Εθνικο Μετσοβιο Πολυτεχνειο
Σχολη Ηλεκτρολογων Μηχανικων και Μηχανικων Υπολογιστων
Τομεας Τεχνολογιας Πληροφορικης και Υπολογιστων

Υλοποίηση αποδοτικού μηχανισμού ελαστικής μνήμης για το Firecracker VMM

Διπλωματικη Εργασια

του

ΒΑΣΙΛΕΙΟΥ-ΠΑΝΑΓΙΩΤΗ ΜΗΛΙΓΓΑ

Επιβλέπων: Νεκτάριος Κοζύρης
Καθηγητής ΕΜΠ

Αθήνα, Νοέμβριος 2025

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

Υλοποίηση αποδοτικού μηχανισμού ελαστικής μνήμης για το Firecracker VMM

Διπλωματικη Εργασια

του

ΒΑΣΙΛΕΙΟΥ-ΠΑΝΑΓΙΩΤΗ ΜΗΛΙΓΓΑ

Επιβλέπων: Νεκτάριος Κοζύρης
Καθηγητής ΕΜΠ

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 5/11/2025.

(Υπογραφή)	*(Υπογραφή)*	*(Υπογραφή)*
.....	.....	.....
Νεκτάριος Κοζύρης	Γεώργιος Γκούμας	Διονύσιος Πνευματικάτος
Καθηγητής ΕΜΠ	Καθηγητής ΕΜΠ	Καθηγητής ΕΜΠ

Αθήνα, Νοέμβριος 2025


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

ΔΗΛΩΣΗ ΜΗ ΛΟΓΟΚΛΟΠΗΣ ΚΑΙ ΑΝΑΛΗΨΗΣ ΠΡΟΣΩΠΙΚΗΣ ΕΥΘΥΝΗΣ

Με πλήρη επίγνωση των συνεπειών του νόμου περί πνευματικών δικαιωμάτων, δηλώνω ενυπογράφως ότι είμαι αποκλειστικός συγγραφέας της παρούσας Πτυχιακής Εργασίας, για την ολοκλήρωση της οποίας κάθε βοήθεια είναι πλήρως αναγνωρισμένη και αναφέρεται λεπτομερώς στην εργασία αυτή. Έχω αναφέρει πλήρως και με σαφείς αναφορές, όλες τις πηγές χρήσης δεδομένων, απόψεων, θέσεων και προτάσεων, ιδεών και λεκτικών αναφορών, είτε κατά κυριολεξία είτε βάσει επιστημονικής παράφρασης. Αναλαμβάνω την προσωπική και ατομική ευθύνη ότι σε περίπτωση αποτυχίας στην υλοποίηση των ανωτέρω δηλωθέντων στοιχείων, είμαι υπόλογος έναντι λογοκλοπής, γεγονός που σημαίνει αποτυχία στην Πτυχιακή μου Εργασία και κατά συνέπεια αποτυχία απόκτησης του Τίτλου Σπουδών, πέραν των λοιπών συνεπειών του νόμου περί πνευματικών δικαιωμάτων. Δηλώνω, συνεπώς, ότι αυτή η Πτυχιακή Εργασία προετοιμάστηκε και ολοκληρώθηκε από εμένα προσωπικά και αποκλειστικά και ότι, αναλαμβάνω πλήρως όλες τις συνέπειες του νόμου στην περίπτωση κατά την οποία αποδειχθεί, διαχρονικά, ότι η εργασία αυτή ή τμήμα της δεν μου ανήκει διότι είναι προϊόν λογοκλοπής άλλης πνευματικής ιδιοκτησίας.

(Υπογραφή)

.....
Βασίλειος-Παναγιώτης Μηλίγγας
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

5/11/2025

Περίληψη

Η ελαστική μνήμη έχει αποκτήσει ιδιαίτερη σημασία στο υπολογιστικό νέφος. Επιτρέπει την προσαρμογή της μνήμης μιας εικονικής μηχανής κατά απαίτηση χωρίς την ανάγκη για επανεκκίνηση, προσφέροντας αποδοτικότερη αξιοποίηση πόρων, ευελιξία στην αντιμετώπιση των μεταβαλλόμενων απαιτήσεων των workloads και αυξημένη διαθεσιμότητα των συστημάτων. Ο υπερεποπτευτής εικονικών μηχανών Firecracker υποστηρίζει ελαστική μνήμη μέσω του μηχανισμού memory ballooning, ωστόσο μελέτες έχουν δείξει πως ο μηχανισμός αυτός εμφανίζει σημαντικές χρονοκαθυστερήσεις. Το virtio-mem αποτελεί έναν πιο αποδοτικό μηχανισμό ελαστικής μνήμης ο οποίος χρησιμοποιείται ήδη απο άλλους υπερεποπτευτές όπως το Cloud Hypervisor. Στην παρούσα διπλωματική υλοποιήθηκε η υποστήριξη της συσκευής virtio-mem στον υπερεποπτευτή εικονικών μηχανών Firecracker, με στόχο την παροχή αποδοτικής ελαστικής μνήμης σε microVMs. Παράλληλα, υλοποιήθηκε και η υποστήριξη snapshot για τη συκσευή, επιτρέποντας την αποθήκευση και επαναφορά της κατάστασής της. Η υλοποίηση αξιολογήθηκε πειραματικά και συγκρίθηκε τόσο με το virtio-mem του Cloud Hypervisor, όσο και με τη συσκευή virtio-balloon του Firecracker, αναδεικνύοντας τα πλεονεκτήματα και τις διαφορές ως προς την απόδοση. Τέλος, πραγματοποιήθηκε αξιολόγηση της υλοποίησης με την χρήση ενός τροποποιημένου πυρήνα Linux (Squeezy) για τον guest, στον οποίο έχει υλοποιηθεί ένας μηχανισμός αποδοτικότερης ανάκτησης (reclamation) μνήμης, εξαλείφοντας τα κόστη των μετακινήσεων σελίδων (migrations) και της αρχικοποίησης τους (zeroing).

Λέξεις Κλειδιά

Υπολογιστικό νέφος, Εικονικοποίηση, microVM, Ελαστική μνήμη, virtio-mem, Firecracker, Υπερεπόπτης

Abstract

Elastic memory has become increasingly important in cloud computing. It enables the dynamic adjustment of a virtual machine's memory on demand without requiring a reboot. This capability leads to more efficient resource utilization, greater flexibility in responding to fluctuating workload demands, and improved system availability. The Firecracker virtual machine monitor (VMM) currently supports elastic memory through the memory ballooning mechanism. However, studies have shown that this approach can introduce significant latency. Virtio-mem is a more efficient elastic memory mechanism that has already been adopted by other VMMs such as the Cloud Hypervisor. In this thesis, we implement virtio-mem support in the Firecracker VMM, aiming to provide efficient elastic memory for microVMs. Additionally, we add snapshot support for the device, allowing its state to be saved and restored. We evaluate and compare our prototype against both the virtio-mem device of Cloud Hypervisor and Firecracker's virtio-balloon device, demonstrating the performance benefits and key differences between them. Finally, we evaluate our prototype when using a modified guest Linux kernel (Squeezy), which implements a more efficient memory reclamation mechanism by eliminating migration and zeroing overheads.

Keywords

Cloud Computing, Virtualization, microVM, Elastic Memory, virtio-mem, Firecracker, Hypervisor

Ευχαριστίες

Αρχικά θα ήθελα να ευχαριστήσω τον καθηγητή κ. Νεκτάριο Κοζύρη για την επίβλεψη της εργασίας. Ιδιαίτερα θέλω να ευχαριστήσω τους Ορέστη Λάγκα-Νικολό και Στράτο Ψωμαδάκη για την πολίτιμη βοήθεια που μου προσέφεραν και για την καθοδήγησή τους καθ΄όλη τη διάρκεια της εργασίας. Η εργασία δεν θα είχε την ίδια πληρότητα χωρίς την επιμονή και τις ιδέες τους.

Τους γονείς μου, Νίκο και Μαρία, τα αδέρφια μου και τη θεία μου την Κατερίνα, για τη στήριξη τους και την ανιδιοτελή τους αγάπη όλα αυτά τα χρόνια.

Όλους τους φίλους μου και όλες τις φίλες μου που ήταν πάντα δίπλα μου και μου χάρισαν αυτά τα πολύ όμορφα φοιτητικά χρόνια. Η φιλία τους αποτελεί παράδειγμα και δεν θα ήμουν ο ίδιος χωρίς αυτούς. Ιδιαιτέρως τον Παναγιώτη, τον Θάνο, τον Δημήτρη, την Αρετή.

Αθήνα, Νοέμβριος 2025

Βασίλειος-Παναγιώτης Μηλίγγας

Περιεχόμενα

Κατάλογος Σχημάτων

Κεφάλαιο 1

Εισαγωγή

Τα τελευταία χρόνια, το υπολογιστικό νέφος (cloud computing) έχει γνωρίσει ραγδαία ανάπτυξη. Οργανισμοί, επιχειρήσεις και τελικοί χρήστες χρησιμοποιούν το cloud επωφελούμενοι απο την δυνατότητα κατανάλωσης πόρων κατά απαίτηση, την οικονομία κλίμακας και την άμεση πρόσβαση σε ένα μεγάλο εύρος τεχνολογικών υπηρεσιών. Με αυτόν τον τρόπο, αποφεύγουν το μεγάλο κόστος που θα είχε η αγορά, η εγκατάσταση και η συντήρηση ιδιόκτητων υποδομών [5].

Επιπλέον, το cloud προσφέρει στους χρήστες τη δυνατότητα να προσαρμόζουν δυναμικά τους πόρους που καταναλώνουν (resource elasticity). Αυτή η δυνατότητα σε συνδυασμό με το μοντέλο χρέωσης pay-as-you-go (οι χρήστες χρεώνονται μόνο για τους πόρους που χρησιμοποιούν πραγματικά) επιτρέπει στους χρήστες να βελτιστοποιήσουν τη χρέωσή τους στο cloud και να κάνουν καλύτερη αξιοποίηση των διαθέσιμων πόρων. Ακόμα, η ελαστικότητα πόρων προσφέρει ευελιξία, καθώς τα VMs που χρησιμοποιούνται στην υποδομή έχουν την δυνατότητα να προσαρμόζονται δυναμικά στις ανάγκες των φόρτων εργασίας (workloads) που εκτελούν, οι οποίοι μπορεί να μεταβάλλονται συνεχώς. Η προσαρμογή γίνεται άμεσα, χωρίς την ανάγκη επανεκκίνησης των VM με νέα διαμόρφωση, κάτι που θα αποτελούσε μια πολύ κοστοβόρα διαδικασία [5][6].

Ένας απο τους πόρους που μπορούν να προσαρμοστούν δυναμικά είναι η μνήμη που διαθέτουν οι εικονικές μηχανές. Υπάρχουν δύο βασικοί μηχανισμοί ελαστικής μνήμης: το memory ballooning και το memory hot(un)plug. Το ballooning, εισάγει σημαντικό overhead κατά την αφαίρεση μνήμης καθώς διαχειρίζεται τη μνήμη σε επίπεδο σελίδας, γεγονός που οδηγεί σε πολλά αιτήματα προς τον οικοδεσπότη της εικονικής μηχανής. Αντιθέτως, το hot(un)plug αποτελεί έναν ταχύτερο μηχανισμό ο οποίος χρησιμοποιείται και από την state-of-the-art συσκευή ελαστικής μνήμης, το virtio-mem. Ωστόσο και αυτός ο μηχανισμός παρουσιάζει κάποιους περιορισμούς κατά την ανάκτηση μνήμης καθώς εισάγει overhead που οφείλεται στις μετακινήσεις και στην αρχικοποίηση σελίδων [3].

Τα τελευταία χρόνια, ένα νέο μοντέλο εκτέλεσης εφαρμογών έχει αναδειχθεί, το Serverless computing. Στο μοντέλο αυτό, ο προγραμματιστής δεν ασχολείται με τη διαχείριση υποδομών ή servers, την ευθύνη αυτή την αναλαμβάνει ο πάροχος των cloud υπηρεσειών. Η βασική διαφορά με παρόμοια μοντέλα του cloud computing (π.χ. το Platform-as-a-service) είναι πως ο χρήστης χρεώνεται μόνο κατά την εκτέλεση της εφαρμογής του, ενώ όσο αυτή δεν εκτελείται δεν υπάρχει καμία χρέωση. Αυτή η ιδιότητα καθιστά το serverless μοντέλο κατάλληλο για την κατ΄ απαίτηση εκτέλεση μικρών προγραμμάτων (συναρτήσεων), με αποτέλεσμα

να χρησιμοποιείται στις FaaS (Function-as-a-service) υπηρεσίες [;].

Στην serverless αρχιτεκτονική, όταν εγείρεται η εκτέλεση κάποιας συνάρτησης (π.χ. από κάποιο συμβάν) δημιουργείται ένα απομονωμένο περιβάλλον για την εκτέλεση της και διαγράφεται όταν αυτή τερματίσει. Προκειμένου να είναι αποδοτικός και ασφαλής αυτός ο μηχανισμός υπάρχει η ανάγκη ταχείας εκκίνησης καθώς και αυξημένης απομόνωσης ενός τέτοιου περιβάλλοντος. Για αυτό το λόγο χρησιμοποιούνται microVMs για την εκτέλεση των συναρτήσεων τα οποία συνδυάζουν την ταχύτητα των container με την απομόνωση των παραδοσιακών VMs. Μια τεχνολογία εικονικοποίησης που εκκινεί και διαχειρίζεται microVMs είναι το Firecracker [7].

Το Firecracker διαθέτει υποστήριξη ελαστικής μνήμης αποκλειστικά μέσω του μηχανισμού ballooning. Λόγω των περιορισμών του ballooning, σε σενάρια όπου οι απαιτήσεις μνήμης των εφαρμογών μεταβάλλονται, η έλλειψη ενός πιο αποτελεσματικού μηχανισμού οδηγεί σε χαμηλή αποδοτικότητα. Για την αντιμετώπιση αυτού του προβλήματος, η εργασία αυτή εστιάζει στην υλοποίηση της συσκευής virtio-mem στο Firecracker. Ακόμα, η υλοποίηση αξιολογείται με την χρήση ενός τροποποιημένου πυρήνα Linux [3] που εξαλείφει τις μετακινήσεις και την αρχικοποίηση των σελίδων κατά την ανάκτηση μνήμης.

Κεφάλαιο 2

Θεωρητικό υπόβαθρο

2.1 Εικονικοποίηση

Η εικονικοποίηση είναι μια τεχνολογία που επιτρέπει την κατανομή των πόρων ενός φυσικού μηχανήματος σε ένα ή περισσότερα εικονικά περιβάλλοντα [8][1]. Ένα εικονικό περιβάλλον (ή εικονική μηχανή) είναι ένα απομονωμένο περιβάλλον εκτέλεσης, το οποίο δίνει την εντύπωση ενός ξεχωριστού φυσικού διακομιστή, παρόλο που οι υποκείμενοι μηχανισμοί που το συγκροτούν είναι τυπικά διαφορετικοί [9][10].

Οι τεχνολογίες εικονικοποίησης χρησιμοποιούνται σε ένα ευρύ φάσμα τομέων, όπως είναι το υπολογιστικό νέφος, η ενοποίηση διακομιστών (server consolidation), η υποστήριξη πολλαπλών λειτουργικών συστημάτων, το debugging, η ανάπτυξη πυρήνα (kernel development) κ.λ.π., με αποτέλεσμα την ευρεία χρήση τους. Παρόλο οι περισσότεροι τύποι εικονικών μηχανών παρουσιάζουν παρόμοιο περιβάλλον λειτουργίας προς στο τελικό χρήστη, τείνουν να διαφέρουν σημαντικά ως προς το επίπεδο αφαίρεσης στο οποίο λειτουργούν και την υποκείμενη αρχιτεκτονική. [10].

Υπάρχουν πολλά στρώματα αφαίρεσης πάνω στα οποία μπορεί να πραγματοποιηθεί η εικονικοποίηση. Δύο από αυτά αποτελούν το επίπεδο υλικού (hardware virtualization) και το επίπεδο λειτουργικού συστήματος (OS level virtualization ή containerization). Οποιοδήποτε και αν είναι το επίπεδο αφαίρεσης η γενική ιδέα παραμένει ίδια: Οι χαμηλότερου επιπέδου πόροι διαμοιράζονται σε πολλαπλές υψηλότερου επιπέδου VMs με τη χρήση ειδικών τεχνικών [10].

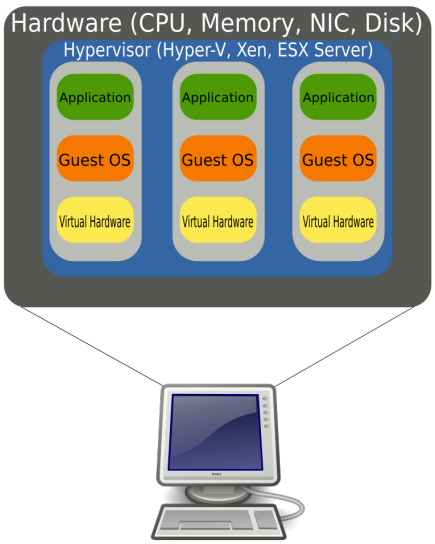
2.1.1 Εικονικοποίηση επιπέδου υλικού (Hardware Virtualization)

Κατά την εικονικοποίηση επιπέδου υλικού οι πόροι του φυσικού υπολογιστή μοιράζονται σε μία ή περισσότερες εικονικές μηχανές. Σε αυτό το επίπεδο μια εικονική μηχανή έχει τη λειτουργικότητα ενός κανονικού φυσικού υπολογιστή και διαθέτει δικό της λειτουργικό σύστημα. Το λογισμικό του host που δημιουργεί μια εικονική μηχανή ονομάζεται υπερεπόπτης (hypervisor) ή υπερεποπτευτής εικονικής μηχανής (Virtual Machine Monitor). Ο υπολογιστής πάνω στον οποίο πραγματοποιείται η εικονικοποίηση ονομάζεται οικοδεσπότης ενώ η εικονική μηχανή ονομάζεται επισκέπτης (guest) [1].

Το λογισμικό που εκτελείται σε αυτές τις εικονικές μηχανές δεν σχετίζεται απαραίτητα με τους πόρους του υποκείμενου υλικού. Για παράδειγμα, ένας υπολογιστής που εκτελεί Linux μπορεί να φιλοξενήσει μια εικονική μηχανή που μοιάζει με υπολογιστή που εκτελεί Windows. Λογισμικό βασισμένο σε Windows μπορεί να εκτελεστεί σε αυτή την εικονική μηχανή [1].

Μοντέλα Hardware Virtualization

Πλήρης εικονικοποίηση: Κατά την πλήρη εικονικοποίηση ο guest αποκτά το δικό του (εικονικό) υλικό, το οποίο αποτελεί μία πλήρη εξομοίωση του υλικού του host. Η πλήρης εξομοίωση του υλικού του host επιτρέπει την εκτέλεση μη τροποποιημένων λειτουργικών συστημάτων στον guest και του προσφέρει πλήρη απομόνωση από το υλικό του host. Αυτό είναι ιδιαίτερα χρήσιμο σε διάφορες περιπτώσεις. Για παράδειγμα, κατά την ανάπτυξη λειτουργικών συστημάτων, ο πειραματικός νέος κώδικας μπορεί να εκτελεστεί ταυτόχρονα με παλαιότερες εκδόσεις, εκτελώντας κάθε μία σε διαφορετικό VM. Υπεύθυνος για την εξομοίωση του υλικού του host είναι ο hypervisor ο οποίος παρέχει στα VMs όλες τις υπηρεσίες που θα προσέφερε ένα φυσικό μηχάνημα, όπως BIOS, συσκευές εισόδου/εξόδου κ.λ.π. [1][11]. Μια εικόνα της δομής της πλήρους εικονικοποίησης φαίνεται στο Σχήμα 2.1



Σχήμα 2.1: *Δομή της πλήρους εικονικοποίησης σε έναν υπολογιστή [1]*

Η πλήρης εικονικοποίηση εξασφαλίζει υψηλό επίπεδο απομόνωσης και ασφάλειας μεταξύ των εικονικών μηχανών ενώ παράλληλα διευκολύνει τη μετεγκατάσταση και τη φορητότητα, καθώς το ίδιο στιγμιότυπο του guest λειτουργικού συστήματος μπορεί να εκτελεστεί είτε σε εικονικοποιημένο είτε σε φυσικό περιβάλλον χωρίς τροποποιήσεις [11]. Ωστόσο, η πλήρης εξομοίωση του υλικού του host εισάγει σημαντικό overhead, καθώς κάθε εντολή που εκτελείται από την εικονική μηχανή πρέπει να ¨παγιδεύεται' από τον hypervisor και στη συνέχεια να εξομοιώνεται μέσω του υλικού του host (trap-and-emulate). Προκειμένου να επιταχυνθεί η εκτέλεση εντολών στην εικονική μηχανή κατά την πλήρη εικονικοποίηση, οι εντολές που δεν χρειάζονται υψηλά δικαιώματα εκτέλεσης (π.χ. δικαιώματα root) εκτελούνται απευθείας στο υλικό του host ενώ το σύνολο των εντολών του επεξεργαστή που απαιτούν εκτέλεση με υψηλότερα δικαιώματα εξομοιώνεται. Αυτή η τεχνική είναι πιο αποδοτική απο την πλήρη εξομοίωση ωστόσο περιορίζει τα λειτουργικά συστήματα που μπορούν να χρησιμοποιήσουν οι guest σε αυτά που υποστηρίζει η CPU του host [12].

Παραεικονικοποίηση Paravirtualization: Η παραεικονικοποίηση (paravirtualization) είναι μια τεχνική η οποία παρέχει στους guest μια διεπαφή για την άμεση επικοινωνία με

τον hypervisor. Σκοπός αυτής της διεπαφής είναι η ανάθεση λειτουργιών στον host οι οποίες θα ήταν πολύ κοστοβόρο να εκτελεστούν στο εικονικοποιημένο περιβάλλον. Η επικοινωνία μεταξύ guest και hypervisor επιτυγχάνεται μέσω ειδικών κλήσεων (hypercalls) [8][11].

Η παραεικονικοποίηση απαιτεί την τροποποίηση του λειτουργικού συστήματος του guest ώστε αυτό να υποστηρίζει την διεπαφή που χρησιμοποιείται για την επικοινωνία με τον host. Ένα συμβατικό λειτουργικό σύστημα που δεν υποστηρίζει την παραεικονικοποίηση δεν μπορεί να εκτελεστεί πάνω σε έναν παραεικονικοποιημένο hypervisor [8].

Εικονικοποίηση υποστηριζόμενη απο το υλικό (Hardware-Assisted Virtualization): Η εικονικοποίηση με υποστήριξη υλικού αποτελεί μια βελτίωση της πλήρους εικονικοποίησης που χρησιμοποιεί τον συνδυασμό άμεσης εκτέλεσης και binary translation. Σε αυτό το μοντέλο, ο hypervisor αποκτά περισσότερα δικαιώματα εκτέλεσης εντολών στη CPU του host. Έτσι, οι εντολές υψηλότερων προνομίων του guest ¨παγιδεύονται' απο τον hypervisor και εκτελούνται απευθείας στο υλικό χωρίς την ανάγκη εξομοίωσης τους, με αποτέλεσμα να μειώνεται ο χρόνος εκτέλεσης τους σε σχέση με το binary translation [12].

Υβριδική Εικονικοποίηση (Hybrid Virtualization): Η υβριδική εικονικοποίηση (hybrid virtualization) είναι μια προσέγγιση που συνδυάζει τα πλεονεκτήματα της παραεικονικοποίησης (paravirtualization) με εκείνα της υποστηριζόμενης απο το υλικό εικονικοποίησης (hardware-assisted virtualization). Στο μοντέλο αυτό, πολλές λειτουργίες εκτελούνται με την υποστήριξη του υλικού, προσφέροντας υψηλή απόδοση και πλήρη συμβατότητα με μη τροποποιημένα λειτουργικά συστήματα, ενώ χρησιμοποιούνται παραεικονικοποιημένοι οδηγοί (paravirtualized drivers) για πιο περίπλοκες λειτουργίες που εγείρουν πολλά traps προς τον host [1].

2.1.2 Εικονικοποίηση επιπέδου λειτουργικού συστήματος (Containerization)

Κατά την εικονικοποίηση επιπέδου λειτουργικού συστήματος (OS-level Virtualization ή Containerization) οι εικονικές μηχανές (ή αλλιώς containers) δεν διαθέτουν δικό τους λειτουργικό σύστημα. Αντιθέτως βασίζονται στο λειτουργικό σύστημα του φυσικού υπολογιστή στον οποίο εκτελούνται ως τυπικές διεργασίες. Αυτό σημαίνει πως σε περιπτώσεις ταυτόχρονης εκτέλεσης πολλαπλών containers το λειτουργικό του host είναι υπεύθυνο για τον διαμοιρασμό των φυσικών πόρων καθώς και για την εξασφάλιση της απομόνωσης μεταξύ των containers [13][14].

Η προσέγγιση του containerization δεν απαιτεί την ύπαρξη hypervisor ή κάποιου στρώματος εξομοίωσης, καθώς τα containers επικοινωνούν με το λειτουργικό του host μέσω system calls, όπως οι κανονικές διεργασίες. Η απουσία στρώματος εικονικοποίησης μειώνει κατά πολύ το overhead λειτουργίας τους σε σχέση με τα VMs, ενώ η ελαφριά τους διαμόρφωση επιτρέπει την ταυτόχρονη εκτέλεση μεγάλου αριθμού containers σε έναν φυσικό διακομιστή [14].

Ωστόσο, η εξάρτηση απο το λειτουργικό σύστημα του host συνεπάγεται πως τα φιλοξενούμενα containers πρέπει να εκτελούν το ίδιο ή παρόμοιο λειτουργικό σύστημα με αυτό του host. Ένα ακόμα μειονέκτημα που παρουσιάζει το containerization είναι πως η χρήση κοινού πυρήνα μειώνει το επίπεδο απομόνωσης μεταξύ των containers, γεγονός που τα καθιστά

λιγότερο ασφαλή σε σχέση με τις εικονικές μηχανές του επιπέδου αφαίρεσης υλικού [14].

2.1.3 Hypervisor (Υπερεπόπτης)

Ο υπερεπόπτης (hypervisor) ή αλλιώς VMM (virtual machine monitor) είναι ένας τύπος λογισμικού (software), ή υλικού (hardware) υπολογιστή, που δημιουργεί και διαχειρίζεται εικονικές μηχανές. Ο υπολογιστής στον οποίο ένας υπερεπόπτης εκτελεί μία ή περισσότερες εικονικές μηχανές ονομάζεται οικοδεσπότης (host machine) και κάθε εικονική μηχανή ονομάζεται επισκέπτης (guest machine).

Κατηγορίες hypervisor

- **Hypervisors τύπου 1: Εγγενείς ή bare-metal**: Αυτοί οι hypervisors εκτελούνται απευθείας στο υλικό του host και ως αποτέλεσμα έχουν άμεση πρόσβαση σε αυτό. Οι hypervisors τύπου 1 είναι υπεύθυνοι για την κατανομή και την διαχείριση των φυσικών πόρων του host (όπως η CPU, η μνήμη, και οι συσκευές εισόδου/εξόδου) στις εικονικές μηχανές. Οι πρώτοι hypervisors που αναπτύχθηκαν από την IBM τη δεκαετία του 1960 ήταν bare-metal. Παραδείγματα hypervisor τύπου 1 είναι οι VMware ESX, KVM και Xen [12][15][16].
- **Hypervisors τύπου 2: VMMs ή φιλοξενούμενοι υπερεπόπτες:** Οι hypervisors τύπου 2 εκτελούνται ως διεργασίες στον host και παρέχουν μια αφαίρεση του λειτουργικού του guest προς αυτόν, δημιουργώντας με αυτό τον τρόπο ένα απομονωμένο σύστημα με το οποίο μπορεί να αλληλοεπιδράσει ο host [15]. Σε αυτό τον τύπο υπερεπόπτη συνήθως ο host εκτελεί I/O αιτήματα εκ μέρους του λειτουργικού του guest. Το λειτουργικό του guest εκδίδει ένα αίτημα I/O που παγιδεύεται απο το VMM, το οποίο με τη σειρά του το στέλνει στη συσκευή που εκτελεί το I/O. Έπειτα το ολοκληρωμένο αίτημα I/O δρομολογείται πίσω στο λειτουργικό σύστημα του guest μέσω του VMM [16]. Παραδείγματα hypervisor τύπου 2 είναι οι Cloud Hypervisor, Firecracker, και QEMU.

Μερικές φορές στη βιβλιογραφία γίνεται διάκριση μεταξύ hypervisor και VMM. Ο hypervisor παραπέμπει στις λειτουργίες του χώρου πυρήνα, ενώ το VMM στις λειτουργίες χώρου χρήστη. Για παράδειγμα στο πλαίσιο των Linux, το KVM είναι ένας hypervisor και το QEMU είναι ένα VMM που χρησιμοποιεί το KVM ως hypervisor [15]. Στο πλαίσιο της παρούσας εργασίας, πολλές φορές όταν αναφερόμαστε στον hypervisor, αναφερόμαστε στο ζευγάρι VMM/hypervisor.

KVM (Kernel-based Virtual Machine)

Το KVM είναι ένα open-source module εικονικοποίησης στον πυρήνα του Linux το οποίο του επιτρέπει να λειτουργεί ως hypervisor τύπου bare-metal. Το KVM απαιτεί επεξεργαστή με επεκτάσεις εικονικοποίησης υλικού όπως Intel VT ή AMD-V [17]. Επειδή πραγματοποιεί εικονικοποίηση στο επίπεδο υλικού, δεν απαιτεί το λειτουργικό σύστημα του guest να είναι τροποποιημένο και ως εκ τούτου μπορεί να υποστηρίξει οποιαδήποτε πλατφόρμα με την προϋπόθεση ότι έχει εγκατασταθεί σε επεξεργαστή που το υποστηρίζει [18].

Στην KVM αρχιτεκτονική, κάθε εικονική μηχανή είναι μια κανονική διεργασία των Linux. Κατά κανόνα μια διεργασία έχει δύο μορφές (modes) εκτέλεσης: πυρήνα (kernel mode) και χρήστη (user mode). Η λειτουργία χρήστη είναι η προκαθορισμένη λειτουργία για διεργασίες, ενώ μια διεργασία μεταβαίνει σε λειτουργία πυρήνα όταν χρειάζεται κάποια υπηρεσία από τον πυρήνα, όπως εγγραφή στο σκληρό δίσκο. Το KVM εισάγει μια τρίτη λειτουργία εκτέλεσης, γνωστή ως λειτουργία επισκέπτη (guest mode), η οποία αφορά τις διεργασίες που εκτελούνται μέσα στην εικονική μηχανή. Η λειτουργία guest mode έχει και αυτή τις δικές της εναλλαγές χώρου πυρήνα και χώρου χρήστη μέσα στην εικονική μηχανή [18].

Το KVM έχει τη δομή μιας τυπικής συσκευής χαρακτήρων των Linux. Εκθέτει έναν κόμβο συσκευής /dev/kvm ο οποίος μπορεί να χρησιμοποιηθεί από το χώρο χρήστη για τη δημιουργία και εκτέλεση εικονικών μηχανών. Οι λειτουργίες που παρέχει ο κόμβος /dev/kvm περιλαμβάνουν:

- Δημιουργία νέας εικονικής μηχανής.
- Εκχώρηση μνήμης σε εικονική μηχανή.
- Ανάγνωση και εγγραφή εικονικών καταχωρητών CPU.
- Διακοπή (interrupt) προς εικονική CPU.
- Εκτέλεση εικονικής CPU [19].

2.1.4 MicroVMs και Firecracker VMM

MicroVMs

Για πολλά χρόνια, οι προγραμματιστές λογισμικού και οι μηχανικοί συστημάτων αντιμετώπιζαν ένα θεμελιώδες δίλημμα κατά την επιλογή τεχνολογιών εικονικοποίησης. Από τη μία πλευρά οι παραδοσιακές εικονικές μηχανές (VM) παρέχουν ισχυρή απομόνωση, εξομοιώνοντας ένα ολόκληρο σύστημα υπολογιστή, το οποίο διαθέτει το δικό του λειτουργικό σύστημα, μνήμη και πόρους. Αυτή η αρχιτεκτονική προσφέρει υψηλό βαθμό ασφαλείας και διαχωρισμού μεταξύ των workloads, εισάγει όμως σημαντικό overhead κατά την εκκίνηση, ενώ επίσης παρουσιάζει αυξημένες απαιτήσεις σε πόρους [20].

Το containerization έχει αναδειχθεί ως μια ελαφριά εναλλακτική λύση. Τα containers μοιράζονται τον πυρήνα του λειτουργικού συστήματος του κεντρικού υπολογιστή, επιτρέποντας την ταχεία κλιμάκωση των στιγμιοτύπων τους με ελάχιστη χρήση πόρων. Η ταχύτητά τους τα καθιστά ιδιαίτερα ελκυστικά για τις σύγχρονες αρχιτεκτονικές που βασίζονται σε microservices. Ωστόσο, αυτή η προσέγγιση κοινής χρήσης πυρήνα εισάγει πιθανούς κινδύνους ασφαλείας καθώς μειώνει το συνολικό επίπεδο απομόνωσης σε σύγκριση με τις VM [20].

Τα microVMs σχεδιάστηκαν ώστε να συνδυάζουν τα πλεονεκτήματα των δύο παραπάνω προσεγγίσεων. Ένα microVM είναι μια ελαφριά εικονική μηχανή. Η εικονικοποίηση στο επίπεδο υλικού διατηρείται, ενώ όλα τα μη απαραίτητα στοιχεία που απαρτίζουν ένα παραδοσιακό VM παραλείπονται προκειμένου να μεγιστοποιηθεί η αποδοτικότητα. Συγκεκριμένα, αποφεύγεται η υποστήριξη περίπλοκων λειτουργιών των guest και η εξομοίωση περιττών hardware συσκευών (π.χ. legacy συσκευές) με στόχο την εξάλειψη του επιπλέον overhead που θα

προκαλούσε η διαχείριση τους. Αντ' αυτού τα microVMs επικεντρώνονται αποκλειστικά στο ελάχιστο σύνολο λειτουργιών που απαιτούνται για την ασφαλή και αποδοτική εκτέλεση εργασιών. Με αυτόν τον τρόπο επιτυγχάνεται σημαντική μείωση τόσο της επιφάνειας επίθεσης (attack surface) όσο και του αποτυπώματος μνήμης τους memory footprint.[7][20].

Το αποτέλεσμα είναι μια εικονική μηχανή που συνδυάζει τις ισχυρές εγγυήσεις απομόνωσης της παραδοσιακής εικονικοποίησης, με χαρακτηριστικά απόδοσης που προσεγγίζουν αυτά των container. Αυτά τα γνωρίσματα τα καθιστούν κατάλληλα για serverless περιβάλλοντα που φιλοξενούν workloads πολλαπλών χρηστών (multi-tenant) όπου η απομόνωση, η ταχύτητα και η επεκτασιμότητα (scalability), έχουν ιδιαίτερη σημασία [7].

Firecracker

Το Firecracker είναι μια τεχνολογία εικονικοποίησης ανοιχτού κώδικα (open-source) που έχει σχεδιαστεί ειδικά για τη διαχείριση workloads σε multi-tenant serverless περιβάλλοντα, καθώς τα εκτελεί σε microVMs, ενώ επικεντρώνεται κυρίως σε υπηρεσίες FaaS. Αναπτύχθηκε από την Amazon Web Services για να βελτιώσει την εμπειρία των πελατών σε υπηρεσίες όπως οι AWS Lambda και AWS Fargate [21][7].

Παρόμοια project αναγνωρίζοντας την ανάγκη για βελτιωμένη απομόνωση των workloads λόγω του multi-tenancy, επέλεξαν την εικονικοποίηση βάσει bare-metal hypervisor. Συνήθως χρησιμοποιείται το ζευγάρι QEMU/KVM ως ζευγάρι VMM/hypervisor. Στο Firecracker επιλέχθηκε να διατηρηθεί το KVM αλλά να αντικατασταθεί πλήρως το QEMU από ένα νέο μοντέλο VMM με πιο λιτή υλοποίηση, γραμμένο σε μια ασφαλή γλώσσα, την Rust, και που διαθέτει API για την διαμόρφωση και διαχείριση των microVMs. Το αποτέλεσμα ήταν το Firecracker να περιέχει περίπου 50 χιλιάδες γραμμές κώδικα, δηλαδή 96% λιγότερες από αυτές του QEMU. Δεδομένης της παροχής λιτής διαμόρφωσης Linux guest πυρήνα, προσφέρει overhead μνήμης μικρότερο από 5MB ανά microVM, εκκίνηση σε λιγότερο από 125ms, και επιτρέπει την δημιουργία έως και 150 microVMs ανά δευτερόλεπτο ανά host. Επίσης λόγω της έλλειψης πολλών στοιχείων των συμβατικών VM έχει μειωμένη επιφάνεια επίθεσης (attack surface) γεγονός που βελτιώνει την ασφάλειά του.

2.2 Εικονική μνήμη και διαχείριση μνήμης στις εικονικές μηχανές

2.2.1 Εικονική Μνήμη

Στην πληροφορική, η εικονική μνήμη είναι μια τεχνική διαχείρισης μνήμης που παρέχει μια ιδανική αφαίρεση των πόρων αποθήκευσης που είναι πραγματικά διαθέσιμοι σε έναν υπολογιστή και δημιουργεί την ψευδαίσθηση στους χρήστες ότι διαθέτουν μια πολύ μεγάλη (κύρια) μνήμη.

Το λειτουργικό σύστημα του υπολογιστή, αντιστοιχίζει τις διευθύνσεις μνήμης που χρησιμοποιούνται από ένα πρόγραμμα, οι οποίες ονομάζονται εικονικές διευθύνσεις, σε φυσικές διευθύνσεις στη μνήμη του υπολογιστή. Ο κύριος χώρος αποθήκευσης όπως φαίνεται από μια διεργασία, εμφανίζεται ως ένας συνεχόμενος χώρος διευθύνσεων ή συλλογή συνεχόμενων τμημάτων [22].

Επιπλέον, χρησιμοποιώντας τον αποθηκευτικό χώρο του δίσκου ως προέκταση της φυσικής μνήμης, το λειτουργικό σύστημα μπορεί να παρουσιάζει στις διεργασίες έναν εικονικό χώρο διευθύνσεων που υπερβαίνει σε μέγεθος την διαθέσιμη φυσική μνήμη του συστήματος [22]. Σε αυτό το μοντέλο διαχείρισης μνήμης, όταν δημιουργείται συμφόρηση στη RAM η μνήμη που έχει προσπελαστεί λιγότερο πρόσφατα (Least Recently Used - LRU) μεταφέρεται στον δίσκο και επαναφέρεται στην RAM όταν προσπελαστεί ξανά. Αυτή η διαδικασία ονομάζεται swapping [23].

Μερικά πλεονεκτήματα της εικονικής μνήμης αποτελούν τα εξής [22]:

- Απελευθέρωση των εφαρμογών από την ανάγκη διαχείρισης ενός κοινόχρηστου χώρου μνήμης.
- Δυνατότητα κοινής χρήσης της μνήμης που χρησιμοποιούν βιβλιοθήκες διεργασιών.
- Αυξημένη ασφάλεια λόγω απομόνωσης απο τη μνήμη αλλών διεργασιώ.
- Δυνατότητα εννοιολογικής χρήσης περισσότερης μνήμης από ό,τι είναι φυσικά διαθέσιμη.

Σελιδοποίηση της Εικονικής Μνήμης

Σχεδόν όλες οι τρέχουσες εφαρμογές της εικονικής μνήμης χωρίζουν τον εικονικό χώρο διευθύνσεων σε σελίδες (pages), δηλαδή κομμάτια συνεχόμενων διευθύνσεων, που αντιστοιχίζονται σε εγγραφές στον πίνακα σελίδων (page table). Τα pages στα σύγχρονα συστήματα έχουν συνήθως μέγεθος τουλάχιστον 4 kilobytes [22]. Ένα page frame είναι το μικρότερο σταθερού μεγέθους κομμάτι φυσικής μνήμης στο οποίο το λειτουργικό σύστημα μπορεί να αντιστοιχίσει κάποιο page μνήμης [24].

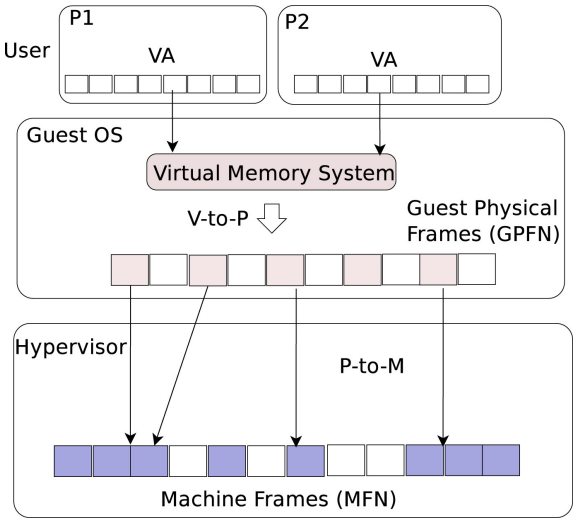
Οι πίνακες σελίδων (page tables) χρησιμοποιούνται για τη μετάφραση των εικονικών διευθύνσεων που βλέπει η εφαρμογή, σε φυσικές διευθύνσεις που χρησιμοποιεί το υλικό. Το υλικό που χειρίζεται αυτή τη μετάφραση είναι γνωστό ως μονάδα διαχείρισης μνήμης (MMU). Κάθε εγγραφή στο page table περιέχει ένα flag που υποδεικνύει εάν η αντίστοιχη σελίδα βρίσκεται στη πραγματική μνήμη ή όχι. Εάν βρίσκεται στην πραγματική μνήμη, η εγγραφή στο page table θα περιέχει τη διεύθυνση στην οποία είναι αποθηκευμένη η σελίδα. Διαφορετικά το υλικό εγείρει ένα page fault, καλώντας τον αντίστοιχο χειριστή (handler) του λειτουργικού συστήματος [22]. Ο page fault handler πρέπει στη συνέχεια να βρει μια ελεύθερη θέση στη μνήμη για αυτή τη σελίδα. Στην περίπτωση που δεν υπάρχει ελεύθερη θέση, το λειτουργικό τοποθετεί τη σελίδα στη θέση μίας μη ελεύθερης σελίδας (π.χ με πολιτική LRU) και η σελίδα αυτή μεταφέρεται σε δευτερεύον αποθηκευτικό χώρο (π.χ. στο δίσκο), εφόσον υποστηρίζεται το swapping. Σε δεύτερη φάση ενημερώνονται οι εγγραφές των σελίδων στο page table ώστε να υποδεικνύουν την θέση τους στη μνήμη (ή την απουσία τους απο αυτή) [25].

2.2.2 Διαχείριση μνήμης στις εικονικές μηχανές

Η εικονικοποίηση εισάγει ένα επιπλέον επίπεδο διαχείρισης μνήμης, το οποίο απαιτεί οι παραδοσιακοί μηχανισμοί που βασίζονται στη μονάδα διαχείρισης μνήμης (MMU) να ελέγχονται από τον hypervisor, προκειμένου να διασφαλίζεται η ορθότητα και η απομόνωση μεταξύ των

εικονικών μηχανών. Η εικονικοποίηση του MMU συνεπάγεται πρόσθετους παράγοντες που πρέπει να ληφθούν υπόψη κατά τον σχεδιασμό και την υλοποίησή της [2].

Οι διεργασίες που εκτελούνται σε μια εικονική μηχανή μοιράζονται το φυσικό χώρο διευθύνσεων του guest με την υποστήριξη μιας 'virtual to physical" (V-to-P) αντιστοίχησης που παρέχεται από το λειτουργικό σύστημα του guest. Οι φυσικές διευθύνσεις του guest αντιστοιχίζονται σε φυσικές διευθύνσεις του host όπως ακριβώς αντιστοιχίζονται οι εικονικές διευθύνσεις διεργασιών του host σε φυσικές. Όπως φαίνεται στο Σχήμα 2.4, κάθε εικονική διεύθυνση (VA) από τις διεργασίες P1 και P2 μεταφράζεται σε μια φυσική διεύθυνση στη μνήμη του guest και στη συνέχεια ο hypervisor μεταφράζει τα physical page frame numbers του guest (GPFN) στα πραγματικά MFN (machine frame numbers) της φυσικής μνήμης του host [2].



Σχήμα 2.2: *Διαχείριση μνήμης στα εικονικά συστήματα με ψευδο-φυσικές σελίδες στον guest που ανιστοιχίζονται σε σελίδες του host [2]*

Τα δύο επίπεδα μετάφρασης μνήμης που απαιτούνται για την εικονικοποίηση της μνήμης είναι η κύρια πηγή πολυπλοκότητας και επιβάρυνσης στα εικονικοποιημένα συστήματα. Χρησιμοποιούνται συνήθως τεχνικές όπως software-based MMU virtualization, paravirtualized MMU virtualization ή hardware assisted MMU virtualization για την βελτίωση της απόδοσης [2].

2.2.3 Δυναμική προσαρμογή μνήμης στις εικονικές μηχανές

Οι απαιτήσεις που έχουν οι εφαρμογές σε μνήμη μπορεί να ποικίλουν με την πάροδο του χρόνου. Ο hypervisor μπορεί να εκμεταλλευτεί αυτό το γεγονός και να κατανείμει ανάλογα την μνήμη, έτσι ώστε η ποσότητα μνήμης που κατέχει μια εικονική μηχανή οποιαδήποτε χρονική στιγμή να είναι ακριβώς αυτή που έχει ανάγκη για τις εφαρμογές που εκτελούνται σε αυτή. Η δυνατότητα δυναμικής αύξησης ή μείωσης της μνήμης που είναι διαθέσιμη σε μια εικονική μηχανή γίνεται όλο και πιο δημοφιλής καθώς βρίσκει εφαρμογή σε πολλούς τομείς. Για παράδειγμα σε ένα cloud περιβάλλον, η δυναμική προσαρμογή της μνήμης επιτρέπει στους χρήστες να βελτιστοποιήσουν τη χρέωσή τους [2][26]. Οι τεχνικές που επιτρέπουν τη δυναμική προσαρμογή μνήμης εκμεταλλεύονται την ευελιξία της φυσικής μνήμης του guest σε

mappings με τη φυσική μνήμη του host. Τέτοιες τεχνικές είναι το memory ballooning, και το memory hot(un)plug.

Memory ballooning

Το memory ballooning είναι ένας μηχανισμός που χρησιμοποιείται για τη μεταφορά φυσικής μνήμης μεταξύ μιας εικονικής μηχανής και του hypervisor της, προσαρμόζοντας δυναμικά το μέγεθος της μνήμης της εικονικής μηχανής. Στο memory ballooning, το λειτουργικό σύστημα του guest διαθέτει έναν οδηγό για την balloon συσκευή (balloon driver) που διαχειρίζεται το μέγεθος του balloon. Ο driver υποστηρίζει δύο βασικές λειτουργίες: Το inflation και το deflation του balloon. Κατά την διαδικασία του balloon inflate ο guest μέσω του driver κάνει allocate μνήμη και την παραδίδει στον hypervisor έπειτα από αίτημα του δεύτερου. Μόλις ο hypervisor παραλάβει τα GPFN, τα κάνει unmap από τον guest, διαγράφει την αντιστοίχιση τους με τα MFN του host και χρησιμοποιεί αυτή τη μνήμη για άλλες διεργασίες του host. Κατά την διαδικασία του balloon deflate ο hypervisor δημιουργεί ξανά την αντιστοίχηση των GPFN με τα MFN και ειδοποιεί τον guest να ελευθερώσει την μνήμη που είχε κάνει allocate προηγουμένως [2][27][28].

Δυναμική προσθήκη μνήμης μέσω του μηχανισμού hotplug

Η δυνατότητα δυναμικής προσθήκης μνήμης (memory hotplug) μελετήθηκε αρχικά στην κοινότητα ανάπτυξης του πυρήνα του Linux. Το κίνητρο πίσω από αυτή την τεχνική ήταν η δυνατότητα επέκτασης της φυσικής μνήμης ενός συστήματος κατά απαίτηση, χωρίς να απαιτείται επανεκκίνηση ή διακοπή της λειτουργίας του. Για να υποστηρίξει αυτή τη λειτουργία ο πυρήνας πρέπει να χρησιμοποιεί το μοντέλο μνήμης SPARSEMEM. Το SPARSEMEM χωρίζει λογικά τη μνήμη σε τμήματα του ίδιου μεγέθους. Το κάθε τμήμα ονομάζεται section και το μέγεθός του εξαρτάται από την αρχιτεκτονική. Για παράδειγμα η x86-64 χρησιμοποιεί sections των 128MB ενώ η PowerPC των 16MB. Η μονάδα μνήμης που γίνεται hot(un)plug είναι ένα section [2][27].

Η λειτουργία του hotplug χωρίζεται σε δύο φάσεις. Σε πρώτη φάση δημιουργούνται τα μεταδεδομένα που αφορούν τη μνήμη (π.χ. page structs), δημιουργούνται page table entries για την άμεση αντιστοίχιση, και έπειτα δημιουργούνται τα μπλοκ μνήμης [29].

Στη δεύτερη φάση η μνήμη που προστέθηκε διατίθεται στον allocator από τον πυρήνα. Μετά από αυτή τη φάση, η μνήμη είναι πλέον ορατή στους χρήστες. Αυτή η φάση ονομάζεται memory onlining [27][29].

Παρόλο που μπορεί να χρησιμοποιηθεί για την προσθήκη ή την αφαίρεση ενός DIMM σε φυσικά συστήματα, ο μηχανισμός hot(un)plug βρίσκει καλύτερη εφαρμογή στα εικονικοποιημένα συστήματα όπου τα DIMM είναι ένας εικονικός πόρος που παρέχεται στην εικονική μηχανή από τον hypervisor [27].

Δυναμική αφαίρεση μνήμης μέσω του μηχανισμού hotunplug

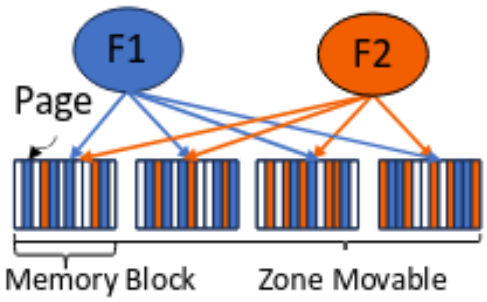
Ομοίως με το hotplug, το hotunplug χωρίζεται και αυτό σε δύο φάσεις. Στην πρώτη φάση η μνήμη αποκρύπτεται απο τον page allocator, ενώ στη δεύτερη φάση η μνήμη αφαιρείται απο το λειτουργικό σύστημα και όλα τα σχετικά με αυτή μεταδεδομένα καταστρέφονται [29].

Η διαδικασία unplug επηρεάζεται σημαντικά από το πρώτο της στάδιο, καθώς αυτό συχνά περιλαμβάνει τη μεταφορά (migration) κατειλημμένων σελίδων μνήμης. Οι μεταφορές αυτές προκαλούνται έμμεσα από τον διαχειριστή μνήμης του guest λειτουργικού συστήματος.

Περιορισμοί του hotunplug

Μεταφορά σελίδων μνήμης

Όπως αναλύεται σε πρόσφατες μελέτες [3] τα Linux τείνουν να διαμοιράζουν τις σελίδες των διεργασιών σε ολόκληρο το εύρος της φυσικής μνήμης με αποτέλεσμα ένα memory block να περιέχει σελίδες πολλών διαφορετικών διεργασιών. Τα Linux εκτελούν allocations μνήμης κατ' απαίτηση (lazy allocation), δηλαδή δεσμεύουν σελίδες μόνο όταν η διεργασία τις προσπελάσει για πρώτη φορά (page fault). Για να εξυπηρετήσει το page fault, το λειτουργικό διαθέτει στη διεργασία οποιαδήποτε διαθέσιμη σελίδα, χωρίς να στοχεύει στη συνεχόμενη κατανομή μνήμης, με αποτέλεσμα τη διάσπαρτη κατανομή των σελίδων στην μνήμη και την αλληλοεπικάλυψη των αποτυπωμάτων μνήμης (memory footprint) διαφορετικών διεργασιών στα blocks μνήμης [3]. Μια αναπαράσταση αυτού του φαινομένου φαίνεται στο Σχήμα 2.3 για τις διεργασίες F1 και F2.



Σχήμα 2.3: *Αλληλοκάλυψη των διεργασίων F1 και F2 στα blocks μνήμης [3]*

Αυτή η αλληλοεπικάλυψη δυσχεραίνει το unplug της μνήμης. Για παράδειγμα, όταν η διεργασία F2 τερματίσει και απελευθερώσει τη μνήμη της, ο host μπορεί να ζητήσει την ανάκτηση (reclaim) της μνήμης αυτής. Ο guest θα προσπαθήσει να απομονώσει μια περιοχή μνήμης ίσου μεγέθους με αυτό της διεργασίας που τερμάτισε, εξετάζοντας γραμμικά τα blocks μνήμης του συστήματος. Επειδή όμως οι σελίδες της F1 παραμένουν αναμειγμένες σε πολλά απο αυτά τα blocks, το λειτουργικό σύστημα θα χρειαστεί να μεταφέρει (migrate) σελίδες για να απομονώσει επαρκή αριθμό blocks που στη συνέχεια θα κάνει offline και unplug [3].

Η μνήμη που προστίθεται δυναμικά (hot-plugged) τοποθετείται συνήθως σε μια ειδική ζώνη του allocator του λειτουργικού συστήματος, τη `ZONE_MOVABLE`. Η ζώνη αυτή έχει σχεδιαστεί ώστε να διαχωρίζει τις μετακινήσιμες (movable) από τις μη μετακινήσιμες (non-movable) σελίδες, με σκοπό να εξασφαλίσει ότι τα blocks που ανήκουν σε αυτή μπορούν να αποσυνδεθούν (offlined) [3].

Αρχικοποίηση σελίδων μνήμης (Page Zeroing)

Ένας ακόμη παράγοντας επιβάρυνσης στη διαδικασία unplug μνήμης προκύπτει από το περιττό μηδενισμό (zeroing) των σελίδων μνήμης.

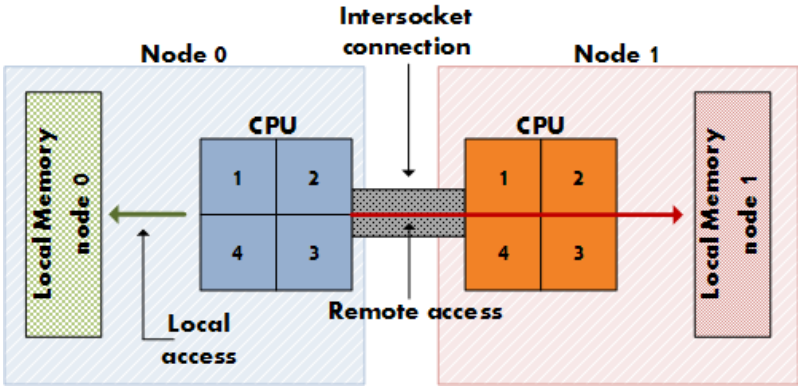
Κατα το offlining της μνήμης το Linux τείνει να αρχικοποιεί τις σελίδες των blocks που πρόκειται να γίνουν offline, ώστε να τα απομονώσει απο το υπόλοιπο σύστημα. Το λειτουργικό δεν γνωρίζει πότε ένα block προορίζεται για unplug και έτσι προχωράει στον περιττό μηδενισμό των σελίδων του, γεγονός που εισάγει αχρείαστο overhead στην hotunplug διαδικασία [3].

2.3 virtio-mem

2.3.1 Υπόβαθρο

Non-Uniform Memory Access (NUMA)

Η Μη Ομοιόμορφη Πρόσβαση στη Μνήμη (Non-Uniform Memory Access – NUMA) αποτελεί μια αρχιτεκτονική σχεδίασης μνήμης που χρησιμοποιείται σε συστήματα πολλαπλών επεξεργαστών (multiprocessor systems). Σε αυτή την αρχιτεκτονική ο χρόνος πρόσβασης στη μνήμη εξαρτάται από τη θέση της μνήμης σε σχέση με τον επεξεργαστή. Σε ένα NUMA σύστημα η μνήμη χωρίζεται σε κόμβους (ζώνες) που ονομάζονται NUMA nodes στους οποίους ανατίθεται μια συγκεκριμένη τοπική CPU, όπως φαίνεται στο Σχήμα 2.4. Ένας επεξεργαστής μπορεί να προσπελάσει τη δική του τοπική μνήμη γρηγορότερα από μια μη τοπική μνήμη, δηλαδή μνήμη που ανήκει σε άλλον επεξεργαστή ή είναι κοινόχρηστη μεταξύ πολλών επεξεργαστών [30][31].



Σχήμα 2.4: *αρχιτεκτονική NUMA [4]*

Η αρχιτεκτονική NUMA είναι ιδιαίτερα αποδοτική για workloads με υψηλό βαθμό τοπικότητας αναφοράς στη μνήμη. Σε τέτοιες περιπτώσεις, κάθε επεξεργαστής εκτελεί λειτουργίες πάνω σε ένα υποσύνολο δεδομένων που βρίσκεται κυρίως στη δική του τοπική περιοχή μνήμης, μειώνοντας έτσι την κυκλοφορία δεδομένων στον δίαυλο μνήμης (memory bus) και βελτιώνοντας τη συνολική απόδοση του συστήματος [30].

VIRTIO

Το VIRTIO είναι ένα πρωτόκολλο επικοινωνίας μεταξύ οδηγών συσκευών (drivers) και εικονικών συσκευών (devices) διάφορων τύπων. Αναπτύχθηκε ως πρότυπο για παραεικονικοποιημένες συσκευές (paravirtualized devices) με σκοπό την αποδοτική επικοινωνία μεταξύ guest και host [32].

Υπεύθυνος για την κατασκευή και την λειτουργία των virtio συσκευών είναι ο hypervisor. Ο hypervisor εκθέτει αυτές τις συσκευές στον guest μέσω τυποποιημένων μηχανισμών, όπως το PCI ή το MMIO, ώστε να μπορούν να αναγνωριστούν και να χρησιμοποιηθούν από το λειτουργικό σύστημα [32].

Η επικοινωνία μεταξύ του οδηγού της συσκευής στον guest και της αντίστοιχης συσκευής στον hypervisor πραγματοποιείται μέσω εγγραφών και αναγνώσεων σε μια κοινόχρηστη περιοχή μνήμης (shared memory). Για τη μεταφορά δεδομένων χρησιμοποιούνται εξειδικευμένες δομές δεδομένων που ονομάζονται virtqueues [32].

2.3.2 Περιορισμοί του memory ballooning και του DIMM (un)plug

Οι μηχανισμοί ανάκτησης μνήμης που παρουσιάστηκαν μέχρι τώρα παρουσιάζουν συγκεκριμένους περιορισμούς που επηρεάζουν την αποδοτικότητα και την ευελιξία τους.

Περιορισμοί του memory ballooning

Παρά τα πλεονεκτήματά του, όπως η απλή υλοποίηση και η ευρεία υποστήριξη από διαφορετικές αρχιτεκτονικές και λειτουργικά συστήματα, το memory ballooning εμφανίζει αρκετούς περιορισμούς [26]. Μερικοί απο αυτούς είναι οι εξής:

- **Έλλειψη NUMA-awareness:** Οι υλοποιημένοι μηχανισμοί ballooning δεν είναι NUMA-aware, δηλαδή δεν λαμβάνουν υπόψη τη δομή και την τοπολογία των NUMA nodes του συστήματος. Τα αιτήματα για inflation ή deflation δεν στοχεύουν συγκεκριμένους NUMA nodes ούτε συγκεκριμένες περιοχές φυσικής μνήμης του guest. Εάν το λειτουργικό του guest δεν έχει ενσωματωμένη υποστήριξη NUMA, ο βασικός page allocator δεν είναι σε θέση να εξυπηρετήσει αιτήσεις δέσμευσης μνήμης σε συγκεκριμένα NUMA nodes. Ως αποτέλεσμα, μπορεί να ανακτηθεί μνήμη απο έναν διαφορετικό NUMA κόμβο από τον επιθυμητό γεγονός που μπορεί να επηρεάσει την απόδοση και άλλων VM του συστήματος [26].

- **Μεγάλος αριθμός VM Exits:** Η απόδοση του μηχανισμού memory ballooning επηρεάζεται σημαντικά από τα VM exits. Κάθε φορά που ο driver του balloon στέλνει ένα αίτημα προς τον hypervisor, πραγματοποιείται ένα VM exit, δηλαδή ο έλεγχος μεταφέρεται προσωρινά από το εικονικό μηχάνημα στον hypervisor. Κατά το διάστημα αυτό, ο driver παραμένει σε κατάσταση αναμονής έως ότου λάβει απάντηση σχετικά με το αίτημα. Επειδή στο memory ballooning η διαδικασία inflation (διόγκωσης του balloon) πραγματοποιείται σε επίπεδο σελίδας, τα VM exits είναι πολλά και ως αποτέλεσμα η διαδικασία είναι ιδιαίτερα κοστοβόρα [3].

Περιορισμοί του παραδοσιακού μηχανισμού hot(un)plug

Ο παραδοσιακός μηχανισμός hot(un)plug είναι NUMA-aware και προκαλεί λιγότερα VM Exits απο το ballooning αφού διαχειρίζεται την μνήμη σε επίπεδο block (section). Ωστόσο, ένας από τους σημαντικότερους περιορισμούς του παραδοσιακού μηχανισμού memory hot(un)plug αφορά το granularity με το οποίο μπορεί να προσθέσει ή να αφαιρέσει μνήμη στην εικονική μηχανή. Σε αντίθεση με το memory ballooning, το οποίο βασίζεται στον page allocator του

guest λειτουργικού συστήματος και μπορεί θεωρητικά να λειτουργήσει σε πολύ μικρή κλίμακα, ο μηχανισμός hot(un)plug λειτουργεί σε πολύ μεγαλύτερα granularities, συνήθως της τάξης εκατοντάδων megabytes ή ακόμη και gigabytes. Η ελάχιστη δυνατή μονάδα μνήμης που μπορεί να προστεθεί ή να αφαιρεθεί (block) εξαρτάται από το guest OS. Για παράδειγμα, στο x86-64 Linux, το μικρότερο hot(un)pluggable memory block κυμαίνεται μεταξύ 128 MiB και 2 GiB, ανάλογα με τη ρύθμιση του συστήματος. Αυτός ο περιορισμός καθιστά το hot(un)plug ακατάλληλο για χρήση σε περιβάλλοντα που απαιτούν πιο fine-grained ελαστική μνήμη καθώς η χρήση του οδηγεί συχνά είτε σε υπερδέσμευση (over-provisioning), είτε σε υποχρησιμοποίηση (underprovisioning) της μνήμης [26].

2.3.3 virtio-mem: Αρχιτεκτονική

Τη λύση στους παραπάνω περιορισμούς έδωσε η ανάπτυξη του virtio-mem. Το virtio-mem ειναι μια paravirtualized συσκευή βασισμένη στο πρωτόκολλο VIRTIO σχεδιασμένη για fine-grained NUMA-aware ελαστική μνήμη [26].

Ο μηχανισμός virtio-mem επιτρέπει την έκθεση μιας δυναμικής ποσότητας μνήμης στην εικονική μηχανή (VM), μέσω ξεχωριστών virtio-mem συσκευών. Συνήθως, κάθε συσκευή αντιστοιχεί σε έναν εικονικό NUMA κόμβο, επιτρέποντας έτσι τη NUMA-aware διαχείριση της μνήμης [26].

Εννοιολογικά, κάθε virtio-mem συσκευή προσομοιώνεται ως ένα μεγάλο DIMM, το οποίο είναι διαιρεμένο σε blocks σταθερού μεγέθους. Κάθε block μπορεί να βρίσκεται σε μία από δύο καταστάσεις: plugged (ενεργό και διαθέσιμο στο VM) ή unplugged (μη διαθέσιμο). Η διαδικασία προσθήκης ή αφαίρεσης blocks συντονίζεται μέσω επικοινωνίας ανάμεσα στον hypervisor και το guest OS, χρησιμοποιώντας τη virtio-mem συσκευή ως διεπαφή. Οι αιτήσεις αλλαγής μεγέθους (resize requests) προέρχονται από τον hypervisor, ενώ το guest OS έχει την ευθύνη να επιλέξει ποια blocks θα γίνουν hotplugged ή hotunplugged ώστε να ικανοποιηθεί το αίτημα [26].

Η λειτουργία της συσκευής βασίζεται σε δύο βασικά properties: Το `plugged_size` που αντιπροσωπεύει τη μνήμη που είναι ήδη plugged στη συσκευή και ορατή στον guest, και το `requested_size` μέσω του οποίου ο hypervisor δηλώνει το επιθυμητό μέγεθος μνήμης για τη συσκευή. Το guest OS συγκρίνει αυτά τα δύο properties για να αποφανθεί αν πρέπει να προστεθούν η να αφαιρεθούν blocks μνήμης [26].

Για σκοπούς βελτιστοποίησης, ένα τμήμα της συνολικής περιοχής μνήμης που διαχειρίζεται η συσκευή μπορεί να ορίζεται ως usable region, δηλαδή περιοχή στην οποία επιτρέπεται το plugging νέων blocks. Ο hypervisor μπορεί να επεκτείνει δυναμικά αυτήν την περιοχή, για παράδειγμα όταν το `requested_size` αυξηθεί πέρα απο αυτή. Με τον τρόπο αυτό, το virtio-mem επιτρέπει σταδιακή και ελεγχόμενη επέκταση της διαθέσιμης μνήμης [26].

Το μέγεθος των blocks στην περιοχή μνήμης που διαχειρίζεται η συσκευή καθορίζεται από τον hypervisor, και κοινοποιείται στο guest μέσω ιδιότητας της συσκευής. Τα plugged blocks συμπεριφέρονται σαν κανονική φυσική μνήμη RAM, ενώ η πρόσβαση στα unplugged blocks θεωρείται αόριστη Παρ' όλα αυτά, σε ορισμένες περιπτώσεις, επιτρέπεται η ανάγνωση των unplugged blocks εντός της usable region, ώστε να διευκολύνεται η δημιουργία memory dumps ή snapshots [26].

2.3.4 Διαδικασία hot(un)plug στο virtio-mem

Όταν ο hypervisor επιθυμεί να αλλάξει το μέγεθος μνήμης του guest, επιλέγει μία από τις virtio-mem συσκευές (για παράδειγμα με βάση κάποια NUMA προτίμηση) και τροποποιεί το `requested_size`. Η αλλαγή αυτή ενεργοποιεί ένα σήμα ειδοποίησης προς τον guest, ενημερώνοντάς τον ότι η διαμόρωση της συσκευής έχει αλλάξει. Ο οδηγός του virtio-mem στο guest OS είναι υπεύθυνος να φέρει τη συσκευή στην επιθυμητή κατάσταση, επιλέγοντας τα κατάλληλα blocks για (un)plug. Στη συνέχεια στέλνει ένα αίτημα προς τη συσκευή το οποίο την πληροφορεί σχετικά με τα υποψήφια προς (un)plug blocks. Η συσκευή μπορεί να αποδεχθεί ή να απορρίψει το αίτημα, παρέχοντας αιτιολόγηση για την απόφασή της. Μια απόρριψη (reject) μπορεί να προκύψει, για παράδειγμα, όταν η συσκευή είναι απασχολημένη, όπως σε περιπτώσεις μετακίνησης ενός VM (live migration) όπου ο hypervisor δεν μπορεί προσωρινά να χειριστεί λειτουργίες (un)plug, ή όταν το plug νέων blocks θα είχε ως αποτέλεσμα να υπερβεί το συνολικό requested size που έχει οριστεί από τον hypervisor. Έπειτα απο έγκριση του αιτήματος:

- Στη περίπτωση του plug ο driver προχωράει σε προσθήκη των blocks στο λειτουργικό του guest.
- Στην περίπτωση του unplug, ο driver αφαιρεί τα blocks απο το λειτουργικό του guest, ενώ η συσκευή ενημερώνει τον host πως η unplugged περιοχή μνήμης δεν χρησιμοποιείται πλέον ώστε αυτός να την αξιοποιήσει σε άλλες διεργασίες.

2.3.5 Επικοινωνία μεταξύ device και guest OS (driver) μέσω του προτύπου VIRTIO

Ο μηχανισμός επικοινωνίας του προτύπου VIRTIO χρησιμοποιείται για την άμεση επικοινωνία του host με τον guest και αντίστροφα. Για παράδειγμα μέσω του virtqueue της συσκευής ο hypervisor πληροφορεί τον guest driver για μεταβολές του `requested_size`, ενώ ο guest driver χρησιμοποιεί το virtqueue για να στέλνει (un)plug αιτήματα στον hypervisor [26]. Τα αιτήματα που υποστηρίζονται αυτή τη στιγμή περιλαμβάνουν:

- το plug ή unplug ενός ή περισσότερων συνεχόμενων blocks,
- τον έλεγχο της κατάστασης (plugged, unplugged ή mixed) ενός ή περισσότερων συνεχόμενων blocks.

Όλες οι ιδιότητες της συσκευής (device properties) είναι ορατές στον driver του guest OS μέσω του VIRTIO configuration space, δηλαδή ενός ειδικού χώρου παραμέτρων που καθορίζει την κατάσταση και τις δυνατότητες της συσκευής.

2.4 Σχετική δουλειά

2.4.1 Squeezy: μηχανισμός αποδοτικής ανάκτησης μνήμης

Αναγνωρίζοντας τις αδυναμίες του hotunplug, και αξιοποιώντας το γεγονός ότι σε serverless περιβάλλοντα οι μέγιστες απαιτήσεις των workloads σε μνήμη καθορίζονται εκ των προ-

τέρων από τους χρήστες, ο μηχανισμός Squeezy αναπτύχθηκε με στόχο την πλήρη εξάλειψη των page migrations και του page zeroing.

Στην υλοποίηση του Squeezy το λειτουργικό σύστημα του guest τροποποιείται ώστε να χωρίζει λογικά την μνήμη σε ανεξάρτητα τμήματα. Κάθε συνάρτηση μπορεί να αιτηθεί, μέσω ενός ειδικού API, την τοποθέτηση ολόκληρου του αποτυπώματος μνήμης της σε ένα από αυτά τα τμήματα, ενώ κάθε τμήμα μπορεί να φιλοξενήσει το πολύ μια συνάρτηση. Με τον τρόπο αυτό, επιτυγχάνεται η απομόνωση των memory footprints των συναρτήσεων και αποφεύγεται η αλληλοκάλυψη τους. Επιπλέον το λειτουργικό σύστημα του guest γνωρίζει την ύπαρξη των τμημάτων αυτών ώστε όταν μια συνάρτηση τερματίσει, να κάνει απευθείας offline τα blocks μνήμης της. Ως αποτέλεσμα η ανάκτηση της μνήμης μετά τον τερματισμό μιας συνάρτησης πραγματοποιείται άμεσα και με μεγάλη ταχύτητα καθώς εξαλείφεται το overhead που εισήγαγαν τα migrations και το zeroing [3].

Κεφάλαιο 3

Υλοποίηση

Η υλοποίηση πραγματοποιήθηκε για την έκδοση 1.9 του Firecracker, χρησιμοποιώντας Ubuntu 22.04.5 rootfs και πυρήνα Linux 5.10. Η ανάπτυξη έγινε στη γλώσσα προγραμματισμού Rust, ενώ ο πηγαίος κώδικας είναι διαθέσιμος στο github: https://github.com/ntua-el20161/firecracker-public.git, στο branch virtio-mem-impl.

3.1 Διαμόρφωση της συσκευής

Όπως αναφέρθηκε στην υποενότητα 2.3.5, κάθε VIRTIO συσκευή διαθέτει ένα συγκεκριμένο χώρο διαμόρφωσης (configuration space) ο οποίος περιέχει πληροφορία για τα πεδία που σχετίζονται με την λειτουργικότητα της συσκευής. Ο σχεδιασμός πρέπει να συμμορφώνεται με τις απαιτήσεις των προδιαγραφών του VIRTIO [33], καθώς αυτά τα πεδία χρησιμοποιούνται από τον driver της συσκευής για τις διάφορες λειτουργίες της. Για το virtio-mem η διαμόρφωση είναι η εξής:

```
pub(crate) struct ConfigSpace {
        pub block_size: u64,
        pub node_id: u16,
        _padding: [u8; 6],
        pub addr: u64,
        pub region_size: u64,
        pub usable_region_size: u64,
        pub plugged_size: u64,
        pub requested_size: u6
}
```

- `block_size`: Το μέγεθος (σε bytes) ενός βασικού block μνήμης που μπορεί να προστεθεί ή να αφαιρεθεί στη συσκευή. Όλες οι λειτουργίες προσθήκης ή αφαίρεσης μνήμης γίνονται σε πολλαπλάσια του `block_size`. Το πεδίο αυτό είναι σταθερό κατά τη διάρκεια λειτουργίας της συσκευής.
- `node_id`: Ο αριθμός του NUMA node στον οποίο αντιστοιχεί η συσκευή.
- `padding`: Δεν χρησιμοποιείται και προορίζεται για μελλοντική χρήση.

- `addr`: Είναι φυσική διεύθυνση (σε bytes) στη μνήμη του guest απο την οποία αρχίζει η περιοχή μνήμης της συσκευής (device managed region). Το πεδίο αυτό είναι σταθερό κατά τη διάρκεια λειτουργίας της συσκευής.

- `region_size`: Είναι το μέγεθος του device managed region της συσκευής. Το πεδίο αυτό είναι σταθερό κατά τη διάρκεια λειτουργίας της συσκευής.

- `usable_region_size`: Είναι το μέγεθος της περιοχής του device managed region που επιτρέπεται το plug/unplug blocks μνήμης. Μπορεί να μεγαλώσει έως ότου εξισωθεί με το `region_size`. Μπορεί να ελατωθεί μόνο από unplug all requests.

- `plugged_size`: Είναι το μέγεθος (σε bytes) της μνήμης που είναι plugged μια δεδομένη στιγμή στη usable region.

- `requested_size`: Είναι το ζητούμενο μέγεθος μνήμης (σε bytes).

3.2 Δημιουργία και αρχικοποίηση μιας virtio-mem συσκευής στο Firecracker

Προκειμένου να είναι ορατή η συσκευή και η διαμόρφωση της μετά την εκκίνηση του microVM, προστίθεται ένα JSON object που περιέχει το configuration της συσκευής, στο αρχείο που χρησιμοποιείται για τη ρύθμιση του microVM κατά την εκκίνηση. Η δομή αυτού του object είναι η εξής:

```
"memory-devices": [
    {
        "id": ,
        "region_size_kib": ,
        "block_size_kib": ,
        "requested_size_kib":
    }
]
```

Στη συνέχεια, γίνεται parsing του JSON αρχείου και αρχικοποίηση της συσκευής virtio-mem με βάση τα πεδία του αρχείου, αφού προηγουμένως ελεγχθεί ότι οι τιμές τους δεν παραβιάζουν τους κανόνες που ορίζουν οι προδιαγραφές του virtio-mem [33]. Σε αυτό το στάδιο, δημιουργείται το configuration space της συσκευής, το virtqueue που χρησιμοποιείται για την επικοινωνία με τον guest, καθώς και ένα bitmap που διατηρεί την κατάσταση κάθε block (δηλαδή αν είναι plugged ή unplugged) μέσα στο device-managed region.

Τα πεδία για τα οποία δεν υπάρχει ακόμη διαθέσιμη πληροφορία (όπως το `addr` του configuration space) αρχικοποιούνται με μια αποδεκτή προσωρινή τιμή, ώστε να μπορεί η συσκευή να ολοκληρώσει τη διαδικασία κατασκευής και να είναι έτοιμη να δεχθεί αιτήματα από τον guest για την ενεργοποίησή της.

3.2.1 Δημιουργία και mapping της μνήμης του guest

Υπόβαθρο: GuestMemoryMmap

Στο Firecracker, η φυσική μνήμη του επισκέπτη (guest) αναπαρίσταται από τη δομή `GuestMemoryMmap`. Η δομή αυτή αποτελεί έναν χάρτη μνήμης που περιέχει πολλαπλές περιοχές μνήμης (`GuestRegionMmap`), καθεμία από τις οποίες αντιστοιχεί σε ένα συνεχόμενο τμήμα διευθύνσεων του guest.

Δημιουργία και mapping της guest memory

Στο επόμενο βήμα δημιουργείται το device managed region της virtio-mem συσκευής ως συλλογή αντικειμένων `GuestRegionMmap`. Οι περιοχές μνήμης αντιστοίζονται σε διευθύνσεις εικονικής μνήμης του host μέσω της κλήσης της `libc::mmap()`. Στη συνέχεια οι περιοχές μνήμης του virtio-mem ενώνονται με αυτές της υπόλοιπης μνήμης του guest, σε ένα κοινό vector απο `GuestRegionMmap`, και από αυτό το vector δημιουργείται το `GuestMemoryMmap` του VM. Ωστόσο η περιοχή μνήμης του virtio-mem δεν γίνεται onlined σε αυτο το στάδιο σε αντίθεση με την υπόλοιπη μνήμη και έτσι δεν είναι ορατή στο σύστημα.

3.2.2 MMIO transport

Το MMIO (Memory-Mapped I/O) είναι ένα μέσο ασύγχρονης επικοινωνίας ανάμεσα στον guest και τις virtio συσκευές. Σε κάθε virtio συσκευή παρέχεται μια virtio-mmio συσκευή. Κάθε τέτοια συσκευή διαθέτει ένα συγκεκριμένο εύρος μνήμης (MMIO region) το οποίο είναι μοιραζόμενο (shared) μεταξύ guest και VMM και χρησιμοποιείται για την επικοινωνία μεταξύ των δύο. Ο guest driver έχει πρόσβαση στους καταχωρητές της συσκευής (π.χ. device status, feature bits) διαβάζοντας και γράφοντας σε αυτή τη μνήμη.

Κατά την εκκίνηση του VM, το λειτουργικό σύστημα του guest ανιχνεύει το MMIO region, αναγνωρίζει τη virtio συσκευή, και ξεκινά τη διαδικασία διαπραγμάτευσης χαρακτηριστικών (feature negotiation) και δημιουργίας ουρών (virtqueues). Η διαδικασία συντονίζεται με το VMM μέσω του καταχωρητή `device_status`. Στο τέλος ο guest γράφει σε αυτόν τον καταχωρητή τιμή `DRIVER_OK` και η virtio συσκευή ενεργοποιείται. Απο εκείνο το σημείο και μετά η επικοινωνία μεταξύ driver και συσκευής γίνεται μέσω του virtqueue της συσκευής.

Στην υλοποίησή μας χρησιμοποιούμε το MMIO έναντι της εναλλακτικής του PCI, το οποίο δεν υποστηρίζεται απο το Firecracker στην έκδοση 1.9.

Σε αυτό το σημείο, εκκινώντας το microVM με τη συσκευή virtio-mem και ορίζοντας 1 GB μνήμης για αυτήν, μπορούμε να παρατηρήσουμε τη διαμόρφωση της μνήμης του guest μέσα απο ένα τερματικό της εικονικής μηχανής, χρησιμοποιώντας την εντολή `cat /proc/iomem`. Η έξοδος της εντολής δείχνει τη δομή της φυσικής μνήμης του guest:

```
root@ubuntu-fc-uvm:~# cat /proc/iomem
00000000-00000fff : Reserved
00001000-0009fbff : System RAM
0009fc00-000dffff : Reserved
000f0000-000fffff : System ROM
```

```
00100000-1fffffff : System RAM
  01000000-01a00c36 : Kernel code
  01c00000-01d86fff : Kernel rodata
  01e00000-01f0917f : Kernel data
  02272000-023fffff : Kernel bss
20000000-5fffffff : virtio0
d0000000-d0000fff : virtio-mmio.0
  d0000000-d0000fff : virtio-mmio.0 virtio-mmio.0
d0001000-d0001fff : virtio-mmio.1
  d0001000-d0001fff : virtio-mmio.1 virtio-mmio.1
d0002000-d0002fff : virtio-mmio.2
  d0002000-d0002fff : virtio-mmio.2 virtio-mmio.2
fee00000-fee00fff : Local APIC
```

Από αυτή τη δομή παρατηρούμε την περιοχή μνήμης που ανήκει στο virtio-mem η όποια είναι η virtio0, καθώς και τα 3 mmio regions που δημιουργήθηκαν για τα αντίστοιχα mmio devices τα οποία αντιστοιχούν στις συσκευές virtio-blk, virtio-mem και virtio-net.

3.3 Λειτουργία της συσκευής

3.3.1 Επέκταση του Firecracker API

Το Firecracker διαθέτει ένα RESTful HTTP-based API, που επιτρέπει τη ρύθμιση και διαχείριση του microVM. Μέσω του API, ο hypervisor μπορεί να λαμβάνει εντολές για τη δημιουργία και παραμετροποίηση συσκευών, τη διαχείριση της μνήμης, καθώς και για τον έλεγχο της κατάστασης της εικονικής μηχανής (start, pause, resume). Η επικοινωνία με το API του Firecracker πραγματοποιείται μέσω ενός Unix socket, το οποιο ο χρήστης μπορεί να καθορίσει κατα την εκκίνηση του microVM. Στο πλαίσιο της παρούσας υλοποίησης, επεκτάθηκε το υπάρχον API με δύο νέα requests.

Εισήχθη ένα `GET` request το οποίο επιστρέφει στον χρήστη το τρέχον configuration της συσκευής, περιλαμβάνοντας βασικά πεδία όπως το `id` της συσκευής, το `block_size`, το `node_id`, το `region_size`, και το `requested_size`. Η σύνταξη του request είναι η εξής:

```
curl --unix-socket $socket_location -i
-X GET 'http://localhost/memory-device
-H 'Accept: application/json'
```

Ένα παράδειγμα επιτυχούς εκτέλεσης είναι το εξής:

```
HTTP/1.1 200
Server: Firecracker API
Connection: keep-alive
Content-Type: application/json
Content-Length: 103
```

```
{
 "id":"memory-dev-1",
 "block_size_kib":512,
 "node_id":0,"region_size_kib":1048576,
 "requested_size_kib":0
}
```

Για την εξυπηρέτηση των hot(un)plug requests εισήχθη ένα PATCH request μέσω του οποίου ο χρήστης μπορεί να αλλάξει το `requested_size` της συσκευής. Η σύνταξη του request είναι η εξής:

```
curl --unix-socket $socket_location -i
-X PATCH 'http://localhost/memory-device
-H 'Accept: application/json'
-H 'Content-Type: application/json'
-d "{ "requested_size_kib": $requested_size }"
```

Κατα την επιτυχή εκτέλεση στον χρήστη επιστρέφεται το εξής:

```
HTTP/1.1 204
Server: Firecracker API
Connection: keep-alive
```

3.3.2 (Un)Plug Request

Όταν ο χρήστης αποστείλει ένα (un)plug request μέσω του API του Firecracker, ξεκινά μια ακολουθία ενεργειών που καταλήγει στην ενημέρωση της μνήμης του guest.

Αλλαγή στη διαμόρφωση της συσκευής απο το VMM

Το Firecracker ενημερώνει την τιμή `requested_size` στο configuration της συσκευής virtio-mem. Μετά απο την μεταβολή το VMM είναι υπεύθυνο να στείλει μια διακοπή προς τον guest ειδοποιώντας τον οτι η διαμόρφωση της συσκευής έχει αλλάξει.

Επεξεργασία από τον guest driver

Ο driver του guest, συγκρίνοντας τις τιμές `requested_size` και `plugged_size`, καθορίζει αν πρόκειται για plug ή unplug request. Στη συνέχεια, επιχειρεί να εντοπίσει τα κατάλληλα blocks μνήμης - είτε unplugged (στην περίπτωση plug) είτε plugged (στην περίπτωση unplug) — με στόχο να πετύχει να εξισώσει το `plugged_size` με το `requested_size`. Αφού επιλέξει τα κατάλληλα blocks, ο driver αποστέλλει προς τη συσκευή requests μέσω της virtqueue, τα οποία περιγράφουν τις φυσικές διευθύνσεις και τα μεγέθη των περιοχών μνήμης που προτείνει να γίνουν (un)plug. Στη συνέχεια ειδοποιεί τη συσκευή για την ύπαρξη νέων αιτημάτων και κάνοντας vm exit αναμένει την απάντησή της. Ο driver στέλνει στη συσκευή requests σε μονάδες του ενός Linux memory block (128MB), εφόσον αυτό είναι δυνατό. Τα requests του guest έχουν την εξής μορφή:

```
struct virtio_mem_req {
    le16 type;
    le16 padding[3];
    union {
        struct virtio_mem_req_plug plug;
        struct virtio_mem_req_unplug unplug;
        struct virtio_mem_req_state state;
    } {u};
}
```

- type: Ο τύπος του request. Οι δυνατοί τύποι είναι οι εξής:
  - VIRTIO_MEM_REQ_PLUG: 0
  - VIRTIO_MEM_REQ_UNPLUG: 1
  - VIRTIO_MEM_REQ_UNPLUG_ALL: 2
  - VIRTIO_MEM_REQ_STATE: 3
- padding: Δεν χρησιμοποιείται και πρέπει να αγνοηθεί απο τη συσκευή
- u: Δεδομένα ανάλογα το request. Έχουν την εξής δομή:

```
struct virtio_mem_req_plug {
    le64 addr;
    le16 nb_blocks;
    le16 padding[3];
}
```

Επεξεργασία του request από τον VMM

Ο VMM λαμβάνει το request από το virtqueue. Στην υλοποίηση μας το request μεταφράζεται σε μια εσωτερική δομή δεδομένων για ευκολία αξιοποίησης. Η δομή είναι εντελώς ανάλογη με αυτή του guest και είναι η εξής:

```
#[repr(C)]
#[derive(Clone, Copy, Debug, Default)]
pub struct VirtioMemReqData {
    pub addr: u64,
    pub nb_blocks: u16,
    pub padding: [u16; 3],
}

#[repr(C)]
#[derive(Clone, Copy, Debug, Default)]
pub struct VirtioMemReq {
    pub req_type: u16,
```

```
    pub padding: [u16; 3],
    pub data: VirtioMemReqData // the union field
}
```

Ανάλογα με τον τύπο του request καλείται ο αντίστοιχος handler. Στους handlers ελέγχεται η εγγυρότητα των δεδομένων, όπως έαν το μέγεθος της δοσμένης περιοχής μνήμης του request υπερβαίνει την προκαθορισμένη περιοχή μνήμης της συσκευής ή εάν η διεύθυνση του πρώτου blocks στο request είναι πολλαπλάσιο του μεγέθους των block της συσκευής. Στη συνέχεια, ελέγχεται η κατάσταση των blocks που περιγράφονται από το αίτημα, στο bitmap της συσκευής: αν πρόκειται για plug request, το VMM επιβεβαιώνει ότι τα blocks είναι πράγματι unplugged, ενώ στην περίπτωση unplug request ελέγχει ότι τα blocks είναι plugged.

Στην περίπτωση του unplug request, το VMM οφείλει να ενημερώσει τον hypervisor ότι τα συγκεκριμένα blocks δεν χρησιμοποιούνται πλέον, ώστε να μπορούν να χρησιμοποιηθούν από άλλες διεργασίες. Για τον σκοπό αυτό γίνεται κλήση στο σύστημα μέσω της συνάρτησης `madvise()` με την επιλογή `MADV_DONTNEED`:

```
libc::madvise(
    (self.host_addr + offset) as *mut libc::c_void,
    unplug_size as libc::size_t,
    libc::MADV_DONTNEED
)
```

Μετά την επιτυχή εκτέλεση του αιτήματος, ενημερώνονται το bitmap και η τιμή του `plugged_size` ώστε να αντικατοπτρίζουν τη νέα κατάσταση της συσκευής. Στη συνέχεια, το Firecracker εγγράφει στο virtqueue μια απάντηση (`VirtioMemResp`) προς τον guest driver, η οποία έχει την εξής μορφή:

```
#[repr(C)]
#[derive(Clone, Copy, Debug, Default)]
pub struct VirtioMemResp {
    pub resp_type: u16,
    pub padding: [u16; 3],
    pub state: u16,
}
```

Το πεδίο `resp_type` μπορεί να λάβει μία από τις ακόλουθες τιμές:

- 0: **ACK** – επιτυχής ολοκλήρωση του αιτήματος,
- 1: **NACK** – απόρριψη του αιτήματος,
- 2: **BUSY** – η συσκευή είναι απασχολημένη,
- 3: **ERROR** – σφάλμα κατά την επεξεργασία.

Το πεδίο state αφορά το state request και περιγράφει την κατάσταση των δοσμένων blocks. Τέλος, το VMM ενεργοποιεί ένα interrupt προς τον guest driver, ενημερώνοντάς τον ότι η απάντηση είναι διαθέσιμη στο virtqueue. Με αυτόν τον τρόπο ολοκληρώνεται ο κύκλος επικοινωνίας για κάθε (un)plug request.

Λήψη απάντησης απο τον guest

Μόλις ο guest driver λάβει μια θετική απάντηση (ACK) από το VMM, προχωρά σε plug ή unplug των υποψήφιων blocks χρησιμοποιώντας τον τυπικό μηχανισμό hot(un)plugging του Linux.

3.3.3 State Request

Η ροή εκτέλεσης ενός state request είναι παρόμοια με αυτή των υπολοίπων αιτημάτων (plug/unplug). Σε αυτό το είδος request, ο guest driver ζητά την τρέχουσα κατάσταση των blocks μνήμης εντός μιας συγκεκριμένης περιοχής. Το VMM απαντά ενημερώνοντας τον guest μέσω του πεδίου state της δομής `VirtioMemResp`, το οποίο υποδεικνύει την κατάσταση των blocks στο ζητούμενο range. Οι δυνατές τιμές του πεδίου state είναι:

- 0 – `VIRTIO_MEM_STATE_PLUGGED`: όλα τα blocks στην περιοχή είναι plugged,
- 1 – `VIRTIO_MEM_STATE_UNPLUGGED`: όλα τα blocks στην περιοχή είναι unplugged,
- 2 – `VIRTIO_MEM_STATE_MIXED`: στο range συνυπάρχουν plugged και unplugged blocks.

3.3.4 Παράδειγμα Εκτέλεσης

Κατά την εκκίνηση του VM, ορίζουμε για το virtio-mem μέγεθος μνήμης 1 GB, ενώ για τη βασική (προκαθορισμένη) μνήμη του guest παρέχουμε 512 MB. Επιπλέον, ζητάμε το αρχικό μέγεθος της "plugged" μνήμης (requested size) να είναι 512 MB. Η ρύθμιση στο αρχείο διαμόρφωσης του Firecracker έχει ως εξής:

```
"machine-config": {
    "vcpu_count": 2,
    "mem_size_mib": 512,
    "smt": false,
    "track_dirty_pages": false
},

"memory-devices": [
    {
        "id": "memory-dev-1",
        "region_size_kib": 1048576,
        "block_size_kib": 2048,
        "requested_size_kib": 524288
    }
],
```

Μετά την εκκίνηση του VM, εκτελούμε στο τερματικό του guest την εντολή `free -h` και παρατηρούμε ότι η συνολική μνήμη που αναγνωρίζει το λειτουργικό σύστημα είναι 1 GiB — δηλαδή 512 MB από τη βασική μνήμη του guest και 512 MB από τη μνήμη που προστέθηκε δυναμικά μέσω του virtio-mem:

```
root@ubuntu-fc-uvm:~# free -h
total used free shared buff/cache available
Mem: 1.0Gi 34Mi 929Mi 4.0Mi 36Mi 943Mi
Swap: 0B 0B 0B
```

Στη συνέχεια, θα δοκιμάσουμε να εξαντλήσουμε τη διαθέσιμη μνήμη του virtio-mem, πραγματοποιώντας plug και για τα υπόλοιπα 500 MB. Από τον host στέλνουμε ένα API request, ζητώντας το `requested_size` να αυξηθεί σε 1 GB. Καλώντας εκ νέου την `free -h` παρατηρούμε ότι το συνολικό μέγεθος μνήμης του VM είναι 1.5GB όπως περιμέναμε:

```
root@ubuntu-fc-uvm:~# free -h
total used free shared buff/cache available
Mem: 1.5Gi 42Mi 1.4Gi 4.0Mi 36Mi 1.4Gi
Swap: 0B 0B 0B
```

Για να επαληθεύσουμε ότι το σύστημα μπορεί πράγματι να χρησιμοποιήσει τη μνήμη του virtio-mem, εκτελούμε στο guest το ακόλουθο πρόγραμμα σε C, το οποίο δεσμεύει και "αγγίζει" 1 GiB μνήμης (γράφοντας σε κάθε σελίδα):

```c
#include <stdio.h>
#include <stdlib.h>

#define SIZE (1024UL*1024*1024) // 1 GiB
#define PAGE_SIZE 4096

int main() {
    printf("Allocating 1 GiB...\n");
    unsigned char *mem = malloc(SIZE);
    if (!mem) {
        perror("malloc");
        return 1;
    }
    printf("Touching memory...\n");
    for (size_t i = 0; i < SIZE; i += PAGE_SIZE) {
        mem[i] = 1;  // write a byte per page
    }

    printf("Done. Press Enter to free memory...\n");
    getchar();

    system("free -h");

    free(mem);
    return 0;
}
```

Κατά την εκτέλεση του προγράμματος, παρατηρούμε την εξής έξοδο:

```
root@ubuntu-fc-uvm:~# ./touch_mem
Allocating 1 GiB...
Touching memory...
```

Πατώντας `enter`, η εντολή `free -h` εμφανίζει τα εξής:

```
total used free shared buff/cache available
Mem: 1.5Gi 1.0Gi 401Mi 4.0Mi 39Mi 410Mi
Swap: 0B 0B 0B
```

Η σημαντική μείωση της διαθέσιμης μνήμης κατά περίπου 1 GiB επιβεβαιώνει ότι το πρόγραμμα πραγματικά προσπέλασε φυσική μνήμη του guest. Δεδομένου ότι η βασική μνήμη του VM είναι μόλις 512 MB, αυτό σημαίνει πως μέρος της δεσμευμένης μνήμης προήλθε από τη virtio-mem περιοχή.

3.4 Υλοποίηση Υποστήριξης Snapshot για τη Συσκευή

3.4.1 Υλοποίηση της δομής `Persist` για το virtio-mem

Η λειτουργικότητα snapshot απαιτεί η συσκευή να μπορεί να αποθηκεύει και να επαναφέρει την εσωτερική της κατάσταση (state). Για το σκοπό αυτό, υλοποιήθηκε η δομή `Persist` για το virtio-mem, η οποία καθορίζει τα πεδία που περιγράφουν πλήρως το runtime state της συσκευής, ενώ επίσης παρέχει τις μεθόδους `save` και `restore` που επιτρέπουν την αποθήκευση και την ανάκτηση αντίστοιχα, της κατάστασης της συσκευής.

Κατάσταση της συσκευής

Η κατάσταση της συσκευής αποθηκεύεται στη δομή VirtioMemState:

```
#[derive(Debug, Clone, Serialize, Deserialize)]
struct VirtioMemConfigSpaceState {
    block_size: u64,
    node_id: u16,
    addr: u64,
    region_size: u64,
    usable_region_size: u64,
    plugged_size: u64,
    requested_size: u64
}

#[derive(Debug, Clone, Serialize, Deserialize)]
pub struct VirtioMemState {
    virtio_state: VirtioDeviceState,
    config_space: VirtioMemConfigSpaceState,
```

```
    id: String,
    memory_bitmap: MemBitmap,
    host_addr: u64,
}
```

Το trait Persist ορίζει έναν τύπο State ο οποίος αντιστοιχεί στον τύπο VirtioMemState.

Μέθοδος save()

Η συνάρτηση save() υλοποιεί τη λογική συλλογής της κατάστασης της συσκευής και επιστρέφει ένα αντικείμενο τύπου VirtioMemState. Ο MMIO Device Manager καλεί αυτή τη συνάρτηση κατά τη δημιουργία snapshot, ώστε να αποθηκευτεί το state στο συνολικό αρχείο snapshot.

```
fn save(&self) -> Self::State {
    VirtioMemState {
        virtio_state: VirtioDeviceState::from_device(self),
        config_space: VirtioMemConfigSpaceState {
            block_size: self.block_size(),
            node_id: self.node_id(),
            addr: self.config_space.addr,
            region_size: self.config_space.region_size,
            usable_region_size: self.config_space.usable_region_size,
            plugged_size: self.config_space.plugged_size,
            requested_size: self.config_space.requested_size
        },
        id: self.id.clone(),
        host_addr: self.host_addr,
        memory_bitmap: self.memory_bitmap.clone()
    }
}
```

Μέθοδος restore()

Η μέθοδος restore() εκτελείται κατά τη φόρτωση ενός snapshot. Λαμβάνει ως είσοδο το αντικείμενο VirtioMemState και αποκαθιστά τα αποθηκευμένα πεδία σε ένα νέο instance της συσκευής:

```
fn restore(
guest_memory: Self::ConstructorArgs,
state: &Self::State,
) -> Result<Self, Self::Error> {
    let mut virtio_mem = Memory::new(
        state.config_space.block_size,
        Some(state.config_space.node_id),
```

```
            state.config_space.region_size,
            state.id.clone(),
            state.config_space.requested_size
        )?;
        virtio_mem.queues = state
            .virtio_state
            .build_queues_checked(
                &guest_memory.mem,
                TYPE_MEMORY,
                1,
                FIRECRACKER_MAX_QUEUE_SIZE
            )
            .map_err(|_| Self::Error::QueueRestoreError)?;
        virtio_mem.irq_trigger.irq_status =
            Arc::new(AtomicU32::new(state.virtio_state.interrupt_status));
        virtio_mem.avail_features = state.virtio_state.avail_features;
        virtio_mem.acked_features = state.virtio_state.acked_features;
        virtio_mem.config_space.addr = state.config_space.addr;
        virtio_mem.config_space.plugged_size = state.config_space.plugged_size;
        virtio_mem.addr_is_set = true;
        virtio_mem.host_addr_is_set = true;
        virtio_mem.host_addr = state.host_addr;
        virtio_mem.memory_bitmap = state.memory_bitmap.clone();
        if state.virtio_state.activated {
            virtio_mem.device_state = DeviceState::Activated(guest_memory.mem);
        }
        Ok(virtio_mem)
    }
```

Γνωστοποίηση της συσκευής στον MMIO Device Manager

Για να μπορέσει η συσκευή να συμπεριληφθεί στη διαδικασία snapshot του Firecracker, έπρεπε να γίνει γνωστή στον MMIO Device Manager. Έτσι έγινε εγγραφή της συσκευής στη δομή `DeviceStates`, η οποία χρησιμοποιείται για να κρατάει τα states των συσκευών που υποστηρίζουν το snapshot, μαζί με τα states των MMIO transport των συσκευών.

```
/// Holds the device states.
#[derive(Debug, Default, Clone, Serialize, Deserialize)]
pub struct DeviceStates {
#[cfg(target_arch = "aarch64")]
    // State of legacy devices in MMIO space.
    pub legacy_devices: Vec<ConnectedLegacyState>,
    /// Block device states.
    pub block_devices: Vec<ConnectedBlockState>,
```

```
    /// Net device states.
    pub net_devices: Vec<ConnectedNetState>,
    /// Vsock device state.
    pub vsock_device: Option<ConnectedVsockState>,
    /// Balloon device state.
    pub balloon_device: Option<ConnectedBalloonState>,
    /// Memory device state.
    pub virtio_mem_device: Option<ConnectedVirtioMemState>,
    /// Mmds version.
    pub mmds_version: Option<MmdsVersionState>,
    /// Entropy device state.
    pub entropy_device: Option<ConnectedEntropyState>,
}
```

Η δομή `ConnectedVirtioMemState` είναι η εξής:

```
/// Holds the state of a virtio-mem device connected to the MMIO space.
#[derive(Debug, Clone, Serialize, Deserialize)]
pub struct ConnectedVirtioMemState {
    /// Device identifier.
    pub device_id: String,
    /// Device_state.
    pub device_state: VirtioMemState,
    /// Mmio transport state.
    pub transport_state: MmioTransportState,
    /// VmmResources
    pub device_info: MMIODeviceInfo,
}
```

Στη συνέχεια, εντελώς ανάλογα με τις υπόλοιπες συσκευές, συμπεριλαμβάνονται στη διαδικασία αποθήκευσης και ανάκτησης οι κλήσεις στις μεθόδους `save()` και `restore()` του virtio-mem.

3.4.2 Παράδειγμα εκτέλεσης

Σε συνέχεια του προηγούμενου παραδείγματος, πραγματοποιήθηκε έλεγχος της λειτουργίας snapshot για τη συσκευή virtio-mem. Αρχικά, εκτελέστηκε το σενάριο plug επιπλέον 1 GB μνήμης στο VM, και έγινε allocation και εγγραφή σε 1 GB μνήμης, όπως περιγράφηκε στην προηγούμενη ενότητα. Στη συνέχεια ακολουθήθηκε η διαδικασία λήψης snapshot σύμφωνα με το επίσημο documentation του Firecracker [7]. Έπειτα, εκκινήθηκε μια νέα εικονική μηχανή, χωρίς τη χρήση configuration file, και στη συνέχεια πραγματοποιήθηκε φόρτωση του snapshot μέσω του Firecracker API. Από τα logs του νέου instance επιβεβαιώνεται ότι η συσκευή virtio-mem ανακατασκευάστηκε επιτυχώς:

```
2025-10-10T17:02:46.388009085 [anonymous-instance:fc_api] The API server
↪  received a Put request
```

```
on "/snapshot/load" with body "{\n "snapshot_path": "./snapshot_file",\n
↪  "mem_backend": {\n "backend_path": "./mem_file",\n "backend_type":
↪  "File"\n },\n "enable_diff_snapshots": true,\n "resume_vm": false\n }".
2025-10-10T17:02:46.388173859 [anonymous-instance:main] [DevPreview]
↪  Virtual machine snapshots is in development preview.
2025-10-10T17:02:46.390812556 [anonymous-instance:main] Host CPU vendor ID:
↪  [71, 101, 110, 117, 105, 110, 101, 73, 110, 116, 101, 108]
2025-10-10T17:02:46.390846805 [anonymous-instance:main] Snapshot CPU vendor
↪  ID: [71, 101, 110, 117, 105, 110, 101, 73, 110, 116, 101, 108]
2025-10-10T17:02:46.394813474 [anonymous-instance:main] Memory::new(524288,
↪  Some(0), 1073741824, memory-dev-1)
2025-10-10T17:02:46.395042317 [anonymous-instance:main] new mmio transport
↪  for device 18
2025-10-10T17:02:46.395265642 [anonymous-instance:main] Memory device
↪  [memory-dev-1].init()
2025-10-10T17:02:46.395293584 [anonymous-instance:main]
↪  Memory.register_activate_event()
2025-10-10T17:02:46.395602439 [anonymous-instance:main] new mmio transport
↪  for device 2
2025-10-10T17:02:46.398870708 [anonymous-instance:main] new mmio transport
↪  for device 1
2025-10-10T17:02:46.400853713 [anonymous-instance:main] [DevPreview]
↪  Virtual machine snapshots is in development preview - 'load snapshot'
↪  VMM action took 12640 us.
2025-10-10T17:02:46.400989792 [anonymous-instance:fc_api] The request was
↪  executed successfully. Status code: 204 No Content.
2025-10-10T17:02:46.401034105 [anonymous-instance:fc_api] 'load snapshot'
↪  API request took 13064 us.
```

Στο τερματικό του guest φαίνεται ότι το πρόγραμμα `touch_mem` βρισκόταν σε αναμονή εισόδου, όπως πριν τη λήψη του snapshot. Μετά το resume της εικονικής μηχανής, δόθηκε ένας χαρακτήρας εισόδου στο πρόγραμμα `touch_mem`, το οποίο συνέχισε την εκτέλεσή του και εμφάνισε εκ νέου τη μνήμη του guest. Η έξοδος της εντολής free -h επιβεβαίωσε ότι η κατάσταση της μνήμης παρέμεινε ακριβώς η ίδια με αυτή που υπήρχε πριν τη λήψη του snapshot:

```
total used free shared buff/cache available
Mem: 1.5Gi 1.0Gi 406Mi 4.0Mi 37Mi 413Mi
Swap: 0B 0B 0B
```

Η παραπάνω παρατήρηση επιβεβαιώνει ότι η συσκευή virtio-mem επαναφέρθηκε επιτυχώς μαζί με την κατάσταση της μνήμης που είχε αποδοθεί στον guest πριν τη λήψη του snapshot.

Κεφάλαιο 4

Αξιολόγηση

Για την αξιολόγηση της απόδοσης του virtio-mem στο Firecracker, πραγματοποιήθηκε μια σειρά πειραμάτων που μετρούν τον συνολικό χρόνο που απαιτείται για την εκτέλεση του unplug διαφορετικών μεγεθών μνήμης. Σύμφωνα με μετρήσεις [3] τα plug requests δεν εισάγουν σημαντικό overhead και για αυτό το λόγο δεν έγινε κάποια μέτρηση πάνω σε αυτά. Μετρήθηκαν επίσης και οι σημαντικότερες συνιστώσες που αποτελούν τον χρόνο ενός unplug request οι οποίες είναι ο χρόνος που απαιτείται για migrations, zeroing και VM exits. Ως σημεία αναφοράς χρησιμοποιήθηκαν αντίστοιχες μετρήσεις για το virtio-mem του Cloud Hypervisor VMM, καθώς και για τη συσκευή virtio-balloon του Firecracker. Επιπλέον, η συσκευή δοκιμάστηκε με τις ίδιες μετρήσεις και σε squeezy guest πυρήνα.

4.1 Σύστημα αξιολόγησης

4.1.1 Πειραματική διάταξη

- Όλες οι μετρήσεις έγιναν σε υπολογιστή με επεξεργαστή Intel Core i3 και 12GB RAM.
- Για την πραγματοποίηση των μετρήσεων που αφορούν το Firecracker, χρησιμοποιήθηκε η έκδοση v1.9, πάνω στην οποία υλοποιήθηκε η υποστήριξη του virtio-mem. Αντίστοιχα, για τις μετρήσεις του Cloud Hypervisor χρησιμοποήθηκε η έκδοση v44.0.
- Και στα δύο περιβάλλοντα, οι εικονικές μηχανές εκκινήθηκαν με δύο εικονικούς επεξεργαστές (vCPUs), ενώ ο μηχανισμός Transparent Huge Pages (THP) επιλέχθηκε να είναι ανενεργός.
- Για τα πειράματα χρησιμοποιήθηκε το memhog microbenchmark, ένα ελαφρύ εργαλείο που κάνει mmap και memset το μέγεθος μνήμης που του υποδεικνύεται.

4.1.2 Μέθοδος μέτρησης των συνιστωσών του χρόνου ανάκτησης μνήμης

Όλες οι μετρήσεις έγιναν κατόπιν τροποποιήσεων στην έκδοση 6.6.30 του πυρήνα του Linux:

- Για την μέτρηση του συνολικού χρόνου που απαιτεί ένα unplug request μετρήθηκε ο χρόνος εκτέλεσης της συνάρτησης `virtio_mem_unplug_request()`

- Για τη μέτρηση του χρόνου που απαιτούν τα page migrations μετρήθηκε ο χρόνος εκτέλεσης της συνάρτησης `migrate_pages()`
- Για τη μέτρηση του χρόνου που απαιτεί η αρχικοποίηση σελίδων (zeroing) μετρήθηκε ο χρόνος εκτέλεσης της συνάρτησης `kernel_init_pages()`.
- Για τη μέτρηση του χρόνου που απαιτούν τα vm-exits στο virtio-mem μετρήθηκε ο χρόνος της συναρτήσεων `virtio_mem_sbm_unplug_sb()` και `virtio_mem_bbm_unplug_bb()` που στέλνουν στον VMM τα υποψήφια προς unplugging blocks μνήμης για τις περιπτώσεις λειτουργίας subblock mode (`block_size` μικρότερο απο 128Mib) και big block mode (`block_size` μεγαλύτερο ή ίσο με 128Mib) αντίστοιχα. Για το virtio balloon μετρήθηκε ο χρόνος που απαιτεί η συνάρτηση `tell_host()`.

4.1.3 Μεθοδολογία του πειράματος

Εκκινούμε την εικονική μηχανή με αρχικό μέγεθος μνήμης ίσο με 1GiB, και με δυνατότητα δυναμικής προσθήκης μνήμης ίσης με το μέγεθος που επιθυμούμε να ανακτήσουμε σε επόμενο στάδιο. Κατα την εκκίνηση κάνουμε hotplug αυτή τη μνήμη στην εικονική μηχανή. Στις μετρήσεις που αφορούν τον μηχανισμό virtio-balloon, η εικονική μηχανή εκκινείται με αρχικό συνολικό μέγεθος μνήμης ίσο με 1 GiB, προσαυξημένο κατά το μέγεθος της μνήμης που πρόκειται να ανακτηθεί στη συνέχεια. Κατά την εκκίνηση, το balloon βρίσκεται σε πλήρως αποσυμπιεσμένη (deflated) κατάσταση, ώστε όλη η διαθέσιμη μνήμη να είναι αρχικά προσβάσιμη από το λειτουργικό σύστημα του guest.

Εκτελούμε 2 στιγμιότυπα memhog στον guest απο τα οποία το ένα δεσμεύει μέγεθος μνήμης ίσο με 1Gib δηλαδή ίσο με το αρχικό μέγεθος μνήμης του guest που χρησιμοποιούμε σε όλα τα πειράματα. Το δεύτερο στιγμιότυπο δεσμεύει μνήμη ίση με το μέγεθος που πρόκειται να ανακτηθεί.

Στη συνέχεια τερματίζουμε το δεύτερο στιγμιότυπο και κάνουμε reclaim την μνήμη που κάλυπτε. Με αυτό τον τρόπο μπορούμε να μετρήσουμε ικανοποιητικά τα migrations για τις virtio-mem συσκευές καθώς περιμένουμε τα 2 memhog στιγμιότυπα να είναι αλληλοκαλυπτούμενα.

Στα αποτελέσματα παρουσιάζεται ο μέσος όρος 10 μετρήσεων για τα 4 διαφορετικά μεγέθη ανάκτησης μνήμης.

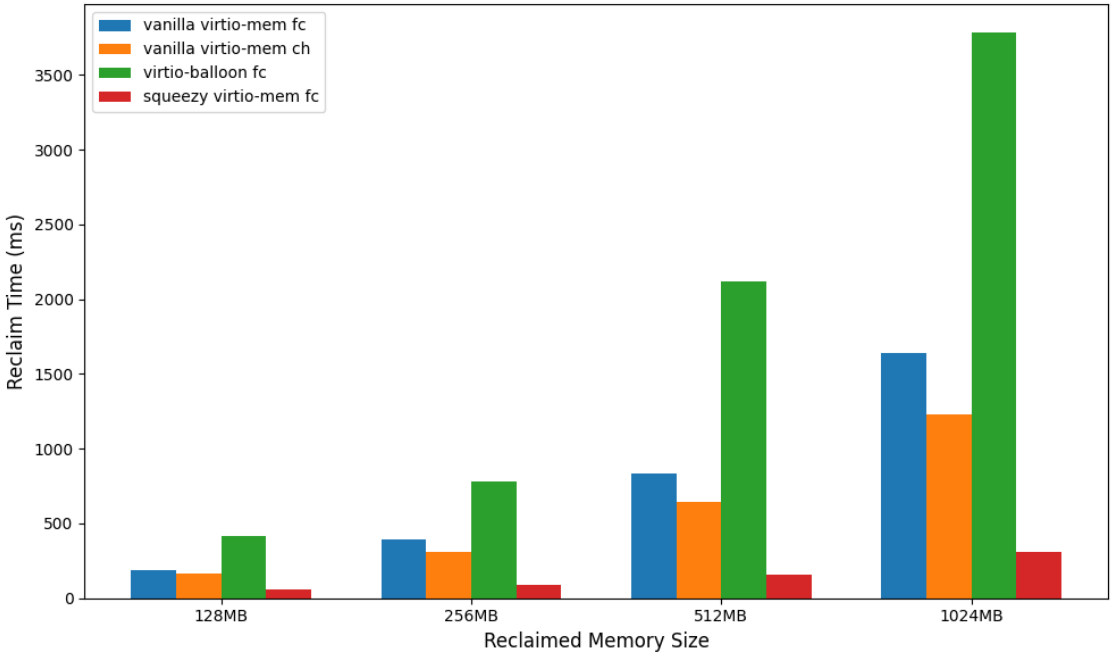
4.2 Αποτελέσματα

4.2.1 Συνολικός χρόνος ανάκτησης μνήμης

Στο Σχήμα 4.1 παρουσιάζονται οι μέσοι χρόνοι ανάκτησης μνήμης για τέσσερα διαφορετικά μεγέθη μνήμης:

Σύγκριση του virtio-mem του Firecracker με αυτό του Cloud Hypervisor

Αρχικά, παρατηρείται ότι το virtio-mem στο Firecracker παρουσιάζει χαμηλότερη απόδοση σε σύγκριση με την αντίστοιχη υλοποίηση του Cloud Hypervisor. Συγκεκριμένα, το virtio-



Σχήμα 4.1: *Συνολικος χρόνος ανάκτησης διαφορετικών μεγεθών μνήμης*

mem του Cloud Hypervisor είναι κατά μέσο όρο 1,27 φορές ταχύτερο από την υλοποίηση σε Firecracker. Η μεγαλύτερη διαφορά εντοπίζεται στη δοκιμή ανάκτησης μνήμης μεγέθους 1024MB, όπου το virtio-mem του Cloud Hypervisor είναι περίπου 1,33 φορές ταχύτερο, εκτελώντας τη διαδικασία reclamation σε περίπου 1,2s, έναντι 1,6s στο Firecracker. Μια πιο λεπτομερής ανάλυση των αποτελεσμάτων, καθώς και μια διερεύνηση των πιθανών αιτιών που οδηγούν σε αυτή τη διαφορά απόδοσης, παρουσιάζονται στη συνέχεια κατά την εξέταση του Σχήματος 4.2.

Σύγκριση του virtio-mem με το virtio-balloon

Όπως ήταν αναμενόμενο, ο μηχανισμός virtio-balloon παρουσιάζει σημαντικά χαμηλότερη απόδοση σε σχέση με το virtio-mem. Συγκεκριμένα, είναι κατά μέσο όρο 2,24 φορές πιο αργός από το virtio-mem του Firecracker και 2,86 φορές πιο αργός από το virtio-mem του Cloud Hypervisor. Όπως αναφέρθηκε στην υποενότητα 2.3.2, η μεγάλη αυτή διαφορά οφείλεται στον τρόπο ανάκτησης της μνήμης: το balloon ανακτά τη μνήμη σε επίπεδο σελίδων, γεγονός που αυξάνει σημαντικά το χρόνο που αφιερώνει σε vm exits.

Η διαφορά στην απόδοση μεταξύ virtio-balloon και virtio-mem γίνεται ιδιαίτερα εμφανής κατά την ανάκτηση 1024MB μνήμης. Συγκεκριμένα, το virtio-balloon απαιτεί σχεδόν 3,8s για να ολοκληρώσει τη διαδικασία, ενώ το virtio-mem χρειάζεται μόλις περίπου 1,2s στο Cloud Hypervisor και 1,6s στο Firecracker.

Ο οδηγός του virtio-mem αποστέλλει τα υποψήφια blocks προς αποδέσμευση (unplug) στη συσκευή σε granularity του Linux memory block, εφόσον αυτό είναι εφικτό — δηλαδή όταν η ζητούμενη μνήμη είναι τουλάχιστον 128MB και όλα τα subblocks του εξεταζόμενου memory block είναι plugged (ο driver εξετάζει πρώτα πλήρως plugged blocks πριν καταφύγει σε μερικώς plugged blocks). Επειδή στα πειράματά μας η μνήμη ζητείται σε πολλαπλάσια των

128MB, ο οδηγός αποστέλλει πάντα τα αιτήματα με blocks αυτού του μεγέθους. Αντίθετα, ο οδηγός του virtio-balloon στέλνει στη συσκευή τις σελίδες που πρόκειται να ανακτηθούν σε buffers των 256 PFNs.

Αυτό σημαίνει ότι, για την αποδέσμευση 128MB μέσω virtio-mem, ένα μόνο VM exit είναι αρκετό για την ικανοποίηση του αιτήματος από τον οδηγό, ενώ στην περίπτωση του balloon, δεδομένου ότι κάθε σελίδα έχει μέγεθος 4KiB, απαιτούνται συνολικά 128 VM exits για τον ίδιο όγκο μνήμης. Αντίστοιχα, για reclamation 1024MB, ο αριθμός των VM exits ανέρχεται σε 8 για το virtio-mem και σε 1024 για το balloon.

Αξιολόγηση της υλοποίησης με χρήση squeezy πυρήνα

Όπως ήταν αναμενόμενο, ο μηχανισμός squeezy στον πυρήνα του guest καθιστά το virtio-mem ιδιαίτερα αποδοτικό. Η εξάλειψη των migrations και του zeroing μειώνει δραστικά το υπολογιστικό κόστος της διαδικασίας ανάκτησης μνήμης, βελτιώνοντας σημαντικά την απόδοση σε σχέση με το virtio-mem όταν δεν χρησιμοποιείται σχυεεζψ πυρήνας.

Συγκεκριμένα, το virtio-mem του Firecracker με squeezy πυρήνα είναι κατά μέσο όρο 4,16 φορές ταχύτερο από το virtio-mem του Firecracker με μη τροποποιημένο πυρήνα. Η διαφορά αυτή αυξάνεται σημαντικά όσο μεγαλώνει το μέγεθος της μνήμης που ανακτάται· στα 128MB, το squeezy είναι 3,36 φορές ταχύτερο, ενώ στα 1024MB φτάνει να είναι 5,25 φορές ταχύτερο. Το γεγονός αυτό ήταν αναμενόμενο αφού σε κάθε δοκιμή διπλασιάζεται το μέγεθος μνήμης που ανακτάται, επομένως σχεδόν διπλασιάζεται και το άθροισμα των migrations με το zeroing. Αντιθέτως με χρήση του squeezy τα 2 αυτά μεγέθη παραμένουν πάντα μηδενικά. Ενδεικτικά, ο χρόνος ανάκτησης στα 1024MB μειώνεται από 1,6s (στο vanilla virtio-mem) σε μόλις 300ms με τον μηχανισμό squeezy.

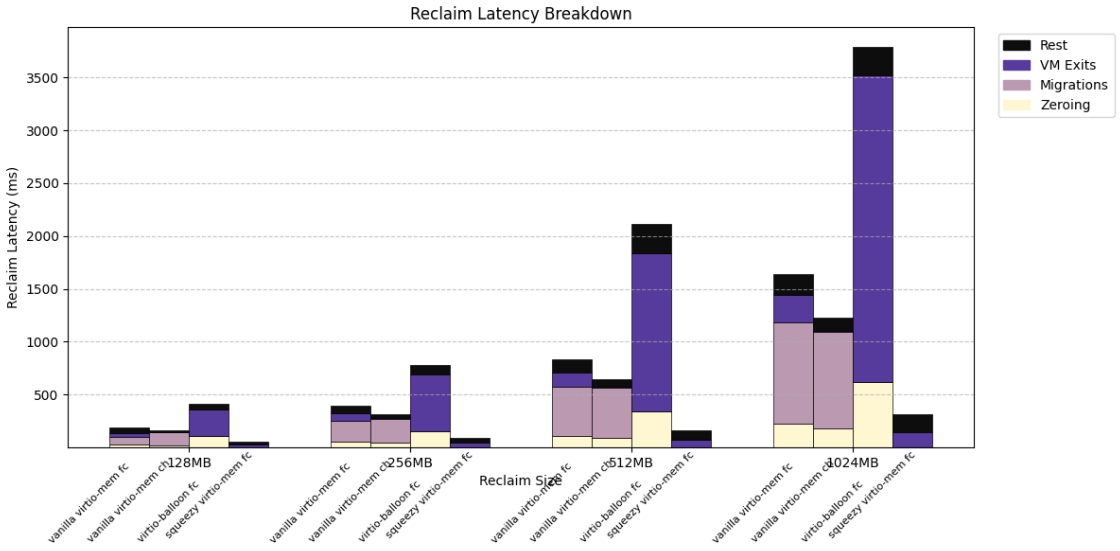
Η υπεροχή του squeezy virtio-mem γίνεται ακόμη πιο εμφανής όταν συγκριθεί με τον μηχανισμό virtio-balloon. Σε αυτή την περίπτωση, το squeezy virtio-mem είναι κατά μέσο όρο 10,3 φορές ταχύτερο, με την ανάκτηση 1024MB να απαιτεί περίπου 3,78s στο virtio-balloon, έναντι μόλις 300ms στο squeezy virtio-mem.

4.2.2 Ανάλυση των συνιστωσών του συνολικού χρόνου

Στο Σχήμα 4.2 παρουσιάζεται μια αναλυτικότερη οπτική των χρόνων ανάκτησης μνήμης για τις διαφορετικές συσκεύες.

Σύγκριση του virtio-mem του Firecracker με αυτό του Cloud Hypervisor σε επίπεδο συνιστωσών

Αρχικά, παρατηρείται ότι η διαφορά στην απόδοση του virtio-mem μεταξύ Firecracker και Cloud Hypervisor οφείλεται κυρίως στο κόστος των VM exits. Το virtio-mem του Firecracker εμφανίζει αισθητά μεγαλύτερες καθυστερήσεις κατά τη διάρκεια των VM exits, τα οποία αποτελούν κατά μέσο όρο το 17,5% του συνολικού χρόνου για ένα δεδομένο μέγεθος μνήμης που ανακτάται, ενώ στο Cloud Hypervisor το αντίστοιχο ποσοστό είναι μόλις 4,2%. Ενδεικτικά, κατά την ανάκτηση 1024MB, το virtio-mem του Firecracker αφιερώνει περίπου 256ms σε VM exits, ενώ στο Cloud Hypervisor ο χρόνος αυτός περιορίζεται σε περίπου 10ms.



Σχήμα 4.2: *Ανάλυση συνιστωσών για τον συνολικό χρόνο ανάκτησης διαφορετικών μεγεθών μνήμης*

Η σημαντική αυτή διαφορά μπορεί να αποδοθεί σε δύο παράγοντες:

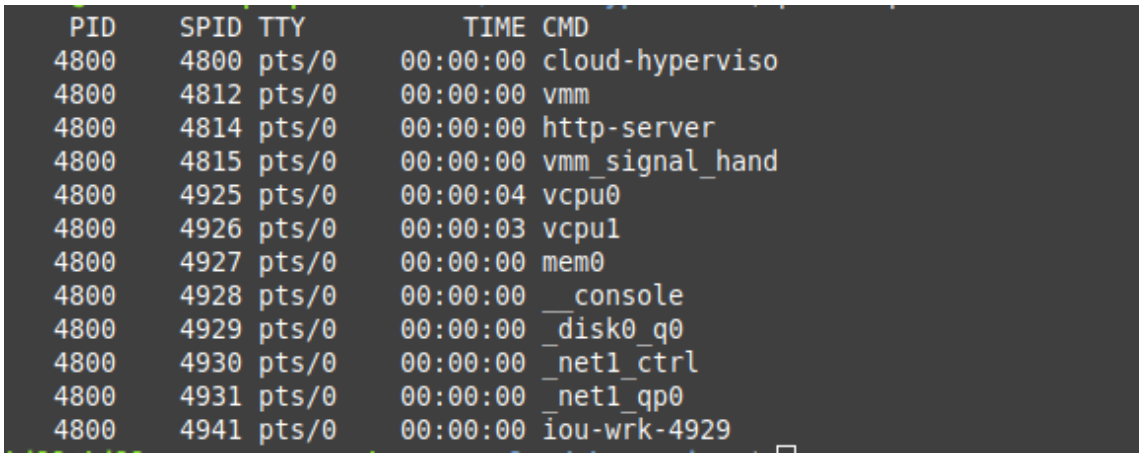
- **Το διαφορετικό μέσο επικοινωνίας (transport) μεταξύ συσκευής και guest**. Το Firecracker χρησιμοποιεί το virtio-MMIO transport, ενώ το Cloud Hypervisor βασίζεται στο virtio-PCI. Το virtio-PCI, προσφέρει ταχύτερη επικοινωνία μεταξύ guest και host. Η διαφορά αυτή οφείλεται κυρίως στον τρόπο χειρισμού των διακοπών (interrupts):

  - Το virtio-PCI χρησιμοποιεί Message Signaled Interrupts (MSI/MSI-X) [34], όπου οι διακοπές αποστέλλονται μέσω εγγραφών στη μνήμη, μειώνοντας τον χρόνο καθυστέρησης.
  - Αντίθετα, το virtio-MMIO χρησιμοποιεί legacy interrupts, τα οποία απαιτούν επιπλέον παγίδευση και επεξεργασία από τον υπερεπόπτη, αυξάνοντας το κόστος ανά I/O λειτουργία.

  Το virtio-MMIO επιλέχθηκε από το Firecracker για λόγους απλότητας και ταχείας εκκίνησης, εις βάρος ωστόσο της απόδοσης σε συχνές λειτουργίες I/O.

- **Η διαφορετική εσωτερική αρχιτεκτονική εκτέλεσης των συσκευών.** Στο Cloud Hypervisor, το virtio-mem εκτελείται σε ξεχωριστό νήμα (thread). Μετά την εκκίνηση μιας εικονικής μηχανής, η εντολή `ps -T -p <PID>` αποκαλύπτει τα ενεργά νήματα της διεργασίας, όπου όπως φαίνεται και απο το σχήμα 4.3 το virtio-mem εκτελείται στο νήμα mem0. Από την ανάλυση του πηγαίου κώδικα, προκύπτει ότι στο συγκεκριμένο νήμα εξυπηρετούνται τα events της συσκευής, δηλαδή και τα VM exits που σχετίζονται με τη συσκευή.

  Αντίθετα, στο Firecracker το virtio-mem μοιράζεται το ίδιο νήμα εκτέλεσης με τις συσκευές virtio-block και virtio-net. Αυτό έχει ως αποτέλεσμα τον ανταγωνισμό για



```
  PID    SPID TTY          TIME CMD
 4800    4800 pts/0    00:00:00 cloud-hyperviso
 4800    4812 pts/0    00:00:00 vmm
 4800    4814 pts/0    00:00:00 http-server
 4800    4815 pts/0    00:00:00 vmm_signal_hand
 4800    4925 pts/0    00:00:04 vcpu0
 4800    4926 pts/0    00:00:03 vcpu1
 4800    4927 pts/0    00:00:00 mem0
 4800    4928 pts/0    00:00:00 __console
 4800    4929 pts/0    00:00:00 _disk0_q0
 4800    4930 pts/0    00:00:00 _net1_ctrl
 4800    4931 pts/0    00:00:00 _net1_qp0
 4800    4941 pts/0    00:00:00 iou-wrk-4929
```

Σχήμα 4.3: *Νήματα που εκτελούνται στον Cloud Hypervisor*

τους πόρους CPU και την καθυστέρηση στην εξυπηρέτηση των VM exits, οδηγώντας σε χαμηλότερη συνολική απόδοση.

Επιβεβαιώνεται ότι τα migrations σελίδων αποτελούν την κύρια πηγή καθυστέρησης του virtio-mem σε guest πυρήνα χωρίς τροποποιήσεις, καθώς αποτελούν κατά μέσο όρο περίπου το 50% του συνολικού χρόνου ανάκτησης μνήμης στο Firecracker και το 73,5% του αντίστοιχου χρόνου στο Cloud Hypervisor.

Επιπλέον, η διαφοροποίηση που παρατηρείται στα αποτελέσματα των migrations για τα μεγέθη 128MB και 256MB θεωρούμε ότι οφείλεται σε στοχαστικούς παράγοντες. Κατά την ανάκτηση μικρότερων μεγεθών μνήμης είναι πιθανό στα sections που θα επιλεχθούν προς unplug, να είναι περισσότερο ή λιγότερο αναμειγμένες σελίδες του πρώτου στιγμιοτύπου memhog (λόγω της τυχαιότητας ανάθεσης σελίδων απο τα Linux) με αποτέλεσμα το κόστος των migrations να διαφέρει απο εκτέλεση σε εκτέλεση.

Παρατηρείται επίσης πως το zeroing αποτελεί κατά μέσο όρο περίπου το 14% του συνολικού χρόνου ανάκτησης στο Firecracker και το 13,9% στο Cloud Hypervisor. Θεωρούμε πως μικρές διαφορές στον χρόνο του zeroing ανάμεσα στο Firecracker και στο Cloud Hypervisor οφείλονται σε τυπικές αποκλίσεις των μετρήσεων.

Ανάλυση των συνιστωσών του virtio-balloon

Επιβεβαιώνεται ότι τα VM exits αποτελούν την κύρια πηγή overhead για το virtio-balloon, καθώς αντιστοιχούν κατά μέσο όρο περίπου στο 69% του συνολικού χρόνου που απαιτείται για την ανάκτηση ενός δεδομένου μεγέθους μνήμης. Ενδεικτικά, κατά την ανάκτηση 1024MB, ο virtio-balloon driver αφιέρωσε περίπου 2,9 δευτερόλεπτα σε VM exits από τα συνολικά 3,8 δευτερόλεπτα που διήρκεσε το αίτημα, γεγονός που αναδεικνύει τον καθοριστικό ρόλο των VM exits στη συνολική καθυστέρηση του μηχανισμού.

Επιπλέον, η διαδικασία του zeroing συνεισφέρει σημαντικά στο συνολικό overhead, αντιπροσωπεύοντας κατά μέσο όρο περίπου 17,6% του συνολικού χρόνου εκτέλεσης.

Ανάλυση των συνιστωσών του squeezy virtio-mem στο Firecracker

Επιβεβαιώνεται ότι με χρήση squeezy πυρήνα στο virtio-mem του Firecracker, οι λειτουργίες page migration και zeroing εξαλείφονται πλήρως. Το μοναδικό latency που παραμένει προέρχεται από τα VM exits και τις υπόλοιπες λειτουργίες, που σχετίζονται με το offlining των blocks μνήμης. Στην περίπτωση αυτή, τα VM exits αντιπροσωπεύουν κατά μέσο όρο περίπου το 48% του συνολικού χρόνου ανάκτησης για ένα δεδομένο μέγεθος μνήμης. Κατά συνέπεια, θα μπορούσε να αναμένεται ακόμη υψηλότερη απόδοση του μηχανισμού εάν χρησιμοποιούνταν το Cloud Hypervisor ως VMM, δεδομένου ότι το VM exit overhead σε αυτό είναι σημαντικά μικρότερο σε σχέση με το Firecracker.

Κεφάλαιο 5

Επίλογος

5.1 Σύνοψη και Συμπεράσματα

Στις FaaS υπηρεσίες όπου ο μικρός κύκλος ζωής των συναρτήσεων οδηγεί στη συχνή δέσμευση και αποδέσμευση μνήμης, η ύπαρξη ενός αποδοτικού μηχανισμού ελαστικής μνήμης αποκτά ιδιαίτερη σημασία. Το Firecracker, ένα VMM που σχεδιάστηκε για την υποστήριξη φόρτων εργασίας που επικεντρώνονται κυρίως σε συναρτήσεις, προσφέρει ελαστική μνήμη αποκλειστικά μέσω του μηχανισμού balloon. Στην ανάλυση που πραγματοποιήσαμε αναδείξαμε πως αυτός ο μηχανισμός εισάγει σημαντικό overhead κατά την ανάκτηση μνήμης, γεγονός που οφείλεται στα πολύ συχνά VM Exits, αφού ο μηχανισμός memory ballooning ανακτά μνήμη σε επίπεδο σελίδων. Το virtio-mem, μια συσκευή ελαστικής μνήμης που βασίζεται στον μηχανισμό memory hot(un)plug, αποτελεί μία αποδοτικότερη εναλλακτική καθώς ανακτά την μνήμη σε επίπεδο block. Σε αυτή την εργασία υλοποιήσαμε το virtio-mem για το Firecracker, και το αξιολογήσαμε συγκριτικά με το virtio-mem του Cloud Hypervisor VMM και το virtio-balloon του Firecracker. Η ανάλυση μας επιβεβαίωσε την υπεροχή του virtio-mem σε απόδοση σε σχέση με το virtio-balloon, ενώ ανέδειξε τους περιορισμούς του, οι οποίοι είναι οι μετακινήσεις και η αρχικοποίηση σελίδων κατά την ανάκτηση μνήμης. Για αυτό το λόγο αξιολογήσαμε την υλοποίηση μας με τη χρήση του Squeezy, ενός τροποποιημένου πυρήνα Linux που εξαλείφει αυτούς τους περιορισμούς. Ακόμα, οι μετρήσεις μας έδειξαν αυξημένο overhead στα VM Exits σε σχέση με την υλοποίηση του virtio-mem στο Cloud Hypervisor, γεγονός που μπορεί να αποδοθεί στην διαφορά του μέσου επικοινωνίας του VMM με τον guest (MMIO στο Firecracker και PCI στο Cloud Hypervisor), καθώς και στην βελτιστοποιημένη υλοποίηση της συσκευής στο Cloud Hypervisor αφού αυτή εκτελείται σε ξεχωριστό thread.

5.2 Μελλοντικές επεκτάσεις

Υπάρχουν αρκετές επεκτάσεις που θα μπορούσαμε να υλοποιήσουμε στο μέλλον, με στόχο τη βελτίωση της αποδοτικότητας της συσκευής.

Αρχικά στην παρούσα έκδοση, η υλοποίηση μπορεί να υποστηρίξει μία μόνο virtio-mem συσκευή. Μια χρήσιμη επέκταση θα ήταν η προσθήκη υποστήριξης περισσότερων virtio-mem συσκευών στο Firecracker. Παράλληλα θα μπορούσε να υλοποιηθεί η δυνατότητα ανάθεσης κάθε συσκευής virtio-mem σε ένα συγκεκριμένο NUMA κόμβο αξιοποιώντας έτσι τις δυνατότητες που παρέχει το virtio-mem.

Όπως έδειξε η αξιολόγηση, το overhead των VM Exits αποτελεί έναν σημαντικό παράγοντα καθυστέρησης κατά τη διαδικασία ανάκτησης μνήμης. Με βάση αυτή την παρατήρηση, μια σημαντική βελτίωση θα ήταν η ενσωμάτωση της υλοποίησης στην πιο πρόσφατη έκδοση του Firecracker, η οποία πλέον υποστηρίζει επικοινωνία με τον guest μέσω virtio-PCI. Ακόμα η υλοποίηση θα μπορούσε να επεκταθεί ώστε να εκτελεί το virtio-mem σε ξεχωριστό thread, όπως συμβαίνει στο Cloud Hypervisor.

Βιβλιογραφία

[1] *Virtualization*. https://en.wikipedia.org/wiki/Virtualization. Ημερομηνία πρόσβασης: 10-8-2025.

[2] Debadatta Mishra και Purushottam Kulkarni. *A survey of memory management techniques in virtualized systems. Computer Science Review*, 29:56–73, 2018.

[3] Orestis Lagkas Nikolos, Chloe Alverti, Stratos Psomadakis, Georgios Goumas και Nectarios Koziris. *Squeezy: Rapid VM Memory Reclamation for Serverless Functions. Proceedings of the 21st European Conference on Computer Systems (EuroSys '26)*, σελίδες 1–16, Edinburgh, Scotland, UK, 2026. ACM.

[4] *What is NUMA*. http://windowstechpro.com/what-is-numa. Ημερομηνία πρόσβασης: 27-10-2025.

[5] *What is cloud computing?* https://aws.amazon.com/what-is-cloud-computing/. Ημερομηνία πρόσβασης: 29-10-2025.

[6] *What is Cloud Elasticity?* https://www.vmware.com/topics/cloud-elasticity. Ημερομηνία πρόσβασης: 29-10-2025.

[7] *Firecracker*. https://firecracker-microvm.github.io/. Ημερομηνία πρόσβασης: 5-10-2025.

[8] *Virtualization*. https://www.ibm.com/think/topics/virtualization. Ημερομηνία πρόσβασης: 10-8-2025.

[9] Kirill Kolyshkin. *Virtualization in linux. White paper, OpenVZ*, 3(39):8, 2006.

[10] Susanta Nanda Tzi cker Chiueh και Stony Brook. *A survey on virtualization technologies. Rpe Report*, 142, 2005.

[11] Diane Barrett και Gregory Kipper. *1 - How Virtualization Happens. Virtualization and Forensics*, σελίδες 3–24. Syngress, Boston, 2010.

[12] Fernando Rodríguez-Haro, Felix Freitag, Leandro Navarro, Efraín Hernánchez-sánchez, Nicandro Farías-Mendoza, Juan Antonio Guerrero-Ibáñez και Apolinar González-Potes. *A summary of virtualization techniques. Procedia Technology*, 3:267–272, 2012.

[13] Rajkumar Buyya, Christian Vecchiola και S. Thamarai Selvi. *Chapter 3 - Virtualization. Mastering Cloud Computing*, σελίδες 71–109. Morgan Kaufmann, Boston, 2013.

[14] James Turnbull. *The Docker Book: Containerization is the new virtualization*. James Turnbull, 2014.

[15] *Hypervisor*. `https://en.wikipedia.org/wiki/Hypervisor`. Ημερομηνία πρόσβασης: 19-8-2025.

[16] Ankita Desai, Rachana Oza, Pratik Sharma και Bhautik Patel. *Hypervisor: A survey on concepts and taxonomy. International Journal of Innovative Technology and Exploring Engineering*, 2(3):222–225, 2013.

[17] *Kernel-based Virtual Machine*. `https://en.wikipedia.org/wiki/Kernel-based_Virtual_Machine`. Ημερομηνία πρόσβασης: 19-8-2025.

[18] Irfan Habib. *Virtualization with KVM. Linux Journal*, 2008(166):8, 2008.

[19] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin και Anthony Liguori. *kvm: the Linux virtual machine monitor. Proceedings of the Linux symposium*, τόμος 1, σελίδες 225–230. Dttawa, Dntorio, Canada, 2007.

[20] *Micro VMs-Qumulus Technology*. `https://www.qumulus.io/micro-vms`. Ημερομηνία πρόσβασης: 5-10-2025.

[21] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka και Diana Maria Popa. *Firecracker: Lightweight virtualization for serverless applications. 17th USENIX symposium on networked systems design and implementation (NSDI 20)*, σελίδες 419–434, 2020.

[22] *Virtual Memory*. `https://en.wikipedia.org/wiki/Virtual_memory`. Ημερομηνία πρόσβασης: 6-10-2025.

[23] Shun Kida, Satoshi Imamura και Kenji Kono. *Revisiting Memory Swapping for Big-Memory Applications. Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region*, HPCASIA '25, σελίδα 33–42, New York, NY, USA, 2025. Association for Computing Machinery.

[24] *Page (computer memory)*. `https://en.wikipedia.org/wiki/Page_(computer_memory)`. Ημερομηνία πρόσβασης: 7-10-2025.

[25] *Page fault*. `https://en.wikipedia.org/wiki/Page_fault`. Ημερομηνία πρόσβασης: 6-10-2025.

[26] David Hildenbrand και Martin Schulz. *Virtio-mem: Paravirtualized memory hot (un) plug. Proceedings of the 17th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, σελίδες 1–14, 2021.

[27] Haikun Liu, Hai Jin, Xiaofei Liao, Wei Deng, Bingsheng He και Cheng zhong Xu. *Hotplug or ballooning: A comparative study on dynamic memory management techniques for virtual machines. IEEE Transactions on parallel and distributed systems*, 26(5):1350–1363, 2014.

[28] *9.10. Balloon Driver | Red Hat Product Documentation*. https://docs.redhat.com/en/documentation/red_hat_virtualization/4.4/html/technical_reference/balloon_driver. Ημερομηνία πρόσβασης: 7-10-2025.

[29] *Memory Hot(Un)Plug — The Linux Kernel documentation*. https://docs.kernel.org/admin-guide/mm/memory-hotplug.html. Ημερομηνία πρόσβασης: 8-10-2025.

[30] *Non-uniform memory access*. https://en.wikipedia.org/wiki/Non-uniform_memory_access. Ημερομηνία πρόσβασης: 27-10-2025.

[31] *NUMA*. https://portal.nutanix.com/page/documents/solutions/details?targetId=BP-2056-MySQL-on-Nutanix:numa.html. Ημερομηνία πρόσβασης: 27-10-2025.

[32] *Virtio on Linux*. https://docs.kernel.org/driver-api/virtio/virtio.html. Ημερομηνία πρόσβασης: 1-11-2025.

[33] *Virtual I/O Device (VIRTIO) Version 1.3*. https://docs.oasis-open.org/virtio/virtio/v1.3/csd01/virtio-v1.3-csd01.pdf. Ημερομηνία πρόσβασης: 8-10-2025.

[34] *Message Signaled Interrupts*. https://en.wikipedia.org/wiki/Message_Signaled_Interrupts. Ημερομηνία πρόσβασης: 29-10-2025.