόσβαση και είναι ένας τρόπος να αποφεύγεται η ολική προ-ανάκτηση από τον δίσκο της μνήμης του VM, που στις περισσότερες των περιπτώσεων είναι αργό.

Η δεύτερη υλοποίηση που έχει υιοθετηθεί απο το εργαλείο μας είναι η “άμεση” αποκατάσταση μνήμης (eager memory restoration). Σε αυτήν την περίπτωση χρησιμοποιείται ένα άλλο αρχείο μνήμης που περιλαμβάνεται στο στιγμιότυπο, ονόματι memory_dump το οποίο περιέχει τις “βρώμικες” σελίδες μνήμης (dirty memory pages), δηλαδή αυτές τις οποίες έχει τροποποιήσει η Function-as-a-Service εφαρμογή μας. Το αρχείο διαβάζεται και κάθε dirty περιοχή μνήμης αντιγράφεται στην μνήμη του VM για το εκάστοτε στρώμα μνήμης για το οποίο έχει επιλεχθεί η συγκεκριμένη τεχνική αποκατάστασης. Για την αντιγραφή των σελίδων μνήμης χρησιμοποιείται η κλήση συστήματος read_vectored(), που διαβάζει από το memory_dump αρχείο τα τροποποιημένα κομμάτια μνήμης και τα αντιγράφει σε "scatter input" buffers [21], όπως φαίνεται και μέσα απο την συνάρτηση *copy_load_memory()* που παρουσιάζεται στο Σχήμα 16. Η “άμεση” αποκατάσταση μνήμης φορτώνει τις σελίδες μνήμης από το δίσκο προτού εκτελεστεί η συνάρτηση, σε παρτίδα (in batch). Η διαδικασία αυτή καθυστερεί την έναρξη της εφαρμογής μας και μπορεί να φορτώσει σελίδες που δεν θα χρησιμοποιηθούν ποτέ, αλλά η ταχύτητα εκτέλεσης είναι ίδια με την ταχύτητα εύρους ζώνης του μέσου αποθήκευσης (storage medium bandwidth speed). Αναμένουμε λοιπόν ταχύτερους χρόνους όσον αφορά την εκτέλεση της εφαρμογής μας αυτης καθ’ αυτής, στο πειραματικό σκέλος.

```
fn copy_load_memory(&self, memory_dump: &mut File, dirty_regions: &RegionList) -> Result<()> {
    let mut i: usize = 0;
    let mut bufs: Vec<IoSliceMut<'_>> = Vec::new();
    // iterate through the regions for the current layer
    self.with_regions_mut(cb: |_, guest_base: GuestAddress, size: usize, ptr: usize| -> Result<()> {

        while i < dirty_regions.len() && (dirty_regions[i].0 < guest_base.offset() + size) {
            // println!("VMM: copying region at {}:{}", dirty_regions[i].0, dirty_regions[i].1);
            let guest_addr: usize = dirty_regions[i].0 * PAGE_SIZE;
            let region_len: usize = dirty_regions[i].1 * PAGE_SIZE;

            let addr: usize = ptr + guest_addr;
            // let addr = ptr + (guest_addr - guest_base.offset());
            let buf: &mut [u8] = unsafe { std::slice::from_raw_parts_mut(data: addr as *mut u8, region_len) };
            bufs.push(std::io::IoSliceMut::new(buf));
            i += 1;
        }
        Ok(())
    })?;
    // getter-scatter read done here
    memory_dump.read_vectored(bufs: bufs.as_mut_slice()).map(op: |_| ()).map_err(op: |e: Error| Error::IoError(e))
}
```

Σχήμα 16: Η συνάρτηση copy_load_memory() του VMM, υπεύθυνη για το eager restoration είτε της μνήμης του στιγμιότυπου συνάρτησης είτε του working set

Για να επανέλθουμε στην υλοποίηση του εργαλείου SnapFaas, όταν επιλέγουμε να εκκινήσουμε το Virtual Machine μας μέσω της χρήσης στιγμιότυπου, ο Διαχειριστής Εικονικής Μηχανής κάνει file-mmap το αρχείο μνήμης memory_dump_sparse του στιγμιότυπου βάσης στην ιδιωτική μνήμη της εικονικής μηχανής. Σαν αποτέλεσμα, οι σελίδες μνήμης από το base snapshot φορτώνονται, όταν γίνει πρόσβαση σε αυτές, μέσω της τεχνικής διαχείρισης πόρων Copy-on-Write (CoW). Η Copy-on-Write τεχνική χρησιμοποιεί το page table [20] για να "μαρκάρει" συγκεκριμένες σελίδες της μνήμης ως read-only και να κρατήσει το πλήθος των αναφορών σε καθεμία. Όταν γράφονται δεδομένα στις σελίδες αυτές ο πυρήνας του λειτουργικού συστήματος διακόπτει την προσπάθεια γραψίματος και εκχωρεί μια νέα φυσική σελίδα, αρχικοποιημένη με τα CoW data. Στη συνέχεια, ο πυρήνας ενημερώνει τον πίνακα σελίδων με τη νέα (εγγράψιμη) σελίδα, μειώνει τον αριθμό των αναφορών σε αυτή και εκτελεί την εγγραφή. Οι σελίδες λοιπόν που γίνονται fetch lazily μέσω του mmaped αρχείου και δεν τροποποιούνται από την εκτέλεση της συνάρτησης (καθώς περιλαμβάνονται στο στιγμιότυπο γλώσσας), δια-μοιράζονται μεταξύ των εικονικών μηχανών που χρησιμοποιούν την ίδια γλώσσα εκτέλεσης, μέσω του base snapshot. Στην συνέχεια, το VMM αντιγράφει κάθε σελίδα από το diff snapshot στην μνήμη του microVM χρησιμοποιώντας, όπως αναφέραμε και προηγουμένως, την κλήση συστήματος readv().

Στο δικό μας πειραματικό περιβάλλον (που θα αναλύσουμε εκτενώς στο 4ο κεφάλαιο), καταλήξαμε – μετά από την εκτέλεση όλων των δυνατών συνδυασμών φόρτωσης στιγμιότυπων – ότι το πιο επικερδές χρονικά στην διαδικασία της "κρύας" εκκίνησης είναι η φόρτωση κατά απαίτηση (on-demand) των base και diff στιγμιότυπων και η άμεση αντιγραφή του working set στην μνήμη της εικονικής μηχανής (μεσω του eager restoration που περιγράψαμε παραπάνω).

3.2.3 Αποθήκευση κατάστασης Virtio συσκευών και Εικονικού Επεξεργαστή

Όταν δοθεί η εντολή, μέσω του runtime workload, για την λήψη ενός στιγμιότυπου πέρα από την "αιχμαλώτιση" της μνήμης που περιγράψαμε, είναι απαραίτητη η αποθήκευση της κατάστασης του vCPU καθώς και των devices που είναι συνδεδεμένες στο microVM. Για την κατάσταση του εικονικού επεξεργαστή (vCPU), αποθηκεύουμε την κατάσταση του επεξεργαστή πολλαπλών καταστάσεων (multistate processor), την κατάσταση των κεντρικών καταχωρητών (core registers – 31 στο πλήθος και οι οποίοι είναι οι καταχωρητές γενικού σκοπού ενός ARM επεξεργαστή), καθώς και την κατάσταση των καταχωρητών συστήματος (system registers – καλούμε την KVM_GET_REG_LIST, μέσω της κλήσης συστήματος *ioctl()* [control device system call [14]], για να πάρουμε όλους τους υπόλοιπους διαθέσιμους καταχωρητές του επεξεργαστή). Τέλος, αποθηκεύουμε τον Multiprocessor Affinity Register (MPIDR) [3], που είναι απαραίτητος για την αποθήκευση της κατάστασης του Interrupt Controller, όπως θα αναλύσουμε παρακάτω (Κεφάλαιο 3.2.5). Η συνάρτηση *dump_vcpu_state()* παρουσιάζεται στο Σχήμα 17.

```
pub fn dump_vcpu_state(&self) -> Result<VcpuState> {
    // Err(Error::UnsupportedAction("Saving the state"))
    let mut state: VcpuState = VcpuState::default();

    // Get this vCPUs multiprocessing state.
    state.mp_state = arch::regs::get_mpstate(vcpu: &self.fd).map_err(op: Error::SaveState)?;

    arch::regs::save_core_registers(vcpu: &self.fd, state: &mut state.regs).map_err(op: Error::SaveState)?;

    arch::regs::save_system_registers(vcpu: &self.fd, state: &mut state.regs).map_err(op: Error::SaveState)?;

    state.mpidr = arch::aarch64::regs::read_mpidr(vcpu: &self.fd).map_err(op: Error::SaveState)?;

    Ok(state)
}
```

Σχήμα 17: Η συνάρτηση dump_vcpu_state()

Αναφορικά με την αποθήκευση των devices, για κάθε συσκευή μπλοκ, δικτύου και το virtual socket, αποθηκεύουμε τις εικονικές ουρές (virtqueues) [18] των συσκευών αυτών, οι οποίες περιέχουν διάφορα δεδομένα όπως το μέγεθος της ουράς, την θέση της εκάστοτε συσκευής μέσα στον πίνακα περιγραφής των συσκευών (description table), την αμέσως επόμενη διαθέσιμη θέση για γράψιμο/διάβασμα δεδομένων μέσα στην ουρά, κλπ. Τα JSON δεδομένα σχετικά με την κατάσταση της εκάστοτε συσκευής, είναι της παρακάτω μορφής:

"block_states":{"queues": [{"max_size":256,"size":256,"ready":true,"desc_table":60768256,"avail_ring":60772352,"used_ring":60776448,"next_avail":936,"next_used":936}]}

3.2.4 Επαναφορά κατάστασης virtio συσκευών και Εικονικού Επεξεργαστή

Στην φάση εκκίνησης της εικονικής μηχανής, μέσα στον Διαχειριστή Εικονικής Μηχανής (VMM) και πιο συγκεκριμένα μέσα στην συνάρτηση *start_microvm() και* αφού έχει

αρχικοποιηθεί η guest memory του microVM, καλείται η συνάρτηση *create_vcpus()* η οποία χρησιμοποιεί την KVM_CREATE_VCPU κλήση στο KVM για να αρχικοποιήσει έναν ARM specific εικονικό επεξεργαστή για την εκκίνηση ενός Linux συστήματος. Έπειτα και αναλόγως αν έχει ζητηθεί απο τον Firecracker επαναφορά ενός microVM μέσα από κάποιο στιγμιότυπο, καλείται η συνάρτηση *load_vcpu_state()*, η οποία θέτει τους εκάστοτε registers του επεξεργαστή με βάση το αποθηκευμένο state του snapshot. Σε διαφορετική περίπτωση οι καταχωρητές "σεττάρονται" για την κανονική εκκίνηση του εικονικού επεξεργαστή.

Για την επαναφορά των virtio συσκευών πρέπει αρχικά να καταχωρηθούν οι συσκευές μέσα στο memory mapped I/O, όπως θα γινόταν κανονικά και στην περίπτωση της κανονικής εκκίνησης. Στην συνέχεια, καλούνται οι συναρτήσεις *restore_block/net/vsock_device()* οι οποίες ενεργοποιούν τις εκάστοτε συσκευές με την χειροκίνητη εγγραφή των κατάλληλων δεδομένων ("manually replay writes") σε αυτή. Παραδείγματος χάριν, γράφοντας το μηδέν (0x0) στον register update_driver_status (0x70), πυροδοτείται ένα reset στην συσκευή. Αφού γίνουν οι κατάλληλες εγγραφές, η εκάστοτε συσκευή ενεργοποιείται και όλες οι ουρές πρέπει να είναι σε έγκυρη κατάσταση, που σημαίνει ότι: 1) πρέπει να έχουν γίνει marked σαν έτοιμες, 2) το μέγεθος τους δεν πρέπει να είναι μηδενικό αλλά ούτε να ξεπερνάει το μέγιστο δυνατό μέγεθος και 3) το descriptor table, το available ring και το used ring πρέπει να εδρεύουν εξ'ολοκλήρου στην μνήμη του microVM. Τότε και μόνο τότε είναι δυνατή η φόρτωση των εκάστοτε αποθηκευμένων εικονικών ουρών (virtqueues) στις υπάρχουσες μη αρχικοποιημένες, μέσω της συνάρτησης *set_queues()*.

3.2.5 General Interrupt Controller

Ένα απαραίτητο εργαλείο για την λειτουργία της εικονικής μηχανής σε ARM επεξεργαστές είναι ο Γενικός Ελεγκτής Διακοπών (Generic Interrupt Controller – GIC) ο οποίος παρέχει ένα μοντέλο χειρισμού διακοπών (interrupt handling scheme). Ο αντίστοιχος χειριστής διακοπών για x86 συστήματα – και ο οποίος είχε υλοποιηθεί για την πρωτότυπη έκδοση του SnapFaas – ονομάζεται Advanced Programmable Interrupt Controller (APIC). O Interrupt Controller είναι από τα πιο σημαντικά περιφερειακά modules ενός συστήματος, καθώς επηρεάζει άμεσα την απόδοση του vCPU. Αντιμετωπίζει όλα τα αιτήματα διακοπών και τα ανακατευθύνει στον επεξεργαστή που είναι "προσκολλημένος" στον Interrupt Controller. Μια βασική διαφορά – ανάμεσα στις 2 αρχιτεκτονικές – που προκύπτει απο την συγκεκριμένη συνθήκη (άμεση συσχέτιση Vcpu-GIC) είναι ότι στην διαδικασία εκκίνησης (booting phase) για ARM, πρέπει να δημιουργηθεί ο εικονικός επεξεργαστής μέσω του KVM (καλώντας την συνάρτηση KVM_CREATE_VCPUS, οπως αναφέρθηκε), προτού εγκατασταθεί ο Interrupt Controller, ώστε να γνωρίζει το πλήθος των εικονικών επεξεργαστών στους οποίους πρέπει να γίνει attached. Εν αντιθέσει, στην υλοποίηση του SnapFaas για x86 συστήματα, ο VMM δημιουργεί πρώτα τον Interrupt Controller για x86 συστήματα και στην συνέχεια καλεί την συνάρτηση KVM_CREATE_VCPUS.

Το irqfd είναι ένας μηχανισμός για την εισαγωγή μιας συγκεκριμένης διακοπής (interrupt) μέσα στο microVM, με την χρήση της κλήσης συστήματος eventfd. Το συγκεκριμένο system call δημιουργεί έναν περιγραφέα αρχείου που χρησιμοποιείται σαν μηχανισμός ειδοποίησης από τον πυρήνα ώστε να ειδοποιήσει (για ένα συμβάν) μια εφαρμογή στο user-space. Ο KVM λοιπόν, δημιουργεί ένα eventfd και μέσω του ioctl(), το συσχετίζει με μία συγκεκριμένη διακοπή του microVM. Γράψιμο (writes) σε αυτό το eventfd, πυροδοτεί διακοπή στον microVM. Ένα ioeventfd επιτελεί ακριβώς την αντίστροφη διαδικασία. Ο hypervisor δημιουργεί ένα eventfd και το συσχετίζει με μια συγκεκριμένη περιοχή της

μνήμης του microVM. Η εγγραφή σε αυτήν την περιοχή μνήμης από το microVM θα κάνει το eventfd αναγνώσιμο χωρίς να να είναι blocking. Αυτές οι 2 λειτουργίες (irqfd, ioeventfd) υλοποιούν είσοδο και έξοδο (input, output) σε συσκευές εικονικοποίησης, οι οποίες προκαλούν το μπλοκάρισμα του επεξεργαστή του microVM. O hypervisor ειδοποιεί το module του πυρήνα σχετικά με τα ioeventfd και τα irqfd της συσκευής καθώς και τις περιοχές μνήμης του guest που έχουν εκχωρηθεί για τις εικονικές ουρές (virtqueues) της συσκευής. Στη συνέχεια, ο πυρήνας ελέγχει για την εμφάνιση ioeventfd's, και φροντίζει για την επεξεργασία των αιτημάτων από το microVM και την πυροδότηση μιας διακοπής στον guest κατά την ολοκλήρωση του αιτήματος μέσω του irqfd.

Έπειτα, για κάθε συσκευή virtio που συνδέεται στην εικονική μηχανή (block/net/vsock device), ο Διαχειριστής Εικονικής Μηχανής (VMM) πρέπει να κάνει register το event της συσκευής, το οποίο όταν σηματοδοτηθεί θα πυροδοτήσει ένα αίτημα διακοπής (IRQ – Interrupt Request). Όταν λοιπόν η συσκευή στείλει το εξατομικευμένο της Interrupt Request, ο επεξεργαστής θα σταματήσει την λειτουργία του, ώστε η συσκευή να επιτελέσει την λειτουργία της. Τυπικά στα x86 συστήματα τα διαθέσιμα IRQs είναι 16, ενώ για ARM αρχιτεκτονικές με βάση το GIC πρωτόκολλο το πλήθος των διακοπων πρέπει να είναι πάνω από 32, κάτω από 1023 και πολλαπλάσιο του 32. Στην υλοποίηση που ακολουθήσαμε, εφαρμόσαμε 128 διαθέσιμες διακοπές. Εν κατακλείδι, μόνο ένας εικονικός Interrupt Controller μπορεί να δημιουργηθεί μέσω της KVM_CREATE_IRQCHIP κλήσης του KVM. Ο συγκεκριμένος Virtual GIC λειτουργεί ως ελεγκτής διακοπών της εικονικής μηχανής, απαιτώντας απο τις συνδεδεμένες συσκευές να εισάγουν διακοπές στον VGIC και όχι απευθείας στον Vcpu. Σημειώνουμε επίσης ότι για το deployment platform που υλοποιήσαμε στο πειραματικό σκέλος, χρησιμοποιήθηκε ο General Interrupt Controller version 2 (GICv2) [22].

3.2.5.1 Αποθήκευση και επαναφορά της κατάστασης του GIC

Στην περίπτωση λήψης ενός στιγμιότυπου, μία από τις καταστάσεις που πρέπει να αποθηκεύσουμε μέσα στο JSON αρχείο του snapshot, είναι και αυτή του Generic Interrupt Controller. Αποκτούμε πρόσβαση, μέσω του handler του Interrupt Controller, στους “Distributor Registers” καθώς και στους “CPU Interface Registers” του IRQCHIP και τους αποθηκεύουμε σε δύο Vectors u32 στοιχείων. Τα γνωρίσματα (attributes) της συσκευής για κάθε έναν από τους προαναφερόμενους τύπους καταχωρητών εμπεριέχουν την κωδικοποίηση δύο τιμών, το offset του κάθε register και τον εικονικό επεξεργαστή με τον οποίο σχετίζεται [Σχήμα 18]. Αυτοί οι δύο πίνακες αρκούν για να αποθηκεύσουν την κατάσταση του GICv2. Αντίστοιχα, όταν αποκαταστήσουμε την εικονική μηχανή μέσω ενός στιγμιότυπου, δημιουργούμε εκ νέου τον Interrupt Controller και στην συνέχεια θέτουμε τους προαναφερόμενους registers απο την αποθηκευμένη πληροφορία του JSON αρχείου.

```
bits:     | 63   ....  40 | 39 ..  32  |  31   ....    0 |
values:   |    reserved   | vcpu_index |      offset     |
```

Σχήμα 18: Το πεδίο γνωρισμάτων (attribute field) του register (οποιουδήποτε τύπου) του irqchip [23]

3.2.6 Flattened Device Tree – FDT

Το Επίπεδο Δέντρο Συσκευών (FDT), είναι μια δομή δεδομένων που χρησιμοποιείται απο τον Linux πυρήνα για να περιγράψει την διαμόρφωση υλικού ενός συστήματος. Στα ενσωματωμένα συστήματα, οι συνδέσεις και τα εξατομικευμένα "εξαρτήματα" διαφοροποιούνται και δεν υπάρχουν συμβάσεις ή κανόνες – ακόμη και μεταξύ των ίδιων συστημάτων – για την διάταξη των διασυνδεδεμένων συσκευών και την κατανομή των πόρων. Περιέχει πληροφορίες για τις συνδεδεμένες συσκευές, τις περιοχές μνήμης, τον Interrupt Controller και άλλες "οντότητες" του συστήματος. Όπως αναφέραμε, το FDT χρησιμοποιείται κατά κύριο λόγο σε ARM-based συστήματα, ενώ σε x86 συναντάται πιο σπάνια (αντ' αυτού υλοποιούνται άλλοι μηχανισμοί για την αρχικοποίηση και το configuration του υλικού, όπως το ACPI – Advanced Configuration and Power Interface [24]). Για τον λόγο αυτό, έπρεπε να προσθέσουμε το δέντρο συσκευών στην υλοποίηση του Virtual Machine Manager.

Το FDT είναι χαρτογραφημένο στην μνήμη της εικονικής μηχανής, με αποτέλεσμα να αποκαθίσταται (στην περίπτωση επαναφοράς της εικονικής μηχανής από ένα στιγμιότυπο) όταν χαρτογραφούμε την μνήμη του microVM από το αρχείο μνήμης του στιγμιότυπου, χωρίς να απαιτείται κάποια άλλη διαδικασία από τον Διαχειριστή Εικονικής Μηχανής (VMM). Κατά την διαδικασία εκκίνησης (booting phase) του VM, αφού έχει αρχικοποιηθεί η κύρια μνήμη, οι εικονικός επεξεργαστής (Vcpu), ο Interrupt Controller, καθώς και τα διάφορα devices που είναι συνδεδεμένα στην εικονική μηχανή, μπορούμε να δημιουργήσουμε το Επίπεδο Δέντρο Συσκευών (FDT). Όλες αυτές οι αρχικοποιήσεις είναι απαραίτητες εφόσον οι κόμβοι του δέντρου μας πρέπει να περιέχουν όλες τις προαναφερόμενες "οντότητες" της εικονικής μηχανής. Πιο συγκεκριμένα, έχουμε ξεχωριστούς κόμβους (nodes) για το guest memory της εικονικής μηχανής, για τον Interrupt Controller, για το PSCI (Power State Coordination Interface System Software on ARM processors), για το ρολόι χρονισμού του microVM (RTC – Real Time Clock), και για τις διάφορες συνδεδεμένες συσκευές. Στην περίπτωση που αγνοήσουμε να προσθέσουμε κάποια οντότητα από τις παραπάνω σαν κόμβο στον δέντρο μας, τότε η λειτουργία του συγκεκριμένου resource δεν θα είναι η αναμενόμενη. Παραδείγματος χάριν, κατα την ανάπτυξη του πηγαίου κώδικα του SnapFaas για την μεταφορά του σε ARM συστήματα, αγνοήσαμε σε πρώτο στάδιο την προσθήκη του κόμβου για το ρολόι του συστήματος (RTC), με αποτέλεσμα να μην έχει συνδεθεί με τον σωστό τρόπο στο σύστημα μας σαν μία Memory Mapped Input/Output (MMIO) συσκευή και έτσι να μην μπορούμε να έχουμε την απαιτούμενη αλληλεπίδραση με αυτήν. Αφού έχουμε ολοκληρώσει λοιπόν την δημιουργία του Δέντρου Συσκευών μας, το τελευταίο βήμα που απομένει είναι η αποτύπωση του στην μνήμη της εικονικής μηχανής, μέσω της συνάρτησης *write_slice_at_addr()*, όπου ένα κομμάτι δεδομένων (στην συγκεκριμένη περίπτωση τα bytes τα οποία αποτελούν το FDT) "γράφεται" σε μία προκαθορισμένη διεύθυνση μνήμης (στην διεύθυνση που έχουμε διαφυλάξει για το FDT). Στο Σχήμα 19 παρουσιάζεται η συνάρτηση *create_fdt()* του VMM, που είναι υπεύθυνη για την δημιουργία του Flattened Device Tree.

```
pub fn create_fdt<T: DeviceInfoForFDT + Clone + Debug>(
    guest_mem: &GuestMemory,
    vcpu_mpidr: Vec<u64>,
    cmdline: &CStr,
    device_info: Option<&HashMap<(DeviceType, String), T>>,
    gic_device: &Box<dyn GICDevice>,
) -> Result<(Vec<u8>)> {

    // Alocate stuff necessary for the holding the blob.
    let mut fdt: Vec<u8> = vec![0; FDT_MAX_SIZE];

    allocate_fdt(&mut fdt)?;

    // Header or the root node as per above mentioned documentation.
    append_begin_node(&mut fdt, name: "")?;
    append_property_string(&mut fdt, name: "compatible", value: "linux,dummy-virt")?;
    // For info on #address-cells and size-cells read "Note about cells and address representation"
    // from the above mentioned txt file.
    append_property_u32(&mut fdt, name: "#address-cells", val: ADDRESS_CELLS)?;
    append_property_u32(&mut fdt, name: "#size-cells", val: SIZE_CELLS)?;
    // This is not mandatory but we use it to point the root node to the node
    // containing description of the interrupt controller for this VM.
    append_property_u32(&mut fdt, name: "interrupt-parent", val: GIC_PHANDLE)?;
    // println!("Prin tin create_cpu_nodes");

    // create_cpu_nodes(&mut fdt, num_cpus)?;
    create_cpu_nodes(&mut fdt, &vcpu_mpidr)?;
    create_memory_node(&mut fdt, guest_mem)?;
    create_chosen_node(&mut fdt, cmdline)?;

    // create_gic_node(&mut fdt, u64::from(num_cpus))?;
    create_gic_node(&mut fdt, gic_device)?;

    create_timer_node(&mut fdt)?;

    create_clock_node(&mut fdt)?;

    create_psci_node(&mut fdt)?;

    device_info.map_or(default: Ok(()), f: |v: &HashMap<(DeviceType, String), …>| create_devices_node(&mut fdt, dev_info: v))?;

    // End Header node.
    append_end_node(&mut fdt)?;

    // Allocate another buffer so we can format and then write fdt to guest.
    let mut fdt_final: Vec<u8> = vec![0; FDT_MAX_SIZE];
    finish_fdt(from_fdt: &mut fdt, to_fdt: &mut fdt_final)?;

    // Write FDT to memory.
    let fdt_address: GuestAddress = GuestAddress(get_fdt_addr(&guest_mem).try_into().unwrap());
    let written: usize = guest_mem &GuestMemory
        .write_slice_at_addr(buf: fdt_final.as_slice(), guest_addr: fdt_address) Result<usize, Error>
        .map_err(op: Error::WriteFDTToMemory)?;
    if written < FDT_MAX_SIZE {
        return Err(Error::IncompleteFDTMemoryWrite);
    }
    Ok(fdt_final)
} fn create_fdt
```

Σχήμα 19: Η συνάρτηση create_fdt()

3.3 Μεταφορα σε ARM αρχιτεκτονική

Η έκδοση του Firecracker, δηλαδή του VMM που χρησιμοποιήθηκε από την πρωτότυπη έκδοση του SnapFaas δεν υποστήριζε την εκκίνηση ενός microVM σε ARM συστήματα. Για να γίνει δυνατή η μεταφορά του SnapFaas σε ARM αρχιτεκτονική, χρειάστηκαν αλλαγές στον πηγαίο κώδικα του Firecracker καθώς και η προσθήκη συσκευών και δομών (structs) που είναι ARM specific, όπως αυτές που περιγράψαμε στο Κεφάλαιο 3.2 (General Interrupt Controller, Flattened Device Tree). Σε αυτό το κομμάτι θα αναλύσουμε τα ζητήματα που προέκυψαν κατά την διάρκεια της εφαρμογής των αλλαγών του πηγαίου κώδικα του Firecracker ώστε να τεθεί σε λειτουργία μια εικονική μηχανή μέσω του SnapFaas σε ARM αρχιτεκτονική και πιο συγκεκριμένα μέσα σε ένα Raspberry Pi 4 Model

b [25]. Όλες οι απαραίτητες αλλαγές στον τροποποιημένο Firecracker VMM, βρίσκονται σε github repository [37].

3.3.1 Ζητήματα που προέκυψαν

→ Η πρωτότυπη έκδοση του SnapFaas χρησιμοποιεί το outl εκτελέσιμο (ανήκει στην οικογένεια συναρτήσεων κλήσης συστήματος, που χρησιμοποιούνται για low level port input και output) για να ενημερώσει μέσα από τον κώδικα εκτέλεσης της εφαρμογής τον Διαχειριστή Εικονικής Μηχανής, μέσω context switch, για την λήψη ενός στιγμιότυπου. Με το outl εκτελέσιμο, το SnapFaas προωθεί κάποια συγκεκριμένα δεδομένα σε μορφή bytes, μέσω του μηχανισμού I/O port [26] σε κάποια θύρα που ορίζουμε. Στα x86 συστήματα κάθε συνδεδεμένη συσκευή έχει διαθέσιμο ένα εύρος θυρών στις οποίες μπορεί να γράψει ή να διαβάσει δεδομένα, κατά κύριο λόγο σε επίπεδο πυρήνα. Η διαδικασία αυτή κάνει trigger ένα Vcpu Exit (προκαλείται ένα context switch), το οποίο μπορεί να ανιχνευθεί απο το VMM στο σημείο που γίνεται η προσομοίωση του εικονικού επεξεργαστή, ελέγχωντας για τα bytes που έχουν σταλεί. Έπειτα, ο Διαχειριστής Εικονικής Μηχανής μπορεί να εκκινήσει όλες τις απαραίτητες διαδικασίες για να ληφθεί ένα στιγμιότυπο. Αυτός λοιπόν ο μηχανισμός ήταν το πρώτο μεγάλο ζήτημα που έπρεπε να επιλύσουμε στην μεταφορά του εργαλείου σε ARM αρχιτεκτονική, καθώς το διάβασμα και το γράψιμο δεδομένων από και σε I/O ports δεν είναι διαθέσιμο σε συστήματα ARM επεξεργαστών. Σε πρώτο στάδιο, απο την εξομοίωση του εικονικού επεξεργαστή στον Virtual Machine Manager, αφαιρέσαμε εξ' ολοκλήρου τις περιπτώσεις Vcpu Exit οι οποίες αφορούσαν reads και writes απο και σε Input/Output Ports.

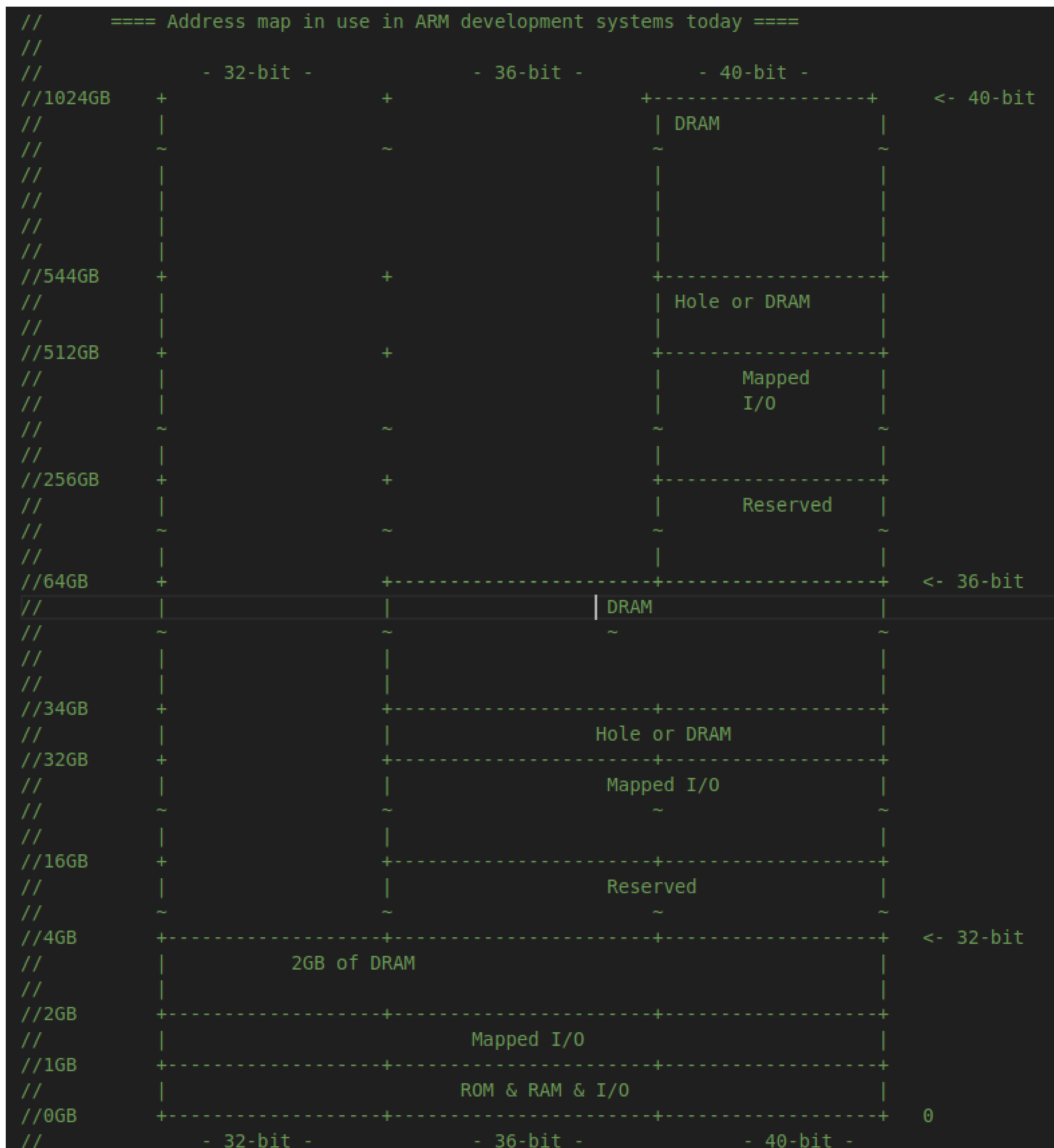
Για την επίλυση του συγκεκριμένου ζητήματος, κάναμε χρήση του μοναδικού διαθέσιμου τρόπου επικοινωνίας, στα ARM συστήματα, μεταξύ επεξεργαστή και συσκευών που ειναι η χαρτογραφημενη μνημη (memory-mapped Input/Output). Το Memory Input/Output λειτουργεί με αντίστοιχο τρόπο αφού για κάθε συνδεδεμένη συσκευή στο σύστημα δεσμεύεται ένα εύρος μνήμης το οποίο μπορεί να χρησιμοποιηθεί για input και output απο και προς την συσκευή αυτή. Κάναμε αρχικά εγγραφή μέσω του VMM μιας καινούργιας συσκευής για την προσομοίωση του ρολογιού πραγματικού χρόνου (real time clock – RTC [27]). Στην συνέχεια, δώσαμε στην συσκευή αυτή μια προκαθορισμένη εικονική διεύθυνση μέσα στον χαρτογραφημένο χώρο διευθύνσεων των υπόλοιπων συσκευών. Έπειτα, μέσα από τον κώδικα εκτέλεσης και συγκεκριμένα από τα σημεία εκείνα που στην πρωτότυπη έκδοση του SnapFaas πυροδοτούν την λήψη στιγμιότυπου (είτε γλώσσας, είτε συνάρτησης), αντικαταστήσαμε την χρησιμοποίηση της outl συνάρτησης με την linux utility εντολή hwclock μέσω της οποίας μπορούμε να επικοινωνήσουμε, σε επίπεδο προγραμματιστικής γλώσσας του χρήστη, με το real time clock device που υπάρχει πλέον στο microVM. Κάθε τέτοια αντίστοιχη εντολή μεταφράζεται σε συγκεκριμένα bytes που πρέπει να διαβαστούν από την συσκευή, στην συγκεκριμένη περίπτωση απο το real-time clock device. Για τον σκοπό αυτό προκαλείται ένα VcpuExit στην προσομοίωση του εικονικού επεξεργαστή, όπως ακριβως συνέβαινε και στην περίπτωση της Input/Output επικοινωνίας μέσω θύρας που είχε υιοθετηθεί για την πυροδότηση της ληψης snapshot στα x86 συστήματα. Τα δεδομένα από την hwclock εντολή μεταφέρονται λοιπόν, μέσω του διαύλου δεδομένων (data bus) της μνήμης και στον κώδικα του Διαχειριστή Εικονικής Μηχανής μπορούμε να τα "συλλάβουμε", εφόσον γνωρίζουμε την προκαθορισμένη διεύθυνση του RTC καθώς και τα ίδια τα δεδομένα που αποστέλλονται. Όταν λοιπόν διαπιστώσουμε την εκτέλεση της εντολής hwclock μέσα από το runtime entrypoint, μπορούμε να εκκινήσουμε όλες τις διαδικασίες εκείνες που απαιτούνται για την "αιχμαλώτιση" της κατάστασης της μηχανής σε ένα στιγμιότυπο.

→ Για τον εντοπισμό των διαφόρων σφαλμάτων που επιλύθηκαν αναφορικά με το booting phase του VM, χρειάστηκε να συνδέσουμε ένα earlycon device μέσω του οποίου ενεργοποιείται μια σειριακή κονσόλα (serial console) στην οποία καταγράφονται όλα τα μηνύματα που πηγάζουν από την φόρτωση του πυρήνα. Μέσω αυτής, καταφέραμε να εντοπίσουμε σφάλματα σχετιζόμενα με άλλες συσκευές που συνδέουμε στην εικονική μηχανή (όπως το σύστημα αρχείου) είτε με την διάταξη μνήμης. Ο λόγος που χρειάστηκε να προσθέσουμε το earlycon device, είναι ότι δεν ήταν εφικτή η ενεργοποίηση του serial console μέσω του I/O port (όπως είχε υλοποιηθεί στην πρωτότυπη έκδοση του Διαχειριστή Εικονικής Μηχανής του Snapfaas), καθώς όπως αναφέραμε και προηγουμένως, η συγκεκριμένη λειτουργία δεν είναι διαθέσιμη σε ARM αρχιτεκτονικές.

→ Για τα αρχεία αυτά μνήμης, μέσα στα στιγμιότυπα, που περιέχουν αποκλειστικά τις σελίδες μνήμης (memory pages) εκείνες που έχουν τροποποιηθεί κατά την διάρκεια εκτέλεσης (dirty pages), ήρθαμε αντιμέτωποι με ένα επιπλέον ζήτημα στην διαδικασία μεταφοράς του SnapFaas εργαλείου σε ARM αρχιτεκτονική. Η πρωτότυπη έκδοση του εργαλείου χρησιμοποιεί την καταχώρηση πίνακα σελίδων (page table entry – PTE) του πυρήνα Linux για καθεμία από τις σελίδες που αποκτά πρόσβαση (διαβάζοντας το αρχείο /proc/<pid_of_firecracker>/pagemap) για να καθορίσει αν η εκάστοτε σελίδα έχει τροποποιηθεί ή όχι. Πιο συγκεκριμένα, μέσα στο page table entry των σελίδων μνήμης υπάρχει ένα bit, το 55ο – soft-dirty bit [28], το οποίο χρησιμοποιεί ο πυρήνας του Linux για να "σημαδέψει" αν η εκάστοτε σελίδα είναι clean (δεν έχει τροποποιηθεί) ή dirty. Αυτή η πληροφορία χρειάζεται ώστε όταν μια σελίδα μνήμης απομακρυνθεί από την cache, στην περίπτωση που δεν έχει τροποποιηθεί, μπορεί απλά να παραμεριστεί, διαφορετικά θα πρέπει να αποθηκευτεί το περιεχόμενο της στο δίσκο. Εσωτερικά, όταν μια διεργασία τροποποιήσει μια σελίδα σε κάποια εικονική διεύθυνση, το page fault που προκαλείται οδηγεί τον πυρήνα να "σεταρει" το soft-dirty bit στο page table entry της σελίδας. Καταλαβαίνουμε λοιπόν ότι το soft-dirty bit παίζει καθοριστικό ρόλο στην διαδικασία που έχει υιοθετηθεί απο το SnapFaas για την λήψη και τον διαχωρισμό των δύο διαφορετικών διαθέσιμων στιγμιότυπων. Το ζήτημα που προέκυψε στην μεταφορά αρχιτεκτονικής αφορούσε την απουσία του συγκεκριμένου bit απο το PTE των σελίδων μνήμης στην ARM αρχιτεκτονική και το οποίο επιλύθηκε εφαρμόζοντας ένα patch [1] για τον ARM Linux πυρήνα μέσω του οποίου ενεργοποιήθηκε η επιλογή CONFIG_HAVE_ARCH_SOFT_DIRTY στο configuration του πυρήνα και μέσω της οποίας μπορέσαμε να ενεργοποιήσουμε την επιλογή ανίχνευσης αλλαγών στην μνήμη CONFIG_MEM_SOFT_DIRTY. Με αυτόν τον τρόπο ενεργοποιήθηκε το soft-dirty bit των PTE's των σελίδων μνήμης σε ARM αρχιτεκτονική.

→ Κατά την διάρκεια των αλλαγών που ήταν απαραίτητες στον κώδικα του Διαχειριστή Εικονικής Μηχανής για να τρέξει το SnapFaas στην ARM αρχιτεκτονική, εντοπίσαμε σε αρκετές περιπτώσεις – κατα την διάρκεια προσπαθειών δημιουργίας ενός στιγμιότυπου – το σφάλμα <u>InvalidGuestAddressRange</u>. Ο λόγος πίσω από το συγκεκριμένο σφάλμα είναι η διαφορετική διάταξη μνήμης σε ARM συστήματα (Σχήμα 20) σε σύγκριση με τα συστήματα x86 επεξεργαστών. Η βασική διαφορά είναι ότι η αρχή της RAM (Random Access Memory) στον χάρτη διευθύνσεων απέχει 2 GB χώρο μνήμης από την αρχή του χώρου διευθύνσεων, σε αντίθεση με τα x86 συστήματα στα οποία η RAM ξεκινάει από την αρχή του χάρτη διευθύνσεων. Επομένως χρειάστηκαν κάποιες αλλαγές ως προς την αντιστοίχιση των περιοχών μνήμης που προσπελάζονται (μετατόπιση των προσπελάσεων αυτών ώστε να περιλαμβάνουν ολόκληρο το μέγεθος της κύριας μνήμης της εικονικής μηχανής) για να αποθηκευτεί εν τέλει η κατάσταση μνήμης στο στιγμιότυπο. Ένα επιπλέον ζήτημα που τέθηκε προς αντιμετώπιση – και το οποίο πηγάζει από την ίδια την

διαφοροποίηση των 2 αρχιτεκτονικών – ήταν και η ενεργοποίηση των συσκευών, που αναφέρθηκε προηγουμένως. Η ενεργοποίηση των συσκευών όταν γίνεται η επαναφορά τους, προϋποθέτει η διεύθυνση του πίνακα περιγραφής (description table) της εκάστοτε συσκευής (οπως είναι το σύστημα αρχείου RootFS ή AppFS) να βρίσκεται εντός του εύρους της μνήμης του guest memory. Αφότου υλοποιήσαμε τις προαναφερόμενες αλλαγές στην μνήμη, η συνθήκη αυτή δεν ίσχυε καθώς η βάση της μνήμης του microVM όταν εκτελείται σε ARM είναι διαφορετική από αυτή στο x86. Μετατοπίζοντας το εύρος μνήμης που πρέπει να ανήκουν οι διευθύνσεις των συσκευών, οι εκάστοτε ενεργοποιήσεις ήταν επιτυχείς και οι συσκευές τέθηκαν κανονικά σε λειτουργία.



Σχήμα 20: Η διάταξη μνήμης σε ένα ARM υπολογιστικό σύστημα [2]

4 Μεθοδολογία

Σε αυτό το κεφάλαιο, θα αναλύσουμε περαιτέρω τους μηχανισμούς και την μεθοδολογία που υλοποιήσαμε για την πραγματοποίηση του πειραματικού σκέλου της διπλωματικής που θα παρουσιάσουμε στο Κεφάλαιο 5. Για να έχουμε ένα μέτρο σύγκρισης για το SnapFaas εργαλείο που μελετήσαμε και τροποποιήσαμε, φτιάξαμε έναν client για να μπορούμε να διαχειριστούμε το Firecracker εργαλείο με σκοπό να προσομοιώσουμε με τον καλύτερο δυνατό τρόπο το SnapFaas για να έχουμε δύο πανομοιότυπες λειτουργίες, οι οποίες μπορούν να συγκριθούν ένα προς ένα, σε ARM αρχιτεκτονική. Σε αυτό το σημείο αξίζει να σημειώσουμε ότι η έκδοση του Firecracker που χρησιμοποιήθηκε για τον client μας (Firecracker v1.6.0-dev) είναι αρκετά μεταγενέστερος της έκδοσης που συνδέθηκε με το SnapFaas, καθώς οι απαραίτητες αλλαγές από το Princeton University άρχισαν να υλοποιούνται πάνω σε έκδοση στην οποία η ARM αρχιτεκτονική δεν είχε υποστήριξη.

4.1 Στιγμιότυπα στο Firecracker

Πριν συνεχίσουμε με την ανάλυση του client που δημιουργήσαμε, θα περιγράψουμε τον τρόπο που δημιουργουνται και τα χαρακτηριστικά των snapshots του Firecracker ώστε να γίνουν κατανοητές οι διαφορές με τα snapshots του SnapFaas. Με την φόρτωση του στιγμιότυπου σε ένα Firecracker process το workload του microVM συνεχίζει την εκτέλεση του από το ίδιο σημείο που σταμάτησε. Για το αρχικό microVM από το οποίο δημιουργήθηκε το στιγμιότυπο, δεν υπάρχουν παρενέργειες από αυτή την διαδικασία (εκτός από την καθυστέρηση που εισάγεται από την διαδικασία δημιουργίας στιγμιότυπου). Υπάρχει πιθανότητα απώλειας πακέτων δικτύου και δεδομένων που ανταλλάσσονται μέσω της virtual socket συσκευής στο microVM που συνεχίζεται από ένα snapshot. Επίσης δεν είναι εγγυημένο ότι η κατάσταση των συνδέσεων δικτύου (network connections) θα παραμείνει σταθερή.

Για να πετύχει την επαναφορά ενός microVM, ο snapshot μηχανισμός του Firecracker αποθηκεύει ολόκληρη την κατάσταση των παρακάτω πόρων: 1) της μνήμης του guest και 2) την εξομοιούμενη κατάσταση του υλικού (emulated HW state), τόσο του υλικού που εξομοιώνεται απο το Firecracker αλλά και του υλικού από το KVM, όπως συμβαίνει και στο SnapFaas. Η κατάσταση των προαναφερώμενων components παράγεται ανεξάρτητα, το οποίο μας προσφέρει ευελιξία στον υποστηριζόμενο snapshot μηχανισμό. Αυτό σημαίνει ότι η λήψη ενός στιγμιότυπου καταλήγει σε δύο αρχεία, το αρχείο μνήμης του φιλοξενούμενου (guest memory file) και το αρχείο κατάστασης της εικονικής μηχανης (microVM state file) που συνθέτουν το ολοκληρωμένο στιγμιότυπο ενός microVM.

Όταν φορτώνεται ένα στιγμιότυπο, το Firecracker δημιουργεί μια MAP_PRIVATE χαρτογράφηση του αρχείου μνήμης (μέσω της κλήσης συστήματος *mmap()*), το οποίο οδηγεί σε φόρτωση των σελίδων μνήμης κατά απαίτηση (on demand) κατά την διάρκεια εκτέλεσης του φόρτου εργασίας (runtime workload) του microVM. Οποιεσδήποτε επόμενες εγγραφές μνήμης μεταφέρονται σε μια αντιγραφή-σε-εγγραφή ανώνυμη μνήμη αντιστοίχισης (Copy-on-Write anonymous memory mapping). Αυτή η διαδικασία φόρτωσης της μνήμης είναι πανομοιότυπη με την μέθοδο lazy restoration του SnapFaas. Αυτή η διαδικασία έχει το πλεονέκτημα της πολύ γρήγορης φόρτωσης του στιγμιότυπου, άλλα έχει το κόστος του να πρέπει να παραμείνει στην μνήμη το guest memory file για όλη την διάρκεια ζωής της εικονικής μηχανής, καθώς και το κόστος των σύγχρονων page faults

που είχαμε αναφέρει και για το SnapFaas στην περίπτωση του lazy restoration, κατά την διάρκεια εκτέλεσης της συνάρτησης.

Όταν ένα microVM είναι σε Paused state, μπορεί να δημιουργηθεί είτε ένα πλήρες στιγμιότυπο (full), είτε ένα στιγμιότυπο διαφοράς (diff). Τα πλήρη στιγμιότυπα δημιουργούν ένα ολοκληρωμένο, συνεχιζόμενο snapshot της τρέχουσας κατάστασης του microVM και της μνήμης του, ενώ τα diff snapshots "αιχμαλωτίζουν" την τρέχουσα κατάσταση του microVM και την μνήμη που έχει τροποποιηθεί, από το τελευταίο στιγμιότυπο (πλήρες ή διαφοράς). Τα στιγμιότυπα διαφοράς δεν είναι συνεχιζόμενα και πρέπει να συγχωνευτούν με ένα υπάρχον πλήρες για να φορτωθούν με επιτυχία. Η σειρά με την οποία τα στιγμιότυπα δημιουργούνται έχει σημασία και πρέπει να συγχωνεύονται με τον ίδιο τρόπο με τον οποίο έχουν δημιουργηθεί. Για την συγχώνευση ενός diff snapshot αρχείου μνήμης πάνω σε ένα αρχείο μνήμης ενός base snapshot, ο χρήστης πρέπει ασύγχρονα να αντιγράψει το περιεχόμενο του πρώτου πάνω από το περιεχόμενο του base. Για να πετύχουμε την συγχώνευση του language snapshot με το function snapshot (όπως τα έχουμε ορίσει στο SnapFaas περιβάλλον, εφόσον έχουμε διατηρήσει την ίδια λογική στην λήψη τους και στην περίπτωση του Firecracker), χρησιμοποιούμε το εργαλειο rebase-snap. Εκτελούμε την εξής εντολή: *rebase-snap --base-file path/to/base --diff-file path/to/layer*. Μετά την εκτέλεση της, η συγχωνευμένη βάση είναι ένα συνεχιζόμενο αρχείο μνήμης στιγμιότυπου που περιγράφει την κατάσταση της μνημης την στιγμή δημιουργίας του τελευταίου στρώματος στιγμιότυπου. Αξίζει να σημειώσουμε ότι τα στιγμιότυπα είναι μοναδικά για το εκάστοτε μέγεθος που εκκινείται η εικονική μηχανή, δηλαδή στην περίπτωση που έχουμε δημιουργήσει ένα στιγμιότυπο βασισμένο σε ένα VM με μέγεθος μνήμης 128 MB, το συγκεκριμένο στιγμιότυπο μπορεί να επαναφέρει αποκλειστικά εικονικές μηχανές του ίδιου μεγέθους. Με την χρήση γνωστόν ballooning τεχνικών [6], θα μπορούσαμε να έχουμε κοινά στιγμιότυπα για όλα τα ζητούμενα μεγέθη, αλλά κάτι τέτοιο δεν έχει υλοποιηθεί στο SnapFaas, οπότε δεν το εφαρμόσαμε ούτε στην υλοποίηση του vanilla Firecracker. Τα αρχεία κατάστασης δεν πρέπει να συγχωνεύονται και αντ' αυτού χρησιμοποιείται το αρχείο κατάστασης που προκύπτει από την δημιουργία του τελευταίου στρώματος στιγμιότυπου.

4.2 Τεχνική υλοποίηση Firecracker client

Τα microVMs του Firecracker μπορούν να εκτελέσουν ενέργειες μέσω του PUT API (Application Programming Interface) REST Request (τα αιτήματα PUT χρησιμοποιούνται συνήθως για την ενημέρωση ενός υπάρχοντος πόρου σε μια εφαρμογή). Μέσω αυτών των action requests μπορούμε να εκτελέσουμε όλες τις δυνατές ενέργειες για την διαμόρφωση των διαφόρων παραμέτρων της εικονικής μηχανής, την εκκίνηση της, την σύνδεση συσκευών, την εξαγωγή μετρικών, καθώς και την λήψη και φόρτωση στιγμιότυπων. Για να είναι συγκρίσιμοι οι χρόνοι του cold-start που πετύχαμε μέσω του vanilla Firecracker (με τον δικό μας client) με την μεταφερόμενη έκδοση του SnapFaas σε ARM αρχιτεκτονικές, χρησιμοποιήσαμε τον ίδιο "απογυμνωμένο" Linux πυρήνα (vmlinux-4.20.0) καθώς και τα ίδια συστήματα αρχείων, RootFS (κοινό για όλα τα benchmarks) και AppFS (ξεχωριστό για κάθε εφαρμογή), για το microVM που ξεκινάμε σε κάθε περίπτωση.

Πιο συγκεκριμένα, φτιάξαμε ένα bash εκτελέσιμο αρχείο, ονόματι client-start.sh, μέσα στο οποίο τοποθετήσαμε όλα τα API requests που είναι απαραίτητα για την εκκίνηση ενός microVM πανομοιότυπου με αυτό που εκκινείται από το SnapFaas εργαλείο. Αρχικά, μέσω του */boot-source* request, καθορίζουμε τον προαναφερόμενο πυρήνα που θα χρησιμοποιηθεί για την εκκίνηση της εικονικής μηχανής, καθώς και τις παραμέτρους του. Μέσω του

/drives/{drive_id} δημιουργούνται οι μονάδες δίσκου για το RootFS (με παράμετρο "is_root_device: true" και read & write privileges) και AppFS (με παράμετρο "is_root_device: false" και read & write privileges) και μέσω του */vsock* endpoint δημιουργείται το vsock device μας που θα χρησιμοποιήσουμε και σε αυτήν την υλοποίηση ώστε να περάσουμε το αίτημα προς επεξεργασία στην εικονική μηχανή που εκτελείται η εφαρμογή μας. Στην συνέχεια, εκτελούμε το API PUT request */machine-config* όπου ενημερώνεται η διαμόρφωση (configuration) του μηχανήματος, με το πλήθος των εικονικών επεξεργαστών, το μέγεθος της μνήμης, καθώς και την ενεργοποίηση ή οχι της δυνατότητας ανίχνευσης των τροποποιημένων σελίδων (track dirty pages). Η συγκεκριμένη παράμετρος είναι άρρηκτα συνδεδεμένη με την λήψη diff snapshot, καθώς μόνο όταν είναι ενεργοποιημένη η συγκεκριμένη λειτουργία έχουμε την δυνατότητα για δημιουργία ενός memory diff snapshot. Αυτό συμβαίνει καθώς μόνο το diff snapshot (και όχι το full snapshot) περιέχει – πέραν της κατάστασης της εικονικής μηχανής – τις σελίδες εκείνες της μνήμης που έχουν τροποποιηθεί (dirtied) σε σχέση με το προηγούμενο snapshot (full). Τα full snapshots περιέχουν μια ολοκληρωμένη αντιγραφή της κύριας μνήμης της εικονικής μηχανής. Θα αναφερθούμε και πιο αναλυτικά στις διαφορές των full και diff snapshots στο περιβάλλον του Firecracker και το τρόπο λήψης και λειτουργίας τους. Στην συνέχεια, μέσω του API request */network-interfaces/{iface_id}* δημιουργούμε ένα network interface eth0, για την σύνδεση του VM με το διαδίκτυο και την ανάθεση σε αυτό μιας συγκεκριμένης IP (Internet Protocol). Επιπλέον, παρέχουμε στην διεπαφή δικτύου μια συγκεκριμένη MAC διεύθυνση και την συνδέουμε με ένα tap0 device (ένα εικονικό Linux network interface που χρησιμοποιείται για την εξομοίωση μιας συσκευής επιπέδου 2 – επιτρέπει στις εικονικές μηχανές να επικοινωνούν με τον host) που έχουμε αρχικοποιήσει σε επίπεδο host μεσω του Linux utility ip και πιο συγκεκριμένα μέσω των εντολών: 1) *ip tuntap add tap0 mode tap* && 2) *ip link set tap0 up*. Τέλος, πυροδοτείται η ενέργεια *InstanceStart* μέσα από το */actions* requests, η οποία ενεργοποιεί το microVM και ξεκινά το λειτουργικό σύστημα του guest. Μπορεί να κληθεί επιτυχώς μόνο μια φορά.

Το bash script που περιγράψαμε είναι το πρώτο που εκτελείται σε μια αλυσίδα εκτελέσεων που αποτελούν τον client που συντάξαμε για το πειραματικό σκέλος της διπλωματικής για την σύγκριση με το SnapFaas. Αποτέλεσμα της εκτέλεσης του είναι η εκκίνηση της εικονικής μηχανής με το Root Filesystem και το οποίο, όπως περιγράψαμε και στην περίπτωση του SnapFaas, φορτώνει όλες τις απαραίτητες βιβλιοθήκες και modules για την αρχικοποίηση της γλώσσας εκτέλεσης των εφαρμογών (Python). Απο την στιγμή που θέλουμε η σύγκριση μας να γίνει με το vanilla Firecracker αυτούσιο, δεν υπάρχει η δυνατότητα αλλαγής στον πηγαίο κώδικα του Virtual Machine Manager ώστε να πυροδοτύσουμε την λήψη των δύο διαφορετικών στιγμιότυπων μέσα από το runtime workload της γλώσσας και να το κάνουμε "catch" στην προσομοίωση του Vcpu. Για τον λόγο αυτό, χρειάστηκε να τροποποιήσουμε το runtime workload του RootFS ώστε να προστεθούν δύο σημεία στα οποία να έχουμε γνώση απο τον host ότι η εκτέλεση του κώδικα έχει φτάσει σε αυτά ώστε να πυροδοτήσουμε χειροκίνητα την λήψη του στιγμιότυπου. Με άλλα λόγια, δύο σημεία στα οποία ο κώδικας "σταματάει" και μας δίνει το περιθώριο να κάνουμε trigger τα snapshots.

Το συγκεκριμένο ζήτημα ήταν από τα πιο δύσκολα που έπρεπε να επιλύσουμε για τον client μας, καθώς προσπαθήσαμε να εφαρμόσουμε διάφορες λύσεις οι οποίες απέτυχαν. Σαν πρώτο βήμα προσπαθήσαμε να εκμεταλλευτούμε την χρήση του virtual socket που υπάρχει μέσα στην εικονική μηχανή και χρησιμοποιούμε για να περάσουμε τα δεδομένα των αιτημάτων προς τις εφαρμογές, καθως και τις απαντήσεις αυτών πίσω στον host. Σε σύγκριση με το connection που εγκαταστήσαμε στο SnapFaas (σύνδεση που ξεκινάει από τον guest – Guest-Initiated connection), στην περίπτωσή μας υλοποιήθηκε η αντίστροφη

διαδικασία, δηλαδή μια σύνδεση που γίνεται initiate απο τον host. Για να το πετύχουμε αυτό, έπρεπε στο runtime workload να δημιουργήσουμε έναν client socket ο οποίος θα περιμένει για μια σύνδεση από τον "εξωτερικό κόσμο". Πιο συγκεκριμένα μέσω της *listen()* συνάρτησης το socket "ακούει" στην πόρτα που έχουμε ορίσει εμείς και αναμένουμε την σύνδεση από τον host. Τέλος, ο κώδικας μπαίνει σε κατάσταση αναμονης μέσω της socket συνάρτησης *accept()*, περιμένοντας την σύνδεση από τον host και πιο συγκεκριμένα από έναν server socket. Αναλυτικά ο κώδικας που χρησιμοποιήθηκε φαίνεται στο Σχήμα 21. Χρησιμοποιούμε το flag VMADDR_CID_ANY [4] ώστε να μπορεί να δεχθεί σύνδεση από οποιαδήποτε διεύθυνση. Εκμεταλλευτήκαμε την κατάσταση αναμονής στην οποία περιέρχεται ο κώδικας για την ληψη του στιγμιότυπου διαφοράς (diff), δηλαδη αφότου έχουμε εισαγάγει το module της εκάστοτε εφαρμογής. Για την δεύτερη αναμονή του κώδικα που χρειαζόμαστε, για την λήψη του base snapshot, προσπαθήσαμε να εφαρμόσουμε πάλι την ίδια λογική, οπότε στην ουσία να έχουμε δύο διαφορετικές accept() συναρτήσεις στις οποίες ο κώδικας μπαίνει σε waiting mode. Μια τέτοια προσέγγιση αποδείχθηκε ανέφικτη καθώς όταν γίνεται αποκατάσταση microVM στο Firecracker, η κατασταση του vsock δεν μπορεί να επανέλθει με αποτέλεσμα να μην είναι δυνατή η επανεκκίνηση της σύνδεσης. Ο συγκεκριμένος περιορισμός στην επαναφορα του snapshot είναι ζήτημα προς επίλυση στην κοινότητα του Firecracker [5].

```python
# for function diff snapshot
######################################################################

print("before accept, for diff snapshot")

VSOCKPORT = 52
sock = socket.socket(socket.AF_VSOCK, socket.SOCK_STREAM)
hostaddr = (socket.VMADDR_CID_ANY, VSOCKPORT)

sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)

sock.bind(hostaddr)

sock.listen()

client_socket2, client_addr2 = sock.accept()
print(f"Accepted connection from {client_addr2}")

######################################################################
```

Σχήμα 21: Ο Python κώδικας για την δημιουργία και την αναμονή του virtual socket

Μια διαφορετική προσέγγιση που εφαρμόσαμε ανεπιτυχώς ήταν η χρησιμοποίηση της *sleep()* συνάρτησης. Πιο συγκεκριμένα, προσθέσαμε την συνάρτηση αυτή από το time module και η οποία αναστέλλει την εκτέλεση για ένα καθορισμένο χρονικό διάστημα, με σκοπό μέσα σε αυτό το διάστημα να πυροδοτήσουμε την λήψη του base στιγμιότυπου και στην επαναφορά της εικονικής μηχανής μέσω του στιγμιότυπου αυτού, η εκτέλεση να συνεχίσει αμέσως μετά την *sleep()*. Η υλοποίηση αυτή δεν εφαρμόστηκε καθώς μετά την επαναφορά του microVM η εκτέλεση παρέμενε "κολλημένη" στην *sleep()* συνάρτηση με πιθανή αιτία τον μη συγχρονισμό του Real Time Clock (RTC) device, με αποτέλεσμα την αδυναμία μηδενισμού του υπολειπόμενου χρόνου της. Μετά απο τις προαναφερόμενες υλοποιήσεις (2η accept(), sleep()), καταλήξαμε για το περιθώριο λήψης του στιγμιότυπου γλώσσας, στην προσθήκη υπολογιστικού φορτίου μέσω μιας *for-loop* η οποία προσθέτει καθυστέρηση στην εκτέλεση του runtime workload. Μέσα σε αυτή την χρονική

καθυστέρηση έχουμε την δυνατότητα χειροκίνητα να πυροδοτήσουμε την λήψη του base snapshot.

Για να το πετύχουμε αυτό δημιουργήσαμε άλλο ένα bash script με την ονομασία client-snapshot.sh, το οποίο εκτελείται χειροκίνητα όταν η εκτέλεση του runtime workload είναι στο σημείο της for-loop που προστέθηκε, πριν την φόρτωση του Application File System και αφότου έχει αρχικοποιηθεί η γλώσσα εκτέλεσης. Αρχικά, πρέπει να σταματήσουμε το τρέχον microVM και τον εικονικό επεξεργαστή του, οπότε καλούμε το API command */vm*, με περασμένη την παράμετρο: "state": "Paused" ώστε να θέσει την κατάσταση του VM στην επιθυμητή. Στην συνέχεια, εκτελουμε το API command */snapshot/create* με τις εξής παραμέτρους:

"snapshot_type": "Full",
"snapshot_path": "/path/to/snapshot_file",
"mem_file_path": "/path/to/mem_file"

Σαν αποτέλεσμα, τα δύο αρχεία που προαναφέραμε (ένα μνήμης και ένα κατάστασης της μηχανής) αποθηκεύονται στα καθορισμένα μονοπάτια. Έπειτα, και αφού έχει ολοκληρωθεί η λήψη του στιγμιότυπου εκτελούμε το client-restore.sh bash script, μέσω του οποίου επαναφέρουμε την εικονική μηχανή εκτελώντας με το curl εργαλείο το API command */snapshot/load*, περνώντας του τις εξής παραμέτρους:

"snapshot_path": "/path/to/base_snapshot_file",
"mem_backend": {
        "backend_path": "/path/to/base_mem_file",
        "backend_type": "File"
},
"enable_diff_snapshots": true,
"resume_vm": true

Το πεδίο backend_type αντιπροσωπεύει τον τύπο μνήμης που χρησιμοποιείται για την φόρτωση του στιγμιότυπου. Με τον τύπο αρχείου που έχουμε επιλέξει και αναλύσει, βασιζόμαστε στον πυρήνα για την διαχείριση σφαλμάτων σελίδας όταν φορτώνουμε το περιεχόμενο του αρχείου μνήμης στην μνήμη. Για την σωστή επαναφορά του microVM, πρέπει να είναι ήδη αρχικοποιημένες οι tap συσκευές και τα vsock backing sockets που είχαν χρησιμοποιηθεί στην αρχική εικονική μηχανή, καθώς και να είναι προσπελάσιμα από την νέα Firecracker διεργασία από το ίδιο μονοπάτι όπως και στο αρχικό microVM. Ως αποτέλεσμα της παραπάνω εντολής, η πλήρης κατάσταση του microVM φορτώνεται απο το στιγμιότυπο στην Firecracker διεργασία. Η εικονική μηχανή συνεχίζει αυτόματα από το σημείο που βρισκόταν και υπάρχει πλέον η δυνατότητα λήψης ενός στιγμιότυπου διαφοράς. Αξίζει να σημειωθεί ότι από εδώ και πέρα και σε σχέση με ένα ενδεχόμενο diff snapshot που μπορεί να προκύψει, όλες οι σελίδες μνήμης θεωρούνται μη τροποποιημένες (clean memory pages).

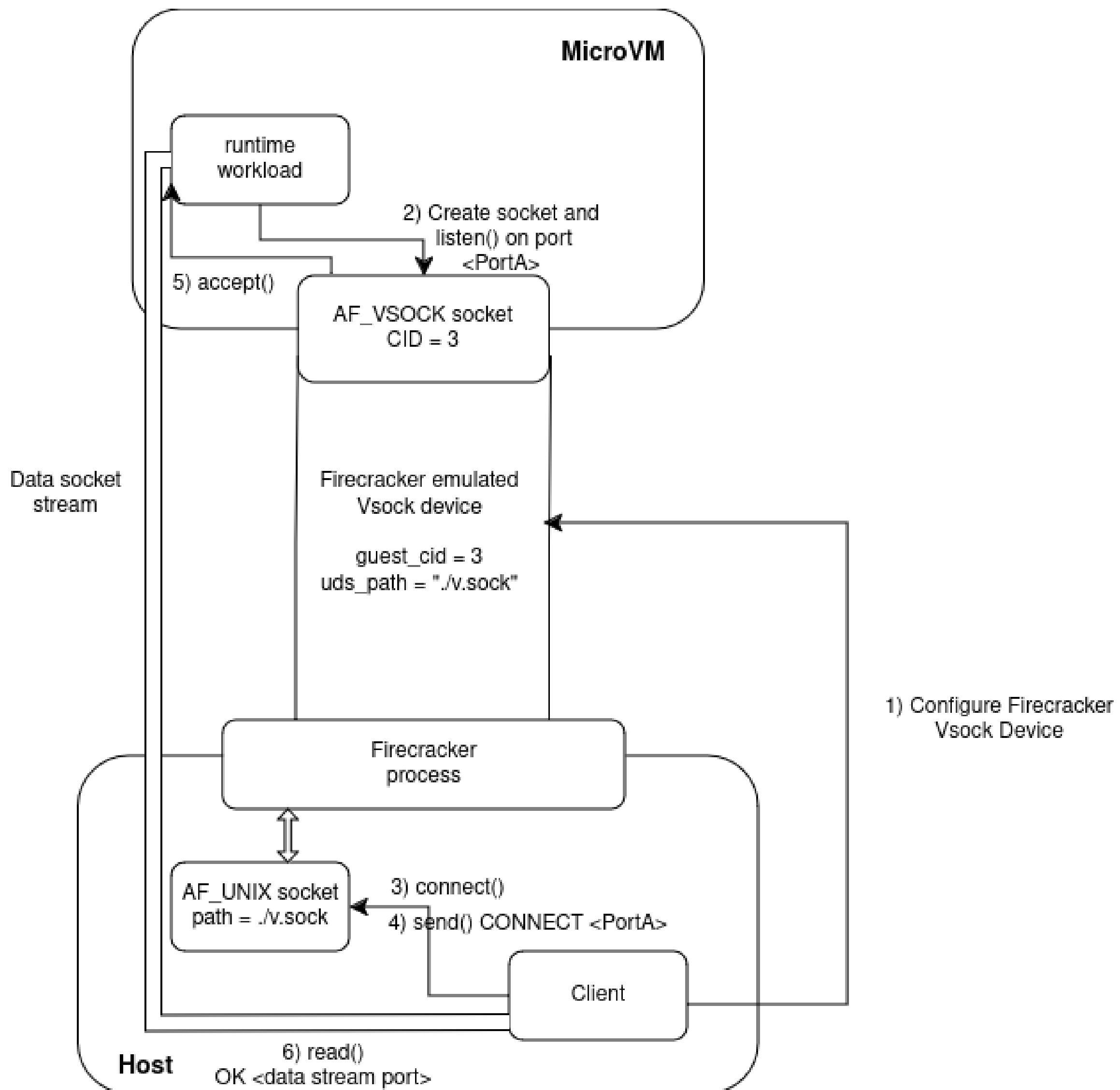
Έπειτα, το runtime workload συνεχίζει την εκτέλεση του και φορτώνει το Application workload με αμέσως επόμενη ενέργεια την αρχικοποίηση του virtual client socket, με τον τρόπο που αναλύσαμε προηγουμένως. Πλέον η εκτέλεση βρίσκεται σε waiting mode στην socket συνάρτηση *accept()* και μπορούμε να πυροδοτήσουμε την λήψη του diff snapshot. Για να το πετύχουμε, εκτελούμε τα ίδια API commands με την περίπτωση του base snapshot, με την μόνη διαφορά το πεδίο “snapshot_type” έχει τιμή “Diff” και όχι “Full”. Η “αιχμαλώτιση” του diff snapshot είναι εφικτή εφόσον το πεδίο *enable_diff_snapshots* έχει

σεταριστει κατά την διάρκεια φόρτωσης του language snapshot. Το πεδίο "mem_file_path" αυτή τη φορά (σε σύγκριση με το full snapshot) περιλαμβάνει μια εν μέρη αντιγραφή (diff copy) της μνήμης του guest, δηλαδή τις σελίδες μνήμης εκείνες που έχουν τροποποιηθεί από όταν λήφθηκε το προηγούμενο snapshot (ακριβώς πριν η εκτέλεση του runtime workload του RootFS, φορτώσει τις αρχικοποιήσεις της εκάστοτε συνάρτησης). Επίσης προσέχουμε το μονοπάτι των αρχείων μνήμης και κατάστασης του microVM να είναι μοναδικό για να μην κάνει overwrite το αρχείο μνήμης του full snapshot, καθώς χρειαζόμαστε και τα δύο για να συγχωνευθούν και να παραχθεί το τελικό memory file, μέσω του *rebase-snap* εργαλείου που περιγράψαμε παραπάνω.

Σε αυτό το σημείο έχουμε έτοιμα όλα τα διαφορετικά στιγμιότυπα που θα χρειαστούμε για την φόρτωση του εκάστοτε microVM με την αντίστοιχη συνάρτηση/benchmark που θέλουμε να τρέξουμε. Μία σημαντική διαφοροποίηση συγκριτικά με το SnapFaas εργαλείο, είναι ότι <u>δεν μπορούμε εξαρχής να έχουμε ένα ενιαίο language snapshot στιγμιότυπο που μπορεί να χρησιμοποιηθεί από όλες τις διαθέσιμες συναρτήσεις και πρέπει να έχουμε τόσα στιγμιότυπα γλώσσας όσα και το πλήθος των συναρτήσεων</u>. Αυτό οφείλεται στο γεγονός ότι στην δημιουργία του microVM δεν μπορούμε να περάσουμε το AppFS σαν ένα απλό placeholder όπως στο SnapFaas, όποτε κάθε στιγμιότυπο γλώσσας είναι άρρηκτα συνδεδεμένο με το εκάστοτε AppFS και άρα με το εκάστοτε benchmark. Μετά το merging των αρχείων μνήμης (του εκάστοτε στιγμιότυπου διαφοράς για κάθε συνάρτηση πάνω στο αρχείο μνήμης του αντίστοιχου στιγμιότυπου βάσης) – η οποία διαδικασία μπορεί να θεωρηθεί ασύγχρονη, υπό την έννοια ότι δεν προστίθεται σαν καθυστέρηση στο συνολικό latency της αποκατάστασης ενός microVM – για κάθε συνάρτηση χρειαζόμαστε ένα ολοκληρωμένο αρχείο μνήμης και το τελικό state file της μικρο-εικονικής μηχανής.

Για να ολοκληρωθεί το λειτουργικό μέρος του client μας, χρειαζόμαστε ένα script το οποίο θα συνδέεται στο virtual socket που περιμένει στην πλευρά του εικονικού μηχανήματος και θα στέλνει τα requests στην εκάστοτε εφαρμογή (αναλόγως του ποιου στιγμιότυπου έχει φορτωθεί). Επιπλέον, πρέπει να λαμβάνει τα δεδομένα της απάντησης και να τα μετατρέπει σε αναγνώσιμα data. Χρησιμοποιήσαμε Python για την γλώσσα εκτέλεσης, για να είναι συμβατή με την γλώσσα εκτέλεσης του runtime workload του microVM, και ονομάσαμε το script <u>host-init_client.py</u>. Επειδή στα πειράματα σύγκρισης (που θα αναλύσουμε στο κεφάλαιο 5) μετράμε τον χρόνο που χρειάζεται η εικονική μηχανή για να είναι έτοιμη να εξυπηρετήσει ένα αίτημα, εκτελούμε το host-init_client.py script αμέσως μετά το bash script με το οποίο γίνεται επαναφορά το τελικό στιγμιότυπο και τοποθετούμε την *connect()* socket συνάρτηση μέσα σε μια *While* λούπα για να ξέρουμε ότι όταν συνδεθεί το Unix socket στο microVM, θα είναι η στιγμή που μόλις είναι έτοιμο να δεχτεί ένα αίτημα και όχι αργότερα. Για να είναι εφικτή η επαναχρησιμοποίηση της *connect()* συνάρτησης (για όσες επαναλήψεις χρειαστούν μέχρι να γίνει η σύνδεση), έχουμε αρχικοποιήσει το virtual socket (μέσα στο microVM) με το flag: *socket.SO_REUSEADDR*, το οποίο επιτρέπει την επαναχρησιμοποίηση της τοπικής διεύθυνσης (local address) που ήδη χρησιμοποιεί. Παραθέτουμε το μεγαλύτερο μέρος του Python κώδικα που χρησιμοποιήθηκε στο Σχήμα 23. Προκειμένου να εγκαθιδρυθεί η σύνδεση, το virtual socket χρειάζεται συγκεκριμένα bytes μετά την αποδοχή της σύνδεσης από τον host (τα οποία αποτελούν το απαραίτητο "handshake"). Αυτά περιλαμβάνουν την λέξη "CONNECT" και το νούμερο της θύρας που έχει επιλεχθεί να ακουει η εικονική υποδοχή (virtual socket) – στην συγκεκριμένη περίπτωση, η 52. Αφού διαβάσουμε, μέσω της *recv()* socket συνάρτησης τα bytes της απάντησης ("OK 52"), η σύνδεση έχει ολοκληρωθεί και μπορούμε πλέον να στείλουμε τα δεδομένα του εκάστοτε request στη εικονική μηχανή για επεξεργασία από την εφαρμογή. Τα δεδομένα που θα σταλούν πρέπει να υποστούν επεξεργασία για να μπορούν να μεταφερθούν μέσω της σύνδεσης unix socket (host) - virtual socket (guest). Πιο

συγκεκριμένα, τα requests είναι hardcoded στην Python σε μορφή dictionary. Για να μπορούν να περάσουν σαν παράμετρος στην εφαρμογή, χρειάζεται να μετατραπεί σε json μορφή, το οποίο επιτυγχάνεται μέσω της *json.dumps()* συνάρτησης. Τα json πλέον δεδομένα του request πρέπει μετατραπούν σε bytes για να μπορούν να μεταφερθούν μεσα απο τα sockets και αυτό επιτυγχάνεται μέσω της *str.encode()* συνάρτησης. Απο την στιγμή που θα σταλεί το request, η εκτελεση του script μεταβαίνει σε waiting mode στην συνάρτηση *recv()*, μέχρι να ληφθεί το πλήθος των bytes που έχουν δηλωθεί σαν παράμετρος (στην συγκεκριμένη περίπτωση 4096). Για να μην έχουμε απώλεια δεδομένων, φροντίζουμε να λαμβάνουμε τα δεδομένα ανά παρτίδες των 4096 bytes γιατί για τόσα είμαστε βέβαιοι ότι η receive συνάρτηση θα λειτουργήσει σωστά. Κάθε τέτοια παρτίδα συναθροίζεται στο συνολικό response το οποίο, λόγω της μεταφοράς του απο τα sockets, χρειάζεται εκ νέου de-serialization (για να μετατραπεί σε αναγνώσιμη μορφή) και το οποίο επιτυγχάνεται μέσω του pickle module και πιο συγκεκριμένα μέσω της συνάρτησης *pickle.loads()*. Η διαδικασία δημιουργίας του καναλιού επικοινωνίας μεταξύ client και microVM, αναπαρίσταται στο Σχήμα 22.



Σχήμα 22: Connection establishment between Unix socket (client) and Virtual Socket (guest)

Για το τελικό στάδιο του client μας, φτιάξαμε ένα bash script (experiments.sh) το οποίο διαχειρίζεται όλα τα επιμέρους εκτελέσιμα με την απαιτούμενη αλληλουχία ώστε να συνεχιστεί η εικονική μηχανή από το τελικό στιγμιότυπο της εκάστοτε συνάρτησης, σε μια καινούργια Firecracker διεργασία και έπειτα να στείλει το κατάλληλο request. Πιο συγκεκριμένα, το bash script experiments.sh δέχεται σαν παραμέτρους το όνομα της συνάρτησης που θέλουμε εκτελεστεί μέσω του microVM (ώστε να περάσουμε το αντίστοιχο στιγμιότυπο στην */snapshot/load* command), το μέγεθος μνήμης του microVM που θα δημιουργηθεί καθώς και αν πρόκειται για "κρύα" (cold-start) ή "ζεστή" (warm-start) εκκίνηση (τις οποίες θα αναλύσουμε παρακάτω). Για την περίπτωση του cold-start εκτελούμε την εντολή: *sudo sync; echo 1 > /proc/sys/vm/drop_caches,* με την οποία "καθαρίζουμε" την Page Cache του host μηχανήματος, ώστε να είμαστε σίγουροι ότι δεν θα υπάρχουν "υπολείμματα" στην προσωρινή μνήμη από προηγούμενες εκτελέσεις του ίδιου microVM. Όλα τα scripts που αποτελούν τον client του Firecracker, καθώς και το τροποποιημένο runtime workload που περιλαμβάνεται στο RootFS, έχουν γίνει upload στο Github [7].

```python
handshake = b'CONNECT 52\n'

while True:
    sock = socket.socket(socket.AF_UNIX, socket.SOCK_STREAM)
    try:
        sock.connect('./worker-52.sock')
        end_vmready = datetime.now()
    except Exception as e:
        # print('oxi ready akoma')
        continue
    # print("Connected to second AF_VSOCK server")
    sock.sendall(handshake)
    # time.sleep(1)
    response = sock.recv(14)
    if response:
        end_vmready = datetime.now()
        break

print("Handshake's response")
print(response)

start = datetime.now()

sock.sendall(str.encode(json.dumps(sent_req)))


response2 = b""
while True:
    packet = sock.recv(4096)
    if not packet:
        break
    response2 += packet

final_response = pickle.loads(response2)

end = datetime.now()
```

Σχήμα 23: Ο Python κώδικας στην πλευρά του host για την επικοινωνία με το VM

4.3 Χαρακτηριστικά Περιβάλλοντος Εκτέλεσης

Για το πειραματικο κομμάτι της διπλωματικής εργασίας, χρησιμοποιησαμε ενα Raspberry Pi 4 Model B με τα εξής χαρακτηριστικά: RAM: 8GB, CPUs: 8, Thread(s) per core: 2 και CPU MHz: 2800.000. Χρησιμοποιησαμε 2 διαφορετικες micro SDs για κάθε περιβάλλον εκτέλεσης, καθώς για την περίπτωση του SnapFaas χρειάστηκε όπως αναφέραμε και προηγουμένως να εφαρμόσουμε patch στο υπάρχων kernel.

Για τις δύο περίπτωσεις (vanilla Firecracker & SnapFaas), το λειτουργικό σύστημα που χρησιμοποιησαμε για τον host είναι Ubuntu 20.04.4 και έκδοση πυρήνα 5.4.0-1080-raspi.

Για τον guest, το λειτουργικό σύστημα είναι Alpine Linux 3.10 με έκδοση πυρήνα 4.20.0. Να σημειώσουμε ότι για τον guest, το λειτουργικό σύστημα και ο πυρήνας είναι κοινά και

για τα 2 συστήματα που συγκρίνουμε καθώς χρησιμοποιούν το ίδιο RootFS (με μόνη διαφορά το τροποποιημένο runtime workload για την αρχικοποίηση της γλώσσας και την φόρτωση του application module) το οποίο στην βάση του έχει δημιουργηθεί με το Alpine 3.10 docker container.

4.4 Benchmarking functions

Εφαρμόσαμε 6 συναρτήσεις στην Python που αντιπροσωπεύουν μια ποικιλία εφαρμογών στις Faas εφαρμογές: επεξεργασία κειμένου, ήχου και εικόνας καθώς και IoT εφαρμογές. Πολλές από αυτές τις εφαρμογές έχουν εξαρτώμενες βιβλιοθήκες, συμπεριλαμβανομένων native βιβλιοθηκών και εκτελέσιμων αρχείων. Παραδείγματος χάριν, η Python3 εφαρμογή thumbnail εξαρτάται από το πακέτο Pillow, το οποίο με την σειρά του χρειάζεται το libjpeg Python module. Υπενθυμίζουμε ότι όλες αυτές οι εξαρτήσεις των εφαρμογών συμπεριλαμβάνονται στο σύστημα αρχείων της εφαρμογής (Application file system - AppFS) το οποίο είναι αποθηκευμένο και φορτώνεται μέσα στο diff στιγμιότυπο (function snapshot). Τα χαρακτηριστικά, οι εξαρτήσεις καθώς και το format των requests/responses των συναρτήσεων που εφαρμόσαμε, είναι καταγεγραμένα στον Πίνακα 2.

Characteristics / Benchmarks	Description	Python modules dependencies	Request	Response
audio-fingerprint	Δημιουργεί μια ψηφιακή περίληψη (digital summary) ενός ηχητικού σήματος που περιέχεται σε ένα αρχείο ήχου	pyacoustid, audioread	{'audio': audio name}	{'duration': audio duration, 'fingerprint': fingerprint}
thumbnail	Τροποποιεί μια εικόνα ώστε να περιέχει μια μικρογραφία (thumbnail) του εαυτού της	Pillow	{'img': image name, 'size': target thumbnail size}	{'serialized_img': serialized thumbnail}

ocr	Ένα οπτικό εργαλείο αναγνώρισης χαρακτήρων. Αναγνωρίζει και διαβάζει κείμενο ενσωματωμένο σε εικόνες	Tesseract OCR engine	{'img': image name}	{'success': 0, 'error': error msg} or {'success': 1, 'text': tesseract output}
image-enhance	Βελτιστοποιεί μια εικόνα στις παραμέτρους του χρώματος, της αντίθεσης, της φωτεινότητας και της σαφήνειας	Pillow	{"img": "libertybell.jpg", "operation": "sharpness/brightness/contrast/color", "factor": int}	{'success': 1, 'img_str': img_str} or {'success': 0, 'error': 'unknown enhancement mode'}
sentiment-analysis	Δίνει μια ανάλυση συναισθήματος του παρεχόμενου κειμένου	nltk textblob	{'bayes': null, 'analyse': paragraph to be analyzed} or {'analyse': paragraph to be analyzed}	{'results': list of results for each sentence, 'num_sentences': number of sentences}

lorem	Δημιουργεί ένα τυχαίο κείμενο όπου το πρώτο γράμμα είναι κεφαλαίο και η κάθε πρόταση τελειώνει σε τελεία ή ερωτηματικό (lorem ipsum text)	lorem	{"value": "lorempy2-1"}	{'request': obj, 'body': lorem.sentence()}

Πίνακας 2: Περιγραφή και χαρακτηριστικά των συναρτήσεων που εφαρμόστηκαν

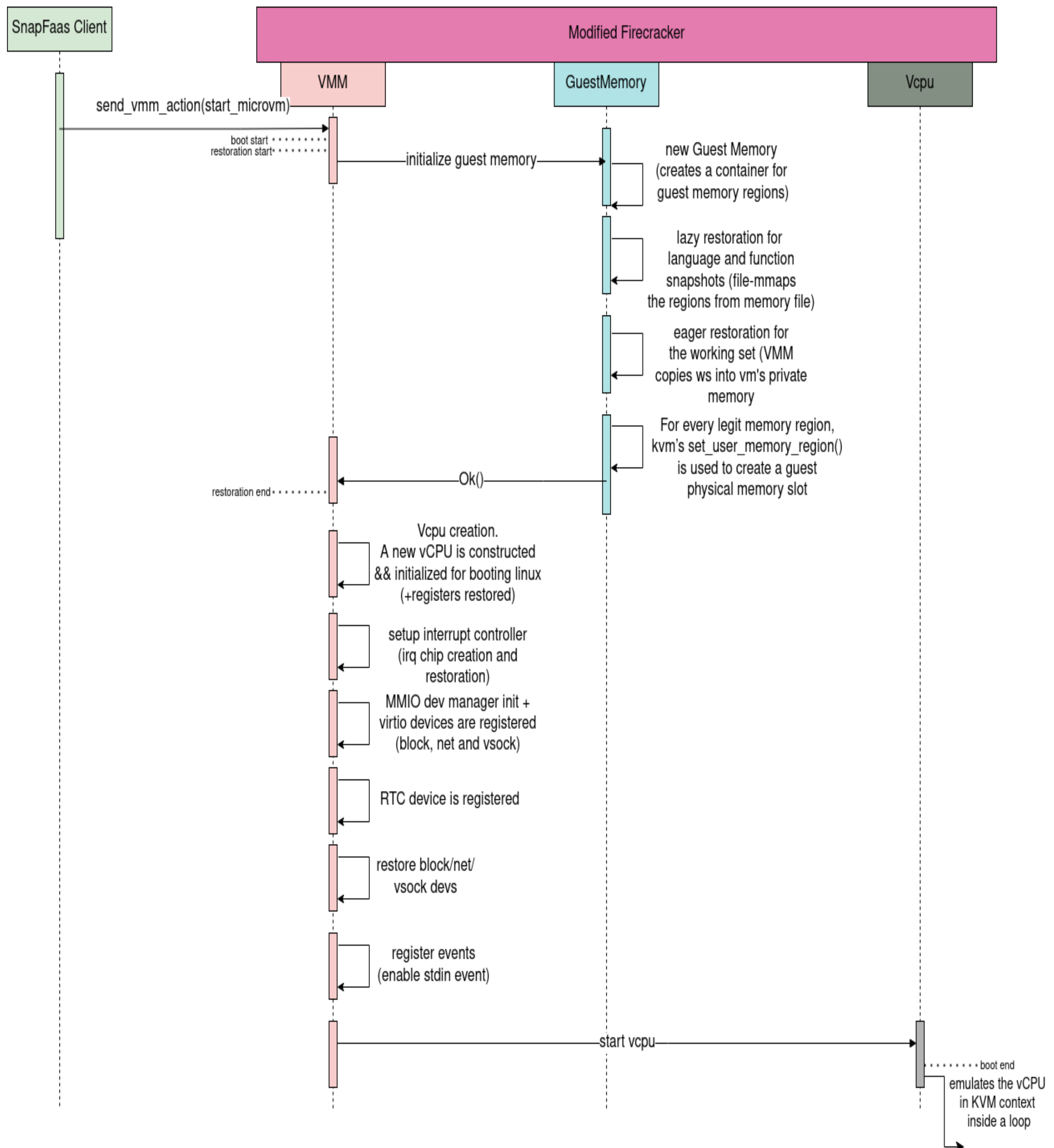
4.5 Ανάλυση Φάσεων Εκτέλεσης

Στο κομμάτι των μετρήσεων, έχουμε χωρίσει την διαδικασία του να δημιουργηθεί ένα microVM και να εξυπηρετηθεί ένα αίτημα, από άκρη σε άκρη (End-to-End), σε 5 διαφορετικά μέρη, προσπαθώντας να κρατήσουμε το κάθε σκέλος όσο το δυνατόν πιο όμοιο για κάθε ένα από τα δύο εργαλεία, ώστε να έχει νόημα η σύγκριση μεταξύ τους.

4.5.1 1ο Μέρος: Αποκατάσταση Μνήμης (Memory restoration)

- ***SnapFaas***

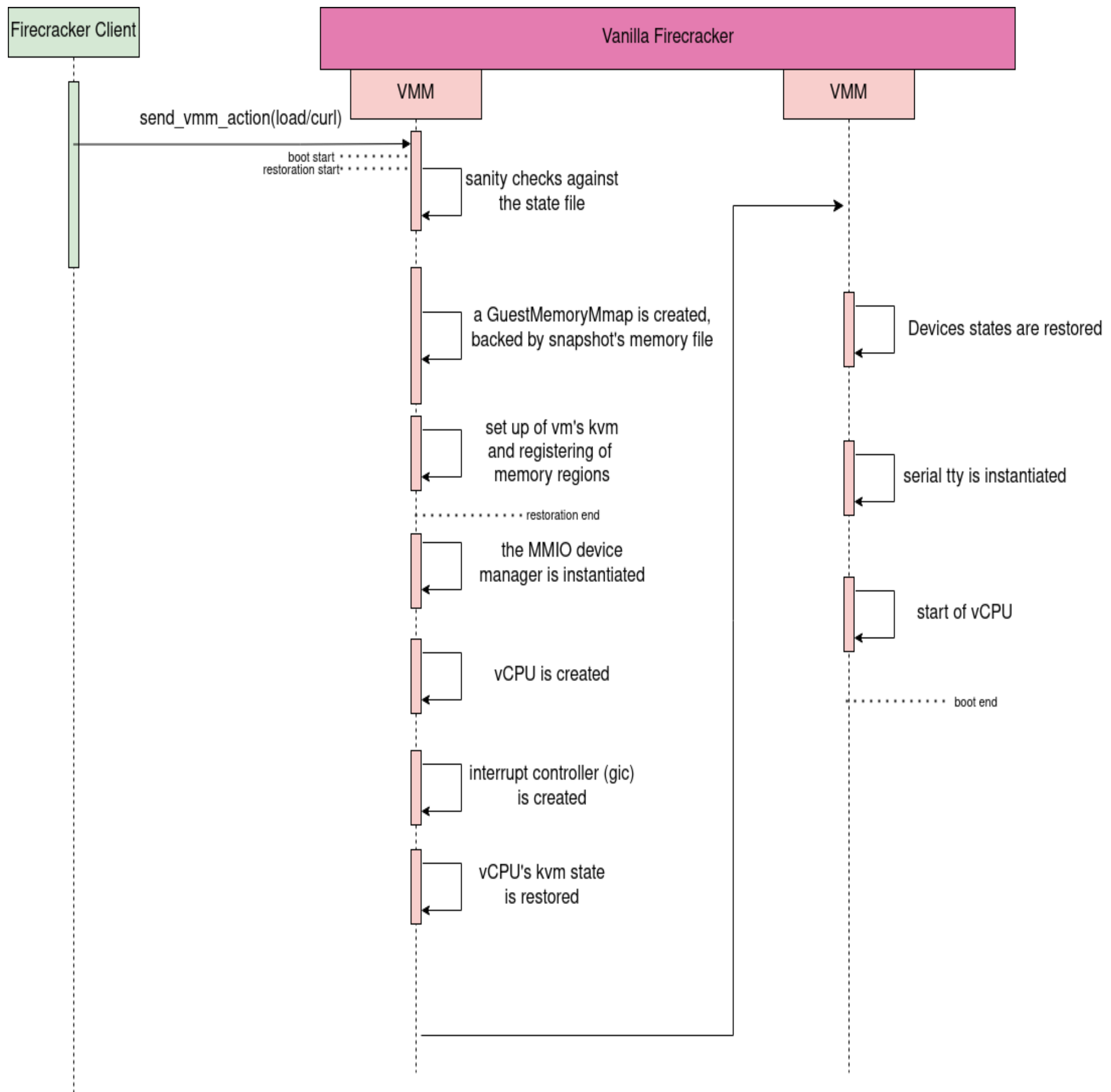
Δοκιμάσαμε όλες τις διαφορετικές διαθέσιμες τεχνικές του memory restoration που μας προσφέρει το SnapFaas. Παρατηρήσαμε ότι η χαρτογράφηση (memory mapping) των αρχείων μνήμης του base αλλά και του diff snapshot (lazy restoration) και η αντιγραφή (χρησιμοποιώντας το system call readv) κάθε σελίδας μνήμης από το ενεργό σύνολο σελίδων της κάθε συνάρτησης (working set - the set of pages that have been accessed during function execution) μέσα στην ιδιωτική μνήμη (private memory) του εκάστοτε microVM (eager restoration), μας δίνει τους καλύτερους δυνατούς χρόνους όσον αφορά την End-to-End περάτωση του αιτήματος. Για τον λόγο αυτό θα χρησιμοποιήσουμε καθολικά τον συγκεκριμένο τρόπο αποκατάστασης μνήμης για τα πειράματα μας με το εργαλείο του SnapFaas. Τα αναλυτικά βήματα της εκτέλεσης μεταξύ των διαφορετικών δομών του SnapFaas, παρουσιάζονται στο Σχήμα 24.



Σχήμα 24: Τα βήματα του memory restoration & του booting phase για το SnapFaas

- ***Firecracker***

Στο vanilla Firecracker εργαλείο, η μνήμη αποκαθίσταται αποκλειστικά μέσω του file memory mapping. Πιο συγκεκριμένα, το συμπτιγμένο αρχείο μνήμης (η διαδικασία συμπτυξης των 2 αρχείων – base και diff memory files) έχει δημιουργηθεί offline μέσω του rebase-snap εργαλείου που μας προσφέρει το Firecracker, όπως έχουμε αναφέρει. Τα αναλυτικά βήματα παρουσιάζονται στο Σχήμα 25.



Σχήμα 25: Τα βήματα του memory restoration & του booting phase για το Firecracker

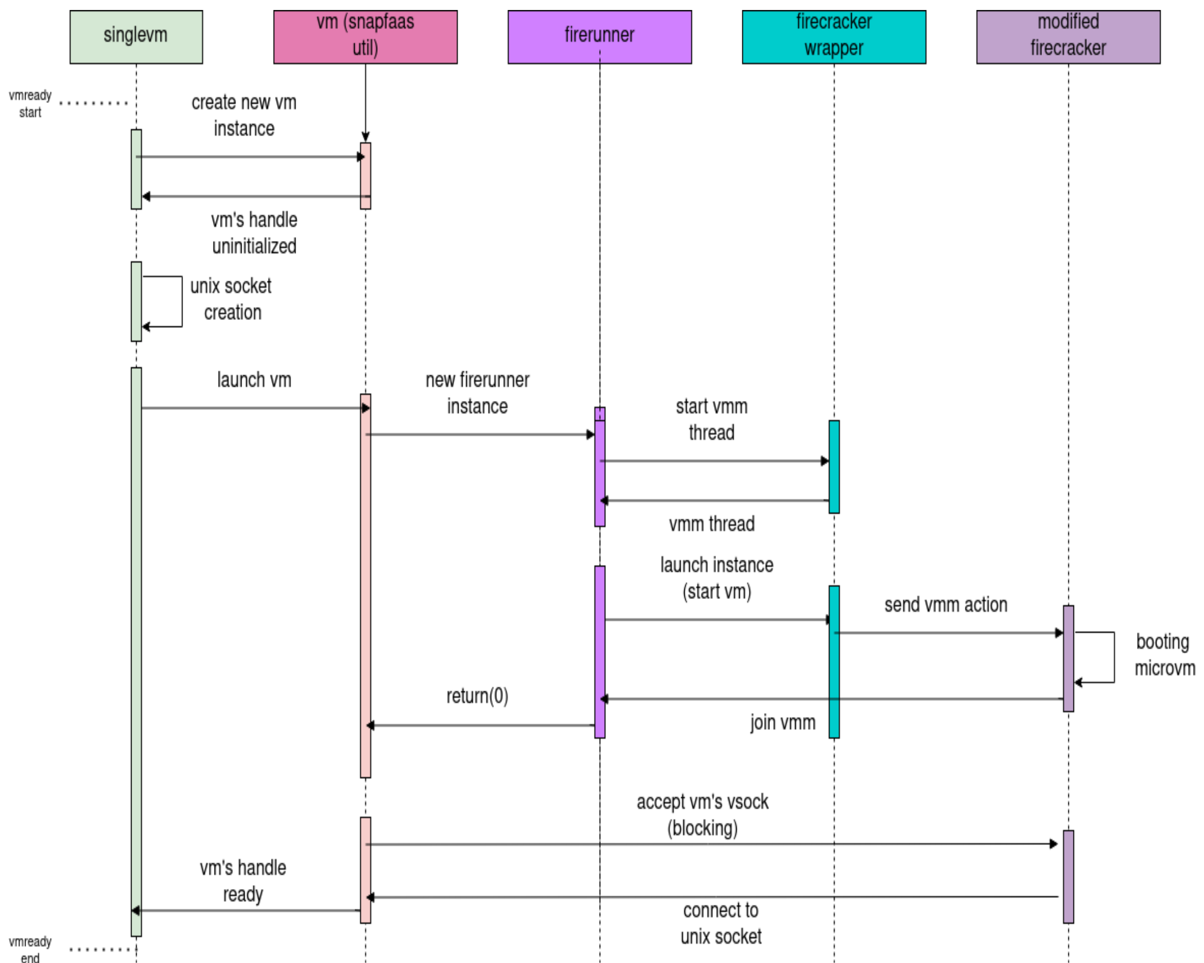
4.5.2 2ο Μέρος: Εκκίνηση μικρο-εικονικής μηχανής (Booting)

Το συγκεκριμένο μέρος, ξεκινάει: 1) από την στιγμή που αρχίσει η εκτέλεση της συνάρτησης *start_microvm()* του Virtual Machine Manager, για την περίπτωση του SnapFaas και 2) μόλις σταλθεί το API request */snapshot/load*, για την περίπτωση του Firecracker. Και για τις δύο περιπτώσεις, περιλαμβάνει μέχρι και το σημείο αμέσως πριν ξεκινήσει η εξομοίωση του εικονικού επεξεργαστή (Vcpu emulation). Διευκρινιστικά, η επαναφορά της μνήμης είναι μέρος της φάσης εκκίνησης, όπως φαίνεται και στα Σχήματα 20 & 21.

4.5.3 3ο Μέρος: Ετοιμότητα μικρο-εικονικής μηχανής (VM Ready)

- ***SnapFaas***

Η συγκεκριμένη φάση ξεκινάει από την στιγμή που ο SnapFaas wrapper, μέσω του singlevm εκτελέσιμου, στείλει την εντολή να δημιουργηθεί μια εικονική μηχανή, μέχρι η εικονική αυτή μηχανή να είναι έτοιμη να δεχτεί αιτήματα (της εκάστοτε συνάρτησης) προς επεξεργασία. Η φάση εκκίνησης (booting phase) συμπεριλαμβάνεται. Αναλυτικά τα βήματα παρουσιάζονται στο Σχήμα 26.

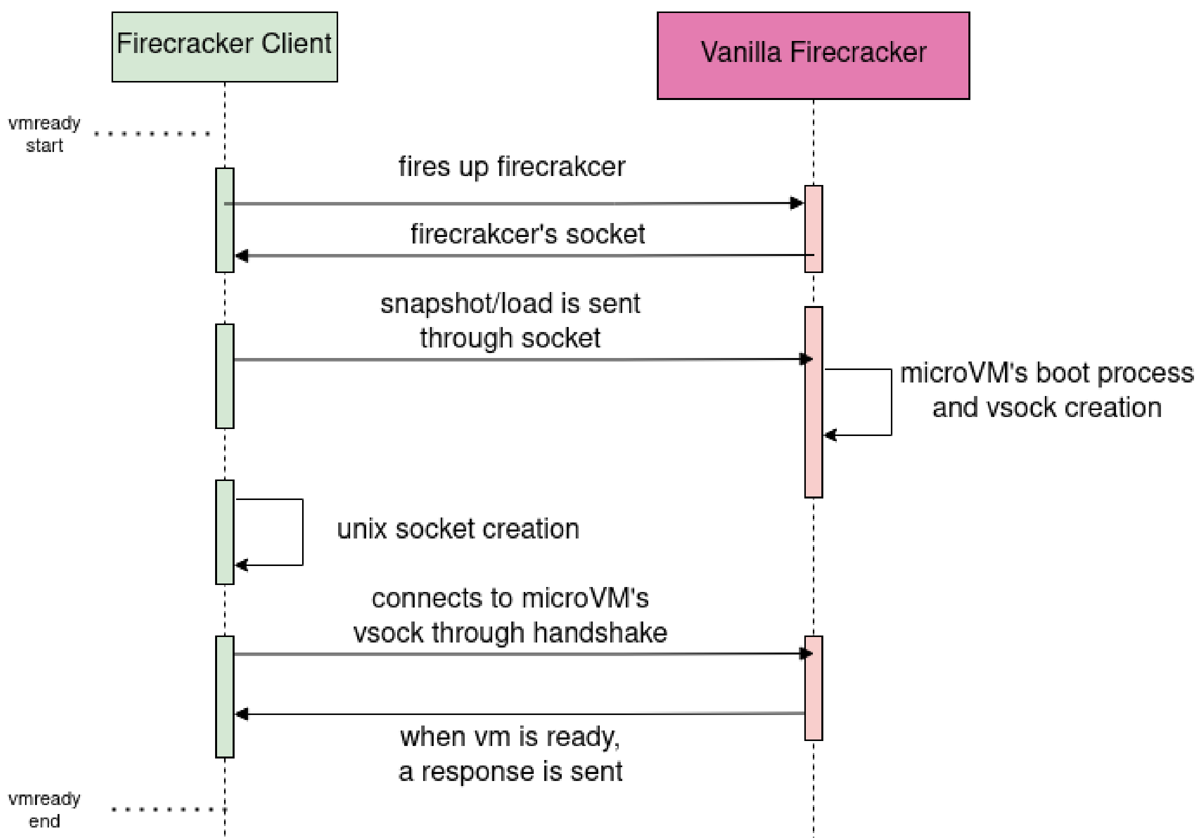


Σχήμα 26: Τα βήματα του VM Ready phase για το SnapFaas

- ***Firecracker***

Όπως αναφέραμε και προηγουμένως, για να καταφέρουμε να συγκρίνουμε τα δύο πρωτόκολλα δημιουργίας εικονικής μηχανής και εκτέλεσης ενός αιτήματος, φτιάξαμε τον client, που αναλύσαμε στο Κεφάλαιο 4, μέσω του οποίου μπορούμε να επαναφέρουμε ένα microVM και να στείλουμε ένα request προς εξυπηρέτηση. Η φάση vmready στην περίπτωση του Firecracker, ξεκινάει ακριβώς πριν στείλουμε το API request μέσω του curl

εργαλείου στο socket που δρα ως τον server του Firecracker. Μέσω του unix socket συνδεόμαστε στο virtual socket του microVM και αφού λάβουμε τα δεδομένα της απάντησης (μέσα από την διαδικασία “χειραψίας” που αναφεραμε προηγουμενως), η εικονική μηχανή είναι έτοιμη να δεχτεί αιτήματα προς επεξεργασία. Τα αναλυτικά βήματα παρουσιάζονται στο Σχήμα 27.



Σχήμα 27: Τα βήματα του VM Ready phase για το Firecracker

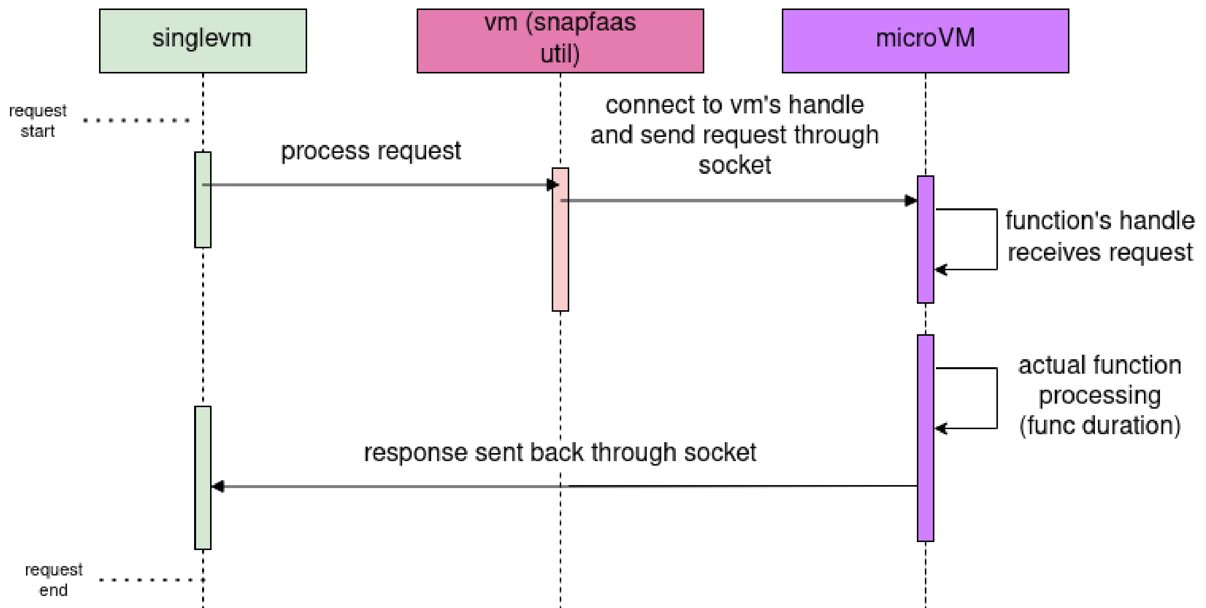
4.5.4 4ο Μέρος: Ολοκλήρωση αιτήματος (Request Returned)

Ο συγκεκριμένος χρόνος περιλαμβάνει τα βήματα που απαιτούνται για να ολοκληρωθεί από άκρη σε άκρη η εξυπηρέτηση ενός αιτήματος. Πιο συγκεκριμένα, από την στιγμή που θα στείλουμε το αίτημα μέσω της εγκατεστημένης σύνδεσης μεταξύ client (είτε του SnapFaas είτε του Firecracker) και microVM, εώς ότου λάβουμε πίσω την απάντηση, μέσω του socket.

4.5.5 5ο Μέρος: Διάρκεια εκτέλεσης (Function)

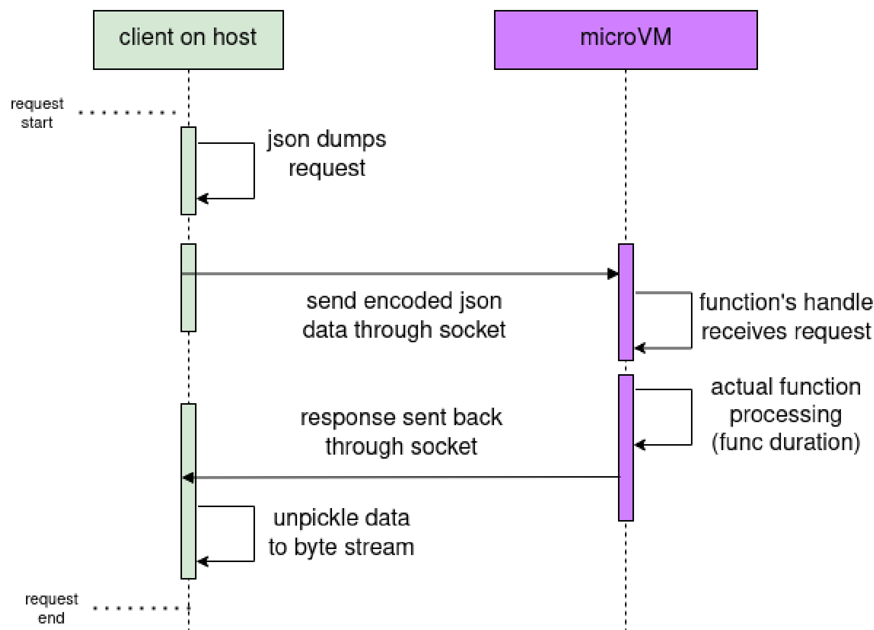
Πρόκειται για τον καθαρό χρόνο εκτέλεσης της συνάρτησης μέσα στο περιβάλλον της μικρο-εικονικής μηχανής. Είναι υποσύνολο του Request Returned phase. Και για τις δύο αυτές φάσεις, παρουσιάζονται τα αναλυτικά βήματα στα Σχήματα 24 (SnapFaas) & 25 (Firecracker).

- ***SnapFaas***



Σχήμα 28: Τα βήματα του Request Returned phase για το SnapFaas

- ***Firecracker***



Σχήμα 29: Τα βήματα του Request Returned phase για το Firecracker

Απο τα παραπάνω διαγράμματα καταλαβαίνουμε ότι η βασική διαφορά των 2 εργαλείων είναι η προεργασία που απαιτείται (στην περίπτωση του Firecracker) στα δεδομένα του

request για να μπορέσουν να σταλούν στην εκάστοτε συνάρτηση και να μετατραπούν σε αναγνώσιμη μορφή, όπως περιγράψαμε στο Κεφάλαιο 4.2.

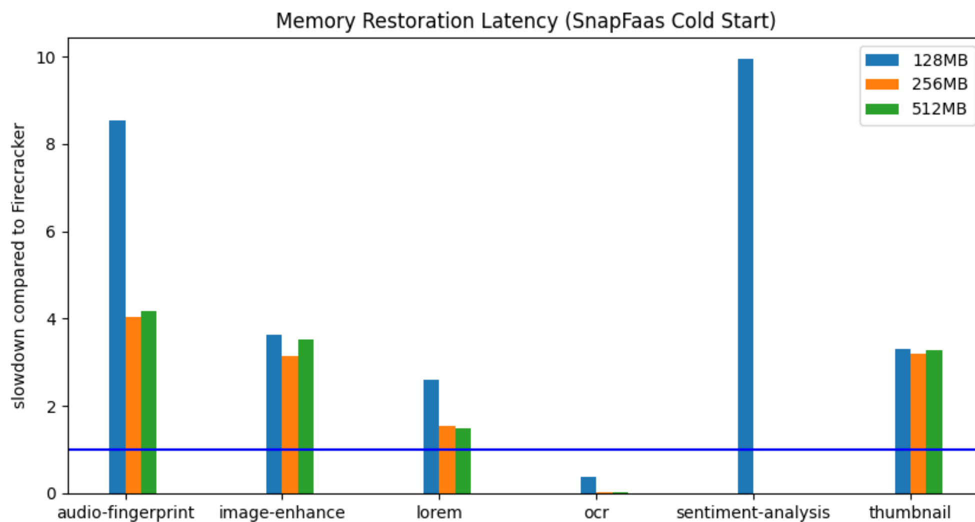
5 Πειραματικό Σκέλος

Για τις εξής συναρτησεις: audio-fingerprint, image-enhance, lorem, ocr, sentiment-analysis και thumbnail, έχουμε πάρει τους χρόνους για τις προαναφερόμενες φάσεις, κρατώντας τον μέσο όρο απο 100 εκτελέσεις.

5.1 Cold-Start

Για να αντιληφθούμε καλύτερα τα πλεονεκτήματα του SnapFaas, στα πειράματα μας έχουμε εφαρμόσει την τεχνική της "κρύας" εκκίνησης (cold-start) στην επαναφορά του microVM. Για να το πετύχουμε αυτό, κάθε φορά που στέλνουμε το αίτημα μας η μικρο-εικονική μηχανή πρέπει να μην είναι σε κατάσταση αναμονής (idle state) και όλη η διαδικασία ανέγερσης (η ολοκλήρωση της vmready φάσης), να πραγματοποιείται από το μηδέν. Επιπλέον, πριν από κάθε εκτέλεση "καθαρίζουμε" την page cache του host, με τον τρόπο που περιγράψαμε στο κεφάλαιο 4.2, ώστε να μην υπάρχει memory footprint της εικονικής μηχανής στον host μας και να μπορεί να φανεί με ξεκάθαρο τρόπο η καθυστέρηση που προσθέτει η αποκατάσταση της μνήμης στην δημιουργία της εικονικής μηχανής, καθώς και η εκτέλεση της συνάρτησης. Όλα τα πειράματά μας τα εκτελέσαμε για 3 διαφορετικά μεγέθη μνήμης: 128, 256 & 512 MB. Η μπλε μπάρα αντιστοιχίζεται στο 128MB, η πορτοκαλί στο 256MB και η πράσινη στο 512MB. Όλες οι μετρήσεις για όλα τα μεγέθη είναι κανονικοποιημένες ως προς το cold-start του Firecracker.

5.1.1 Cold-Start Memory restoration phase 128MB/256MB/512MB

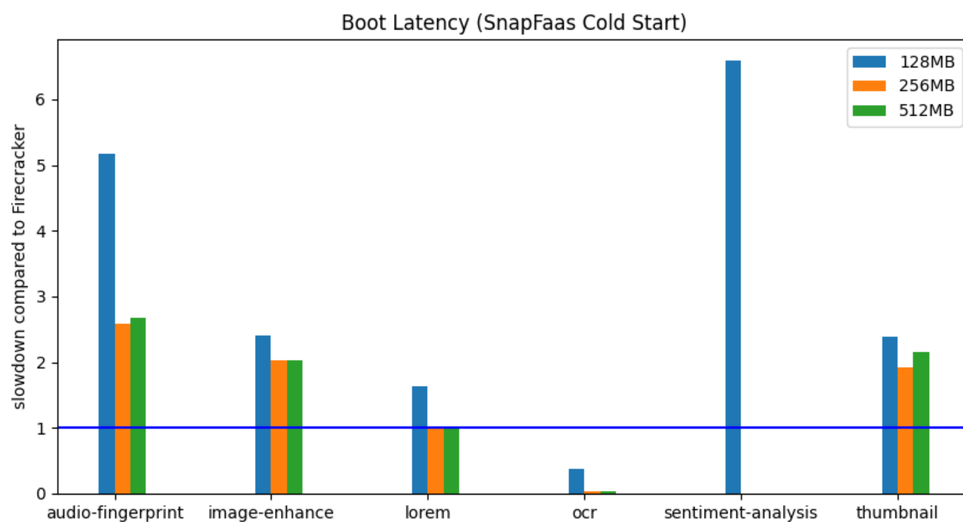


Σχήμα 30: Memory restoration Latency, SnapFaas slowdowns for both language and function snapshots restored lazily and working set eagerly

Παρατηρούμε στο Σχήμα 30, ότι για σχεδόν όλες τις συναρτήσεις το SnapFaas είναι αρκετά πιο αργό για την φάση του memory restoration. Για να καταλάβουμε τον λόγο που συμβαίνει αυτό πρέπει να επισημάνουμε τον διαφορετικό τρόπο που συμβαίνει η αποκατάσταση μνήμης σε κάθε εργαλείο εκτέλεσης. Ενώ και τα δύο συστήματα που συγκρίνουμε χρησιμοποιούν την τεχνική του file memory mapping για το restoration την μνήμης, το Firecracker χρειάζεται να χαρτογραφήσει ένα μοναδικό αρχείο μνήμης το οποίο

είναι η σύμπτυξη των δύο στιγμιοτύπων (language and function) και έχει μέγεθος όσο και η μνήμη του microVM. Η διαδικασία της σύμπτυξης των δύο αρχείων γίνεται offline μεσω του rebase-snap tool του Firecracker και ο χρόνος αυτός συνυπολογίζεται στο χρόνο που εξετάζουμε (για την ocr συνάρτηση που έχει μεγάλη διάρκεια εκτέλεσης, το merging διαρκεί πολύ με αποτέλεσμα να είναι η μοναδική εφαρμογή που παρατηρείται speedup για το SnapFaas). Στο SnapFaas εν αντιθέση, η συγκεκριμένη διαδικασία γίνεται δύο φορές, μία για κάθε αρχείο μνήμης που περιλαμβάνεται σε κάθε στιγμιότυπο (language and function snapshots). Όπως και στην περίπτωση του vanilla Firecracker, κάθε ένα από τα δύο αρχεία έχει μέγεθος όσο και το μέγεθος της μνήμης του microVM. Επιπλέον, σημαντική καθυστέρηση, στην περίπτωση του SnapFaas, προσδίδει η αντιγραφή του working set της κάθε συνάρτησης στην private memory του microVM μέσω του system call *readv*. Αναφορικά με τα διαφορετικά μεγέθη μνήμης της εκάστοτε μηχανής, παρατηρούμε ότι για σχεδόν όλες τις συναρτήσεις, τα μεγέθη 256MB και 512MB έχουν μικρότερη χρονική καθυστέρηση σε σχέση με το Firecracker, απ'ότι τα miroVMs με μεγεθος 128MB. Αυτό συμβαίνει καθώς για τα μεγέθη 256 & 512 MB, τα working set αρχεία είναι μικρότερου μεγέθους, με αποτέλεσμα η αντιγραφή των σελίδων μνήμης να διαρκεί λιγότερο.

5.1.2 Cold-Start Boot phase 128MB/256MB/512MB

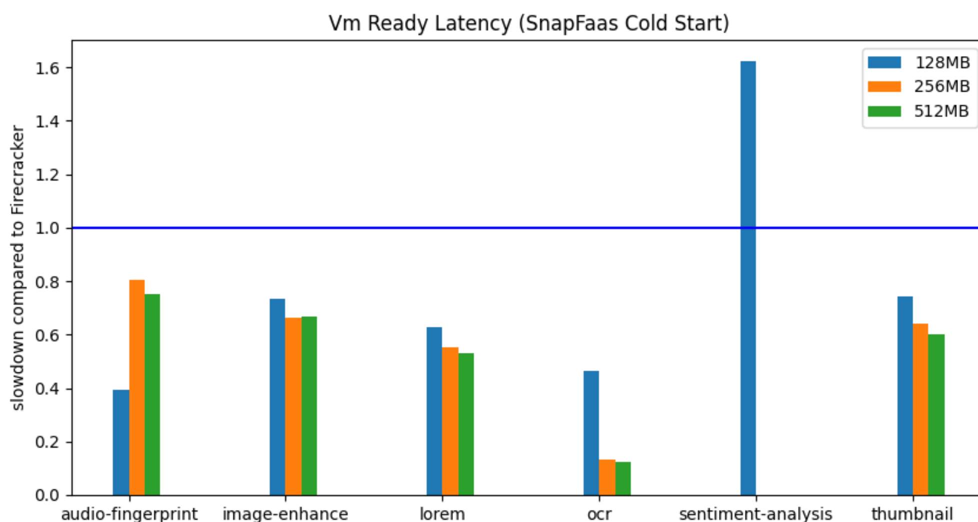


Σχήμα 31: Boot Latency, SnapFaas slowdowns for both language and function snapshots restored lazily and working set eagerly

Ένα μέρος της καθυστέρησης για το boot phase προσδίδεται απο την αποκατάσταση της μνήμης. Πιο συγκεκριμένα, για το Firecracker η αποκατάσταση μνήμης καταλαμβάνει γύρω στο 65% του συνολικού χρόνου εκκίνησης, ενώ για το SnapFaas το αντίστοιχο ποσοστό είναι της τάξεως του 90%. Τα υπόλοιπα απαραίτητα κομμάτια για την εκκίνηση των δύο microVMs, όπως η δημιουργία και η επαναφορά των διαφόρων συσκευών και του εικονικού επεξεργαστή, χρειάζονται λιγότερο χρόνο στο SnapFaas για να ολοκληρωθούν. Αυτό οφείλεται στο γεγονός ότι το SnapFaas αρχικοποιεί λιγότερα components. Η χρονική καθυστέρηση λοιπόν που προκύπτει για το SnapFaas, περιμένουμε να είναι μικρότερη σε σχέση με την αποκατάσταση μνήμης, όπως και συμβαίνει με βάση το Σχήμα 31. Επιπλέον, παρατηρούμε ότι και στην φάση εκκίνησης, στις περισσότερες περιπτώσεις για 256 & 512 MB, η χρονική καθυστέρηση μειώνεται σε σχέση με το Firecracker. Τα διαφορετικά slowdowns μεταξύ των συναρτήσεων, οφείλονται και πάλι στην χρονική καθυστέρηση που

προκύπτει από την αποκατάσταση της μνήμης. Πιο συγκεκριμένα, οι συναρτήσεις για τις οποίες τα slowdowns είναι πιο μικρά σε σύγκριση με το Firecracker (lorem, ocr) έχουν μικρότερα αρχεία μνήμης *memory_dump* και *WS_dump*, οπότε η αποκατάσταση της μνήμης στο SnapFaas είναι πιο γρήγορη.

5.1.3 Cold-Start VM Ready phase 128MB/256MB/512MB

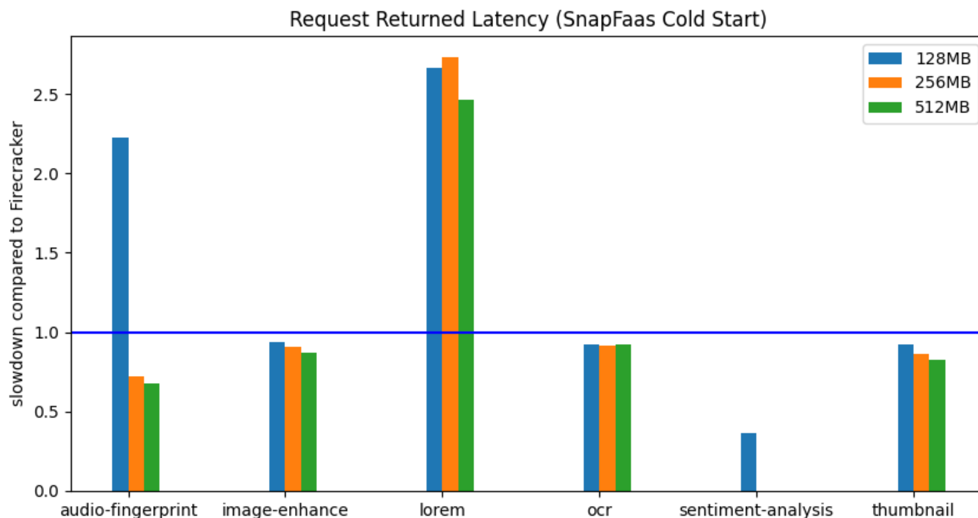


Σχήμα 32: VMReady Latency, SnapFaas slowdowns for both language and function snapshots restored lazily and working set eagerly

Για τον συνολικό χρόνο προετοιμασιας μιας εικονικής μηχανής, παρατηρούμε ότι το SnapFaas είναι πιο γρήγορο για όλες σχεδόν τις περιπτώσεις, με βάση το Σχήμα 32. Η μοναδική περίπτωση που αυτό διαφοροποιείται (sentiment-analysis - 128MB) σχετίζεται με τα memory pages που αντιγράφονται, καθώς όπως αναφέραμε και στην φάση εκκίνησης, σε αυτή την περίπτωση είναι περισσότερες (50 MB file length), με αποτέλεσμα η επαναφορά της μνήμης (που περιλαμβάνεται στο συνολικό χρόνο ετοιμασίας του microVM) να προσδίδει μεγαλύτερη καθυστέρηση. Ως αποτέλεσμα το memory restoration για την συγκεκριμένη συνάρτηση έχει μεγαλύτερη διάρκεια που αποτυπώνεται και σε αυτή την μέτρηση.

Παρολα αυτα, λόγω της αντιγραφής του working set στην μνήμη της εικονικής μηχανής, θα περιμέναμε η συγκεκριμένη μέτρηση να είναι ταχύτερη για τον Firecracker. Πρέπει όμως για τον χρόνο του Firecracker, να λάβουμε υπόψιν ότι συνυπολογίζεται και η εγκατάσταση της σύνδεσης μεταξύ των δύο sockets που περιλαμβάνει το handshake που έχουμε αναφέρει. Επιπλέον, στις περισσότερες των περιπτώσεων το working set των συναρτήσεων (δηλαδή το πραγματικό πλήθος σελίδων μνήμης στις οποίες γίνεται πρόσβαση κατά την εκτέλεση της συνάρτησης) και το οποίο γίνεται copy μέσω της readv κλήσης συστήματος, είναι μικρό σε μέγεθος με αποτέλεσμα η αντιγραφή του να μην προσδίδει τόσο μεγάλη καθυστέρηση.

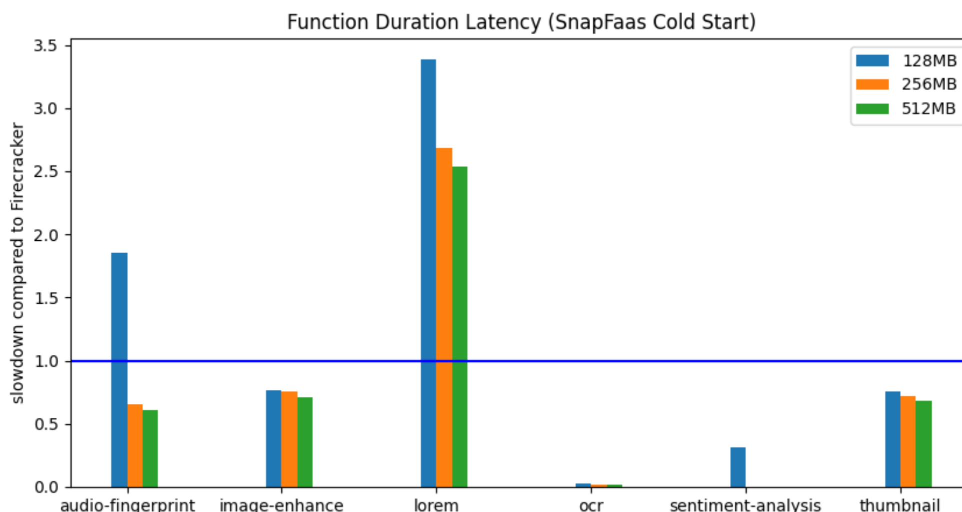
5.1.4 Cold-Start Request Returned phase 128MB/256MB/512MB



Σχήμα 33: Request Returned Latency, SnapFaas slowdowns for both language and function snapshots restored lazily and working set eagerly

Στο Σχήμα 33 παρουσιάζονται τα slowdowns του SnapFaas σε σχέση με το Firecracker, για την χρονική διάρκεια που χρειάζεται ένα request να σταλεί και να πάρουμε πίσω το response της FaaS εφαρμογής. Στις περισσότερες περιπτώσεις, το SnapFaas είναι ταχύτερο. Εξαιτίας του eagerly restored working set, έχουμε λιγότερα Copy-on-Write page faults και έτσι μικρότερη καθυστέρηση κατα την διάρκεια εκτέλεσης των συναρτήσεων. Για του λόγου το αληθές, η συνάρτηση sentiment-analysis όπου οι σελίδες που γίνονται copy απο τον δισκο πριν ξεκινήσει η εκτέλεση είναι περισσότερες, ο χρόνος εξυπηρέτησης του αιτήματος είναι ακόμα πιο ταχύς.

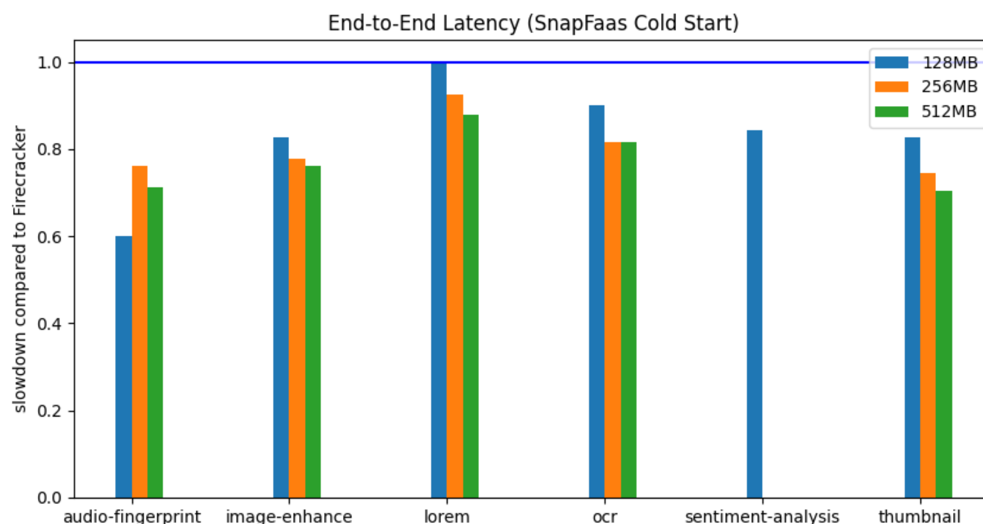
5.1.5 Cold-Start Duration phase 128MB/256MB/512 MB



Σχήμα 34: Duration Latency, SnapFaas slowdowns for both language and function snapshots restored lazily and working set eagerly

Στο Σχήμα 34, έχουμε αναπαραστήσει τα speedups του SnapFaas αποκλειστικά για την διάρκεια εκτέλεσης των συναρτήσεων, χωρίς να λαμβάνεται υπόψη ο χρόνος που απαιτείται για την μεταφορά των δεδομένων του request μέσω του socket connection, καθώς και ο αντίστοιχος χρόνος για την μεταφορά των δεδομένων του response πίσω στον host (και την μετατροπή τους σε human readable μορφή). Εδώ λοιπόν φαίνεται πιο καθαρά το speedup του SnapFaas στον χρόνο εκτέλεσης της συνάρτησης, Για παράδειγμα, η ocr συνάρτηση η οποία έχει μεγάλο χρόνο εκτέλεσης, στην περίπτωση του SnapFaas που εκμεταλλεύεται την χρήση του working set (με αποτέλεσμα οι "χρήσιμες" σελίδες μνήμης να υπάρχουν στην cache του microVM) και των τροποποιημένων σελίδων που περιλαμβάνονται στο αρχείο μνήμης του diff snapshot, έχει πολύ μεγάλο speedup σε σύγκριση με το Firecracker. Το speedup αυτό παρουσιάζεται μειωμένο στον αντίστοιχο χρόνο της περάτωσης του request, καθώς για το parsing των δεδομένων του response στον host, η Python του Firecracker client χρησιμοποιεί το pickle module που είναι πιο αποδοτικό σε σχέση με το αντίστοιχο εργαλείο της Rust του SnapFaas client. Επιπλέον αξίζει να σημειώσουμε ότι σε εφαρμογές μικρής διάρκειας εκτέλεσης (π.χ. lorem), η επίδραση του working set μειώνεται με αποτέλεσμα το SnapFaas να είναι πιο αργό.

5.1.6 Cold-Start End-to-End 128MB/256 MB/512 MB



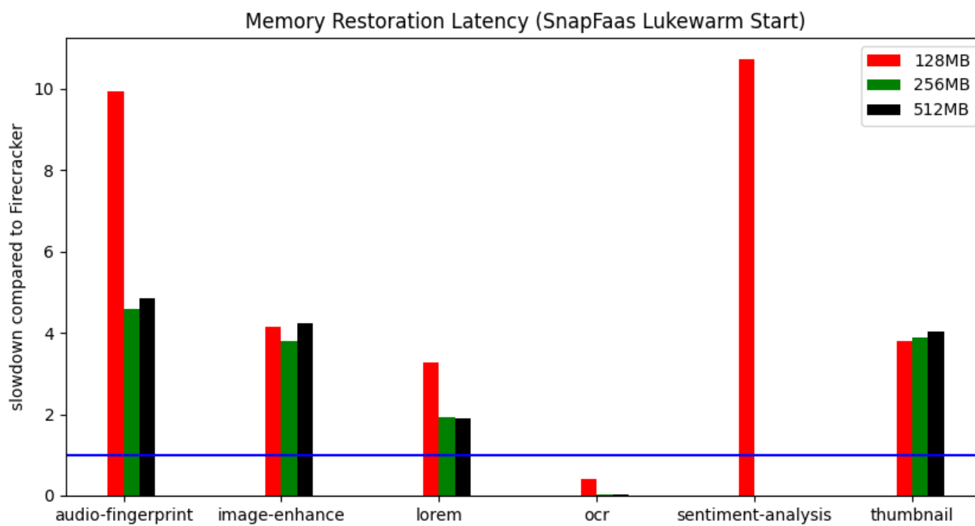
Σχήμα 35: End-to-End Latency, SnapFaas slowdowns for both language and function snapshots restored lazily and working set eagerly

Στο Σχήμα 35, παρουσιάζονται τα End-to-End αποτελέσματα για την σύγκριση του SnapFaas με το Firecracker. Για να καταλήξουμε σε αυτές τις μετρήσεις προσθέσαμε τον χρόνο που απαιτείται για να είναι έτοιμο το microVM (VMReady phase) και το χρόνο περάτωσης του αιτήματος (Request Returned phase). Παρατηρούμε λοιπόν ότι για όλες τις συναρτήσεις και για όλα τα sizes των microVMs, έχουμε speedup του SnapFaas σε σύγκριση με το Firecracker. Με την χρήση του working set και του layering των διαφόρων στρωμάτων στιγμιότυπων (function & language snapshots) επωφελούμαστε για την συνολική διάρκεια που απαιτείται ώστε ένα αίτημα προς κάποιο microVM που είναι ανενεργό, να εξυπηρετηθεί.

5.2 Lukewarm-Start

Εκτελέσαμε τα ίδια πειράματα για την αποκατάσταση του microVM, χωρίς να "καθαρίζουμε" την page cache του host πριν από κάθε εκτέλεση. Αυτό έχει σαν αποτέλεσμα να υπάρχει το memory footprint του microVM στον host. Όλα τα πειράματά μας τα εκτελέσαμε για 3 διαφορετικά μεγέθη μνήμης: 128, 256 & 512 MB. Η κόκκινη μπάρα αντιστοιχίζεται στα 128MB, η πράσινη στα 256MB και η μαύρη στα 512MB. Όλες οι μετρήσεις για όλα τα μεγέθη είναι κανονικοποιημένες ως προς το lukewarm-start του Fiecracker.

5.2.1 Lukewarm-Start Memory restoration phase 128MB/256MB/512MB

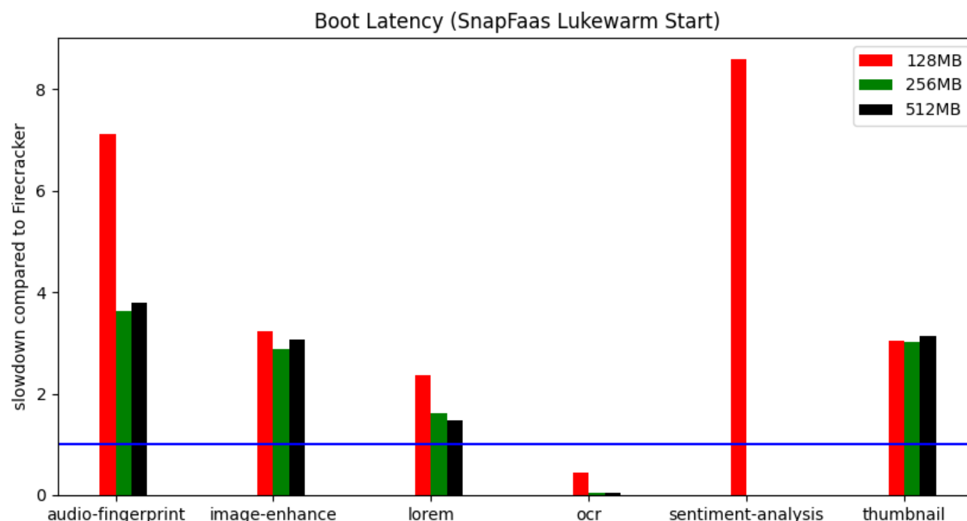


Σχήμα 36: Memory Restoration Latency, SnapFaas slowdowns for both language and function snapshots restored lazily and working set eagerly

Παρατηρούμε στο Σχήμα 36, ότι για όλες τις συναρτήσεις το SnapFaas και είναι πιο αργό για την φάση του memory restoration. Όπως και στην περίπτωση του cold-start, τα 2 εργαλεία που συγκρίνουμε χρησιμοποιούν την τεχνική του file memory mapping για το restoration την μνήμης και το Firecracker χρειάζεται να αντιστοιχίσει ένα μοναδικό αρχείο μνήμης ενώ στο SnapFaas η συγκεκριμένη διαδικασία γίνεται δύο φορές, μία για κάθε αρχείο μνήμης που περιλαμβάνεται σε κάθε στιγμιότυπο (language and function snapshots). Και στην περίπτωση του lukewarm-start, για το SnapFaas υπάρχει επιπλέον καθυστέρηση λόγω της αντιγραφής του working set της κάθε συνάρτησης στην ιδιωτική μνήμη της εικονικής μηχανής μέσω του system call *readv*. Αναφορικά με τα διαφορετικά μεγέθη μνήμης του εκάστοτε microVM, παρατηρούμε ότι για σχεδόν όλες τις συναρτήσεις, τα μεγέθη 256MB και 512MB έχουν μικρότερη χρονική καθυστέρηση σε σχέση με το Firecracker, απ'ότι τα miroVMs με μεγεθος 128MB. Η διαφορά της φάσης του memory restoration στο lukewarm-start σε σχέση με το cold-start, είναι η ακόμα μεγαλύτερη καθυστέρηση που προσδίδεται στο SnapFaas για όλα τα benchmarks. Αυτό είναι αποτέλεσμα της μείωσης του χρόνου που διαρκεί η αποκάσταση της μνήμης στο vanilla Firecracker. Αυτό συμβαίνει, καθώς πλέον, το state file του στιγμιότυπου του vanilla βρίσκεται στην page cache της μνήμης του microVM με αποτέλεσμα να μην εκτελείται

Input και Output (καθώς ολόκληρη η μνήμη στο vanilla εργαλείο έχει ήδη χαρτογραφηθεί στην μνήμη).

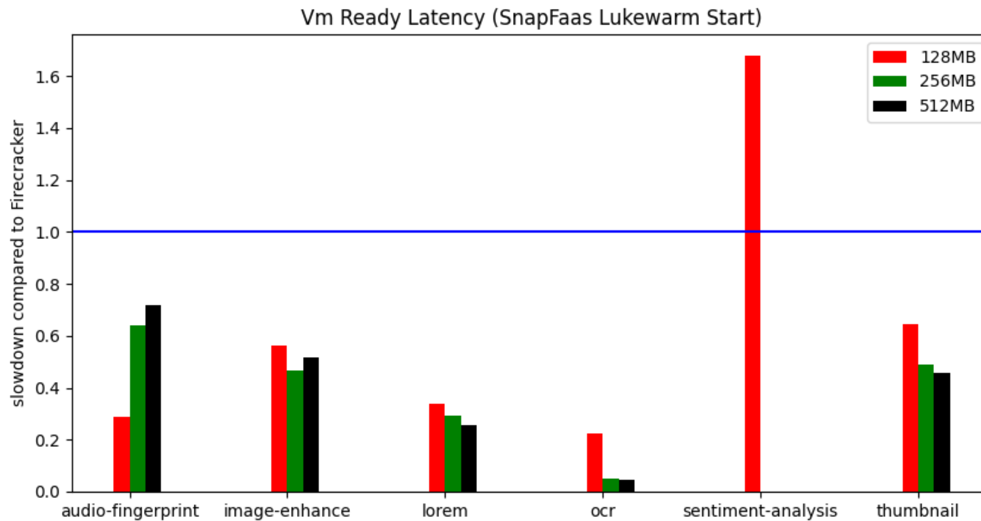
5.2.2 Lukewarm-Start Boot phase 128MB/256MB/512MB



Σχήμα 37: Boot Latency, SnapFaas slowdowns for both language and function snapshots restored lazily and working set eagerly

Στην περίπτωση του Boot phase, με βάση το Σχήμα 37, παρατηρούμε slowdowns του SnapFaas για όλες τις συναρτήσεις και πάλι όπως στην περίπτωση του cold-start, η οποία οφείλεται στην μεγαλύτερη χρονική διάρκεια που χρειάζεται το SnapFaas για την αποκατάσταση της μνήμης που περιλαμβάνεται στο Boot phase (πλην της ocr εφαρμογής, για τον ίδιο λόγο που αναφέραμε και στην περίπτωση του cold-start). Τα υπόλοιπα απαραίτητα κομμάτια για την εκκίνηση των δύο microVMs, όπως η δημιουργία και η επαναφορά των διαφόρων συσκευών και του εικονικού επεξεργαστή, χρειάζονται λιγότερο χρόνο στο SnapFaas για να ολοκληρωθούν. Επιπλέον, παρατηρούμε ότι και στην φάση εκκίνησης, στις περισσότερες περιπτώσεις για 256 & 512 MB, η χρονική καθυστέρηση μειώνεται σε σχέση με το Firecracker. Οι συναρτήσεις για τις οποίες τα slowdowns είναι πιο μικρά σε σύγκριση με το Firecracker (lorem, ocr) έχουν μικρότερα αρχεία μνήμης *memory_dump* και *WS_dump*, οπότε η αποκατάσταση της μνήμης στο SnapFaas και τελικά η χρονική διάρκεια του boot phase είναι πιο γρήγορη.

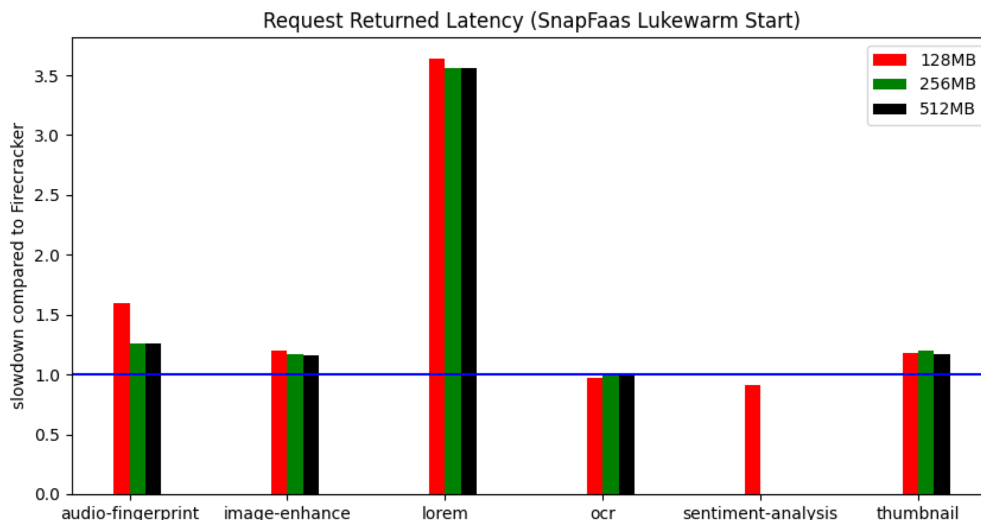
5.2.3 Lukewarm-Start VM Ready phase 128MB/256MB/512MB



Σχήμα 38: VMReady Latency, SnapFaas slowdowns for both language and function snapshots restored lazily and working set eagerly

Για τον συνολικό χρόνο προετοιμασιας μιας εικονικής μηχανής, παρατηρούμε ότι το SnapFaas είναι πιο γρήγορο για όλες σχεδόν τις περιπτώσεις, με βάση το Σχήμα 38. Η μοναδική περίπτωση που αυτό διαφοροποιείται, είναι η sentiment-analysis συνάρτηση για 128MB, όπως και στην περίπτωση του cold-start.
Παρατηρούμε ότι το speedup για το SnaFaas είναι ακόμα μεγαλύτερο σε σχέση με το cold-start, καθώς με το να μην "καθαρίζουμε" την page cache, υπάρχουν σελίδες μνήμης στον guest που δεν χρειάζονται μεταφορά από τον δίσκο και βοηθούν στην πιο γρήγορη επαναφορά του microVM.

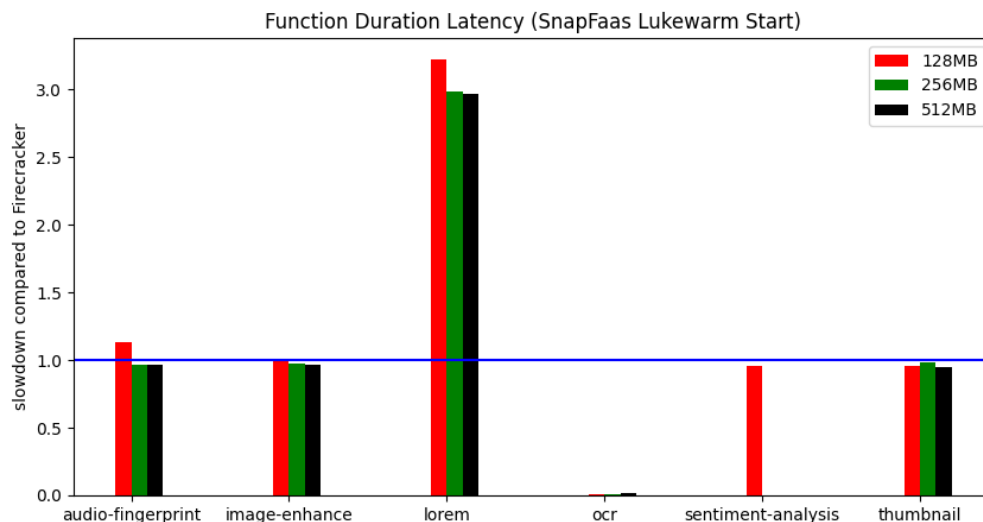
5.2.4 Lukewarm-Start Request Returned phase 128MB/256MB/512MB



Σχήμα 39: Request Returned Latency, SnapFaas slowdowns for both language and function snapshots restored lazily and working set eagerly

Στο Σχήμα 39 παρατηρούμε ότι το SnapFaas έχει μικρά slowdowns σε σχέση με το Firecracker, για την χρονική διάρκεια που χρειάζεται ένα request να σταλεί και να πάρουμε πίσω το response της Faas εφαρμογής. Η αιτία πίσω από τα συγκεκριμένα αποτελέσματα βρίσκεται και πάλι στην page cache που έχει παραμείνει ως έχει κατά την διάρκεια των επαναλαμβανόμενων εκτελέσεων, καθώς το αντεγραμμενο working set δεν βελτιώνει τόσο το SnapFaas σε σχέση με το Firecracker, αφού πλέον και οι περισσότερες σελίδες μνήμης που χρειάζεται το microVM του Firecracker, βρίσκονται ήδη στην page cache.

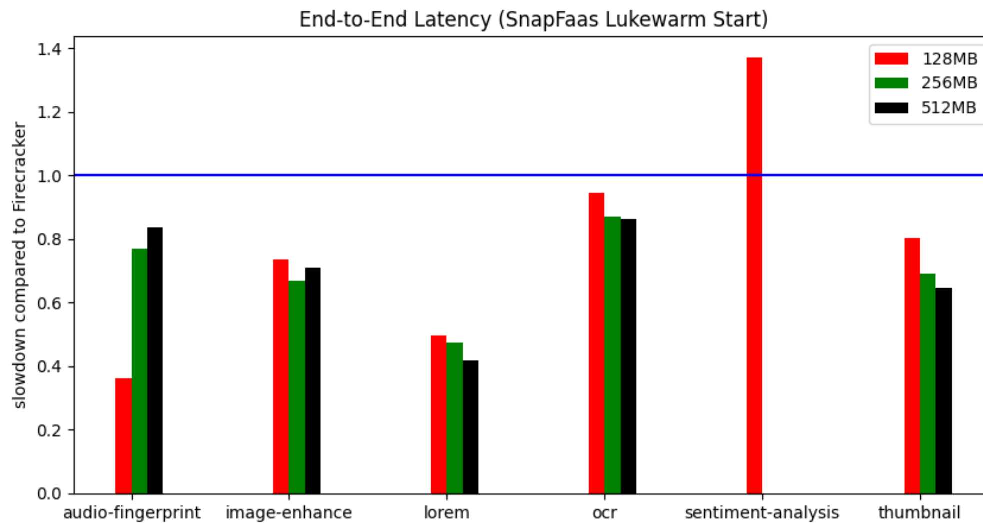
5.2.5 Lukewarm-Start Duration phase 128MB/256MB/512MB



Σχήμα 40: Duration Latency, SnapFaas slowdowns for both language and function snapshots restored lazily and working set eagerly

Στο Σχήμα 40, έχουμε αναπαραστήσει τα speedups/slowdowns του SnapFaas αποκλειστικά για την διάρκεια εκτέλεσης των συναρτήσεων Εδώ λοιπόν φαίνεται πιο καθαρά οι συγκρίσεις για τον χρόνο εκτέλεσης της συνάρτησης, Για παράδειγμα, η ocr συνάρτηση η οποία έχει μεγάλο χρόνο εκτέλεσης, στην περίπτωση του SnapFaas που εκμεταλλεύεται την χρήση του working set (με αποτέλεσμα οι "χρήσιμες" σελίδες μνήμης να υπάρχουν στην cache του microVM) και των τροποποιημένων σελίδων που περιλαμβάνονται στο αρχείο μνήμης του diff snapshot, έχει πολύ μεγάλο speedup σε σύγκριση με το Firecracker. Για τις υπόλοιπες συναρτήσεις, το SnapFaas είναι πιο κοντά στους χρόνους εκτέλεσης της συνάρτησης σε σχέση με το Firecracker, καθώς όπως αναφέραμε και στην περίπτωση του VM Ready phase, το SnapFaas δεν είναι τόσο αποδοτικό λόγω της ύπαρξης των χρησιμοποιούμενων σελίδων μνήμης στην page cache.

5.2.6 Lukewarm-Start End-to-End 128MB/256MB/512MB



Σχήμα 41: End-to-End Latency, SnapFaas slowdowns for both language and function snapshots restored lazily and working set eagerly

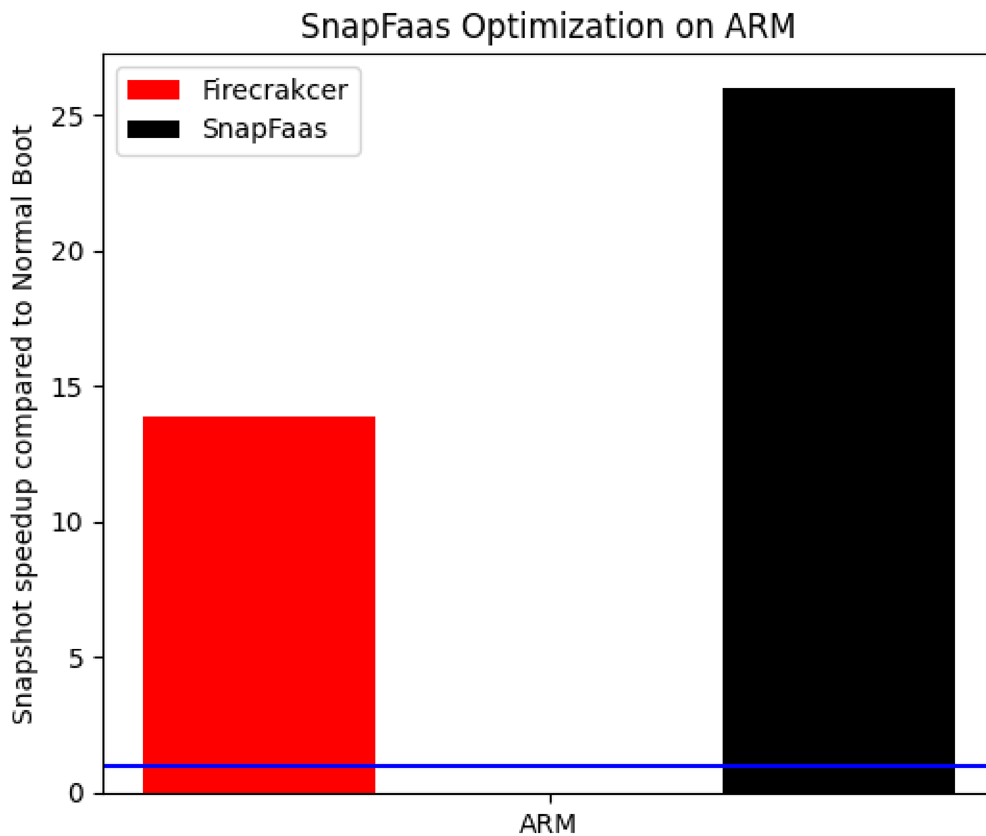
Στο Σχήμα 41, παρουσιάζονται τα End-to-End αποτελέσματα για την σύγκριση του SnapFaas με το Firecracker, για το lukewarm-start. Παρατηρούμε λοιπόν ότι για όλες τις συναρτήσεις και για όλα τα sizes των microVMs (πλην της συνάρτησης sentiment-analysis - 128MB), έχουμε speedup του SnapFaas σε σύγκριση με το Firecracker. Με την χρήση του working set και του layering των διαφόρων στρωμάτων στιγμιότυπων (function & language snapshot) επωφελούμαστε για την συνολική διάρκεια που απαιτείται ώστε ένα αίτημα προς κάποιο microVM που είναι ανενεργό, να εξυπηρετηθεί ακόμα και στην περίπτωση που η page cache δεν "αδειάζεται" ενδιάμεσα των προωθήσεων των αιτημάτων στα microVMs.

6 Συμπεράσματα

Ολοκληρώνοντας την εργασία, είναι σημαντικό σε αυτό το κεφάλαιο να πραγματοποιήσουμε μια σύνοψη των όσων παρουσιάστηκαν αλλά και τα συμπεράσματα που μπορούν να προκύψουν από αυτά.

6.1 Συζήτηση - Σύνοψη

Μέσα απο την εργασία έγινε αντιληπτό πόσο σημαντικό είναι για το περιβάλλον του Serverless η μείωση της καθυστέρησης που προσδίδει το cold-start, καθώς είναι απαραίτητο για την καλύτερη διαχείριση τον πόρων που παρέχονται από τους εκάστοτε cloud providers. Διερευνήσαμε την χρήση snapshots για την βελτίωση των cold-starts, σαν μια ευρέως μελετημένη τεχνική από την τεχνολογική κοινότητα. Τα cold-starts προσθέτουν χρονοβόρα καθυστέρηση και περιορίζουν την χρήση των FaaS εφαρμογών σημαντικά όταν οι χρόνοι εκτέλεσης τους είναι πολύ χαμηλοί. Παρουσιάσαμε εκτενώς το SnapFaas, ένα εργαλείο στιγμιότυπων για FaaS εφαρμογές, βασισμένο σε Linux Virtual Machines. Αφορμώμενοι απο την ανοδική τάση των IoT συσκευών που συμβαδίζουν με την event-based αρχιτεκτονική των Function-as-a-Service εφαρμογών και χρησιμοποιούν κατά κύριο λόγο ARM-based επεξεργαστές, μελετήσαμε και πετύχαμε την μεταφορά του SnapFaas εργαλείου σε συστήματα ARM αρχιτεκτονικής. Το SnapFaas, εκμεταλλευόμενο την σχεδίαση των διαχωρισμένων base & diff snapshots, καθώς και την χρησιμοποίηση του working set (που είχε παρουσιαστεί απο το REAP πρώτη φορά), καταφέρνει να πετύχει πολύ καλούς χρόνους στο cold-start, αλλά και στο lukewarm-start, με την λιγότερη δυνατή χρήση storage (εφόσον το base snapshot μπορεί να διαμοιραστεί ανάμεσα στα microVMs με την ίδια γλώσσα εκτέλεσης). Πιο συγκεκριμένα, και σε σύγκριση με το vanilla Firecracker που είναι ένα συμπαγές εργαλείο χρήσης στιγμιότυπων, πετυχαίνει μέχρι και 1,7 φορές γρηγορότερη End-to-End εξυπηρέτηση αιτημάτων στα cold-starts και μέχρι και 2,5 φορές γρηγορότερη εξυπηρέτηση αιτημάτων στις lukewarm-start εκτελέσεις. Στο Σχήμα 42 παρουσιάζονται τα speedups της αποκατάστασης μέσω snapshot, σε σύγκριση με το normal boot, για το vanilla Firecracker και για το SnapFaas, όπου επιβεβεώνονται τα παραπάνω νούμερα.



Σχήμα 42: Firecracker vs SnapFaas optimization on ARM

6.2 Περιορισμοί και Επόμενα βήματα

Η χρήση στιγμιότυπων για την γρήγορη επαναφορά των microVMs δημιουργεί κάποιες ανησυχίες για την ασφάλεια των εικονικών αυτών μηχανών και άρα και του host μέσα στον οποίο εδρεύουν. Πιο συγκεκριμένα, microVMs που επαναφέρονται από το ίδιο base στιγμιότυπο, μοιράζονται ένα μεγάλο μέρος της μνήμης τους. Αυτό βελτιώνει την απόδοση των microVMs αλλά τα αφήνει "ευάλωτα" σε επιθέσεις (π.χ. από ένα κακόβουλο φορτίο αιτήματος). Μελέτες του παρελθόντος [34] [35], έχουν δείξει ότι η εκ νέου τυχαιοποίηση (re-randomizing) της μνήμης και του κώδικα μπορούν να εφαρμοστούν προσθέτοντας λογικές καθυστερήσεις. Με παρόμοιο τρόπο, η χρησιμοποίηση των Copy-on-Write διαμοιραζόμενων σελίδων μνήμης για την μνήμη του πυρήνα και του language runtime της Python, εισάγει τον κίνδυνο χρήσης καναλιών χρονισμού βασισμένα στην κρυφή μνήμη (cache based timing channels) μεταξύ συναρτήσεων στο ίδιο εικονικό μηχάνημα [36].

Πιο "φανεροί" περιορισμοί με τους οποίους ήρθαμε και αντιμέτωποι κατά την διάρκεια της ανάπτυξης και υλοποίησης του SnapFaas σε Arm αρχιτεκτονικές και οι οποίοι χρήζουν αντιμετώπισης, ειναι οι παρακάτω:

→ Όλα τα microVMs που τρέξαμε για τα διάφορα benchmarks έχουν τον περιορισμό ότι λειτουργούν με έναν μόνο εικονικό επεξεργαστή (1 Vcpu). Υπάρχει δυνατότητα επέκτασης του κώδικα του VMM του SnapFaas για να υποστηρίξει microVMs με παραπάνω επεξεργαστές το οποίο αναμένεται να βελτιώσει περαιτέρω τους End-to-End χρόνους εξυπηρέτησης.

→ Με την υπάρχουσα υλοποίηση κάθε base snapshot υποστηρίζει ένα συγκεκριμένο μέγεθος μνήμης του microVM. Για παράδειγμα, ένα Python snapshot που έχει δημιουργηθεί με βάση ένα microVM των 128MB μνήμης, δεν μπορεί να χρησιμοποιηθεί για να ξεκινήσει μια συνάρτηση των 256MB. Η υποστήριξη ενός μεγέθους microVM απαιτεί ένα ξεχωριστό στιγμιότυπο βάσης. Παρολ’αυτά, ο συγκεκριμένος περιορισμός επιδέχεται διόρθωσης χρησιμοποιώντας γνωστές τεχνικές memory ballooning.

Βιβλιογραφία

[1] “arm64: dirty memory check, enable soft dirty for arm64 - Patchwork,” patchwork.kernel.org. Available:https://patchwork.kernel.org/project/linux-arm-kernel/patch/1512029649-61312-1-git-send-email-bin.lu@arm.com/

[2] “Documentation – Arm Developer,” developer.arm.com. http://infocenter.arm.com/help/topic/com.arm.doc.den0001c/DEN0001C_principles_of_arm_memory_maps.pdf

[3] “Documentation – Arm Developer,” developer.arm.com. https://developer.arm.com/documentation/ddi0595/2021-12/AArch32-Registers/MPIDR--Multiprocessor-Affinity-Register

[4] “vsock(7) - Linux manual page,” www.man7.org. https://www.man7.org/linux/man-pages/man7/vsock.7.html

[5] VSOCK device limitation, https://github.com/firecracker-microvm/firecracker/blob/main/docs/snapshotting/snapshot-support.md#vsock-device-limitation

[6] Balloon device with Firecracker, https://github.com/firecracker-microvm/firecracker/blob/main/docs/ballooning.md

[7] Vanilla Firecracker Client, https://github.com/sampalekos96/firecracker-client

[8] Docker Overview, https://docs.docker.com/get-started/overview/

[9] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting. In Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS ’20, page 467–481, New York, NY, USA, 2020. Association for Computing Machinery.

[10] Google gVisor, https://gvisor.dev/

[11] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. SOCK: Rapid task provisioning with serverless-optimized containers. In 2018 USENIX Annual Technical Conference (USENIX ATC 18), pages 57–70, Boston, MA, 2018. USENIX Association.

[12] Anil Madhavapeddy and David J. Scott. Unikernels: Rise of the virtual library operating system. Queue, 11(11):30:30–30:44, December 2013.

[13] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight virtualization for serverless applications. In the 17th USENIX Symposium on Networked Systems Design

and Implementation (NSDI 20), pages 419–434, Santa Clara, CA, February 2020. USENIX Association.

[14] Kvm ioctls on Rust,
https://docs.rs/kvm-ioctls/latest/kvm_ioctls/

[15] Rootfs setup on Firecracker,
https://github.com/firecracker-microvm/firecracker/blob/main/docs/rootfs-and-kernel-setup.md#creating-a-rootfs-image

[16] OpenRC init system,
https://wiki.gentoo.org/wiki/OpenRC

[17] Alpine Linux,
https://alpinelinux.org/

[18] Virtqueues and virtio ring: How the data travels,
https://www.redhat.com/en/blog/virtqueues-and-virtio-ring-how-data-travels

[19] B-Tree struct,
https://en.wikipedia.org/wiki/B-tree

[20] Linux's Page Tables,
https://docs.kernel.org/mm/page_tables.html

[21] Scatter - Gather I/O,
https://www.oreilly.com/library/view/linux-system-programming/0596009585/ch04.html

[22] ARM Generic Interrupt Controller Architecture version 2.0 - Architecture Specification,
https://developer.arm.com/documentation/ihi0048/latest/

[23] Kernel Docs, ARM Virtual Generic Interrupt Controller v2,
https://www.kernel.org/doc/html/v5.9/virt/kvm/devices/arm-vgic.html
[24] Len Brown, Anil Keshavamurthy, David Shaohua Li, Robert Moore, Venkatesh Pallipadi and Luming Yu, ACPI in Linux. In the Intel Open Source Technology Center, pages 59-76, 2005

[25] Raspberry Pi 4 Tech Specs,
https://www.raspberrypi.com/products/raspberry-pi-4-model-b/specifications/

[26] I/O Ports and I/O Memory,
https://www.oreilly.com/library/view/linux-device-drivers/0596000081/ch02s05.html

[27] Real Time Clock (RTC) Drivers for Linux,
https://docs.kernel.org/admin-guide/rtc.html

[28] Soft - Dirty PTEs,
https://www.kernel.org/doc/Documentation/vm/soft-dirty.txt

[29] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. Benchmarking, analysis, and optimization of serverless function snapshots. In Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2021, New York, NY, USA, 2021. Association for Computing Machinery.

[30] Yue Tan, David Lio, Nanqinqin Li and Amit Levy, How Low Can You Go? Practical cold-start performance limits in FaaS. In arXiv:2109.13319v1 [cs.DC], 27 Sep 2021. Princeton University

[31] SnapFaas Code on Github,
https://github.com/faasten/faasten
[32] AWS Lambda,
https://aws.amazon.com/lambda/

[33] Azure Functions,
https://azure.microsoft.com/en-us/products/functions/

[34] David Bigelow, Thomas Hobson, Robert Rudd, William Streilein, and Hamed Okhravi. Timely Rerandomization for Mitigating Memory Disclosures. In Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, CCS '15, page 268–279, New York, NY, USA, 2015. Association for Computing Machinery.

[35] Kangjie Lu, Stefen Nurnberger, Backes Michael, and Wenke Lee. How to Make ASLR Win the Clone Wars: Runtime Re-Randomization. In The Network and Distributed System Security Symposium 2016, NDSS '16, 2016.

[36] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, You, Get off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds. In Proceedings of the 16th ACM Conference on Computer and Communications Security, CCS '09, page 199–212, New York, NY, USA, 2009. Association for Computing Machinery.

[37] Modified SnapFaas Firecracker code,
https://github.com/sampalekos96/firecracker/tree/snapshot