ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΥΠΟΛΟΓΙΣΤΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

Υλοποίηση μηχανισμού ελαστικής μνήμης utmem σε περιβάλλον unikernel

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

Ιωάννη Αραβαντινού-Σιμωνέτου

Επιβλέπων: Γεώργιος Γκούμας
Αναπληρωτής καθηγητής Ε.Μ.Π.

Αθήνα, Μάρτιος 2021

Εθνικο Μετσοβιο Πολυτεχνειο
ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΥΠΟΛΟΓΙΣΤΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

Υλοποίηση μηχανισμού ελαστικής μνήμης utmem σε περιβάλλον unikernel

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

Ιωάννη Αραβαντινού-Σιμωνέτου

Επιβλέπων: Γεώργιος Γκούμας
Αναπληρωτής Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 11η Μαρτίου 2021.

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

Αθήνα, Μάρτιος 2021

.....

Ιωάννης Αραβαντινός-Σιμωνέτος
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Περίληψη

Με την σταδιακή μετακίνηση της επεξεργασίας των δεδομένων και της εκτέλεσης των εφαρμογών από τοπικά υπολογιστικά συστήματα σε εικονικοποιημένα cloud συστήματα, γίνεται εμφανές πως τα συμβατικά λειτουργικά συστήματα αδυνατούν να επιτελέσουν αποδοτικά τον ρόλο τους. Σχεδιασμένο δεκαετίες πριν, ένα συμβατικό λειτουργικό σύστημα εισάγει περιττές εξαρτήσεις, αυξημένες ανάγκες σε υπολογιστικούς πόρους και χρονικές επιβαρύνσεις κατά την εικονικοποιημένη εκτέλεσή του στο cloud. Η τεχνολογία των unikernel στοχεύει στην εξάλειψη των ανώτερων προβλημάτων, δημιουργώντας μια ελαφριά, εξειδικευμένη και ταχύτατη εικονική μηχανική. Τα περιβάλλοντα unikernel επιτρέπουν την εκτέλεση μιας μοναδικής εφαρμογής και ταιριάζουν καλύτερα στο cloud περιβάλλον. Ταυτόχρονα, η εικονικοποίηση των υπολογιστικών συστημάτων οδηγεί αναπόφευχτα σε μη βέλτιστη χρησιμοποίηση των διαθέσιμων πόρων, και ειδικά της μνήμης. Ο μηχανισμός διαχείρισης μνήμης utmem εκμεταλλεύεται την συνεργασία μεταξύ του εικονικοποιημένου συστήματος και του επόπτη, εφαρμόζοντας τεχνική παραεικονικοποίησης, με σκοπό την αυξημένη απόδοση χρήσης της μνήμης από τις εικονικοποιημένες εφαρμογές. Σκοπός της παρούσας εργασίας είναι η μελέτη των χαρακτηριστικών και της φιλοσοφίας διάφορων unikernel περιβάλλοντων, καθώς και του μηχανισμού της utmem με τελικό στόχο την ενσωμάτωση του τελευταίου σε κάποιο unikernel πλαίσιο. Τέλος, αποτιμάται πειραματικώς η χρηστική αξία του εν λόγω συνδυασμού των δύο αυτών καινοτόμων τεχνολογιών, ως προς συμβατικά περιβάλλοντα εικονικοποίησης και περιβάλλοντα lightweight εικονικοποίησης.

Λέξεις-κλειδία: εικονικοποίηση, υπολογιστικό νέφος, λειτουργικά συστήματα, unikernel, Rumprun, transcedent memory, utmem

Abstract

During the last decades, the data processing and application execution have moved from local computer system to virtualized cloud systems. This transition made apparent that conventional operating systems are unable to fulfill their role effectively. A traditional operating system is characterized by unnecessary software dependencies, increased needs in computational resources and temporal delays when executed on the cloud as a guest. The technology of unikernel manages to eliminate the aforementioned problems, by creating a lightweight, specialized and fast virtual machine. This kind of machines supports execution of a unique application and fits better on the cloud environment. Meanwhile, the virtualization of a computer system inevitably leads to suboptimal usage of the available resources, especially memory. Utmem is a mechanism of memory management that takes advantage of the cooperation between guest and hypervisor, being a paravirtualization technique. Its purpose is to elevate the efficiency of memory used by virtualized applications. This thesis studies the philosophy and the characteristics of various unikernel frameworks, as well as the mechanism of utmem, with ultimate cause the incorporation of the latter to a unikernel environment. Finally, the worth of that combination of technologies is evaluated, by making comparisons inside both a tradition Linux environment and a lightweight unikernel environment.

Key words: virtualization, cloud computing, operating systems, unikernel, Rumprun, transcedent memory, utmem

Ευχαριστίες

Ευχαριστώ θερμά τον επιβλέποντα καθηγητή κύριο Γεώργιο Γκούμα για την ευκαιρία που μου έδωσε να εκπονήσω την παρούσα εργασία. Θέλω, επίσης, να ευχαριστήσω τα παιδιά από το Εργαστήριο Υπολογιστικών Συστημάτων της σχολής μας για τη βοήθεια, τις συμβουλές και την συνεργασία τους για όσο διάστημα διήρκησε αυτό το ταξίδι. Ευχαριστώ, λοιπόν, τους υποψήφιους διδάκτορες Στράτο Ψωμαδάκη, Κωνσταντίνο Παπαζαφειρόπουλο και Ορέστη Λάγκα Νικολό.

Τέλος δεν θα μπορούσα να παραλείψω από τις ευχαριστίες του κοντινούς μου ανθρώπους. Ένα μεγάλο ευχαριστώ, λοιπόν, πηγαίνει στην οικογενειά μου που βρίσκεται πάντα δίπλα μου. Ένα μεγάλο ευχαριστώ και στους φίλους μου για όλες τις στιγμές που μοιραστήκαμε μαζί αυτά τα χρόνια. Κύριως, όμως, ευχαριστώ από την καρδιά μου την Ελένη για όλα αυτά τα ανεκτίμητα που μου έχει προσφέρει.

Ιωάννης Αραβαντινός-Σιμωνέτος
Αθήνα, 11 Μαρτίου 2021

Περιεχόμενα

Κατάλογος Σχημάτων

Κατάλογος Πινάκων

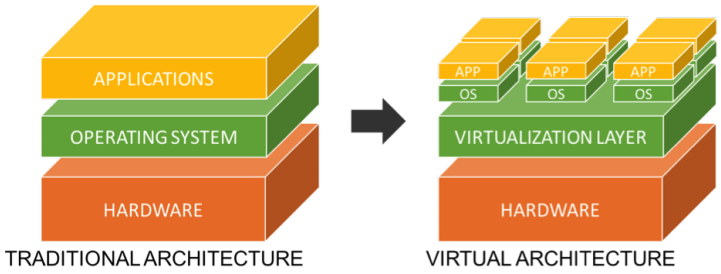
Κεφάλαιο 1

Εισαγωγή

Με την αυγή της τρίτης δεκαετίας του εικοστού πρώτου αιώνα, έχει καταστεί σαφές πως οι υπηρεσίες του cloud computing αποτελούν ένα από τα ισχυρότερα εργαλεία της σύγχρονης τεχνολογίας των υπολογιστών. Ως εκ τούτου, αποτελεί αντικείμενο σχολαστικής μελέτης από πανεπιστημιακά ιδρύματα και ερευνητικά κέντρα παγκοσμίως. Ο όρος cloud computing αναφέρεται σε ένα αφηρημένο μοντέλο υπολογιστικών δομών ανεξάρτητό από το φύσικό υλικό (hardware) όπου γίνονται οι υπολογισμοί, σε αντίθεση με τα παραδοσιακά IT infrastructure όπου το υλικό είναι άμεσα προσβάσιμο και αλληλένδετο με την εκάστοτε επεξεργαστική δραστηριότητα. Το μεγάλο πλεονέκτημα του cloud computing, είναι η on-demand πρόσβαση σε υπηρεσίες ή και υπολογιστικούς πόρους χωρίς την άμεση εμπλοκή του χρήστη, από απομακρυσμένο περιβάλλον[1].

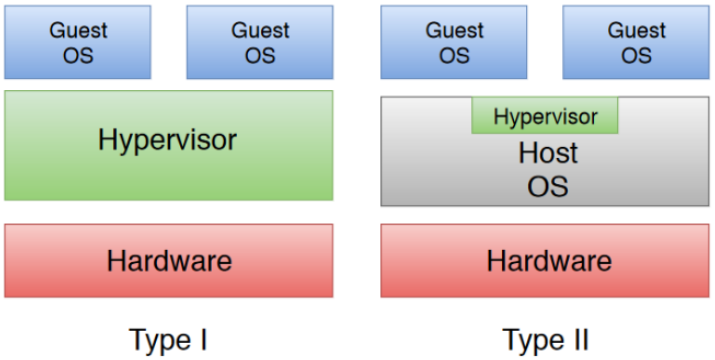
Το cloud computing στηρίζεται σε μεγάλο βαθμό στην εικονικοποίηση (virtualization). Με τον όρο εικονικοποίηση περιγράφεται η διαδικασία εκτέλεσης ενός εικονικού (σε αντιδιαστολή με το πραγματικό) στιγμιοτύπου ενός υπολογιστικού συστήματος, σε αφαίρεση από το πραγματικό υλικό επάνω στο οποίο γίνεται η εκτέλεση. Υπάρχουν διάφορα είδη εικονικοποίησης. Ένα αρκετά γνωστό στον μέσο προγραμματιστή είδος είναι η εικονικοποίηση σε επίπεδο εφαρμογών, όταν η γλώσσα προγραμματισμού επιβάλει την εκτέλεση της εφαρμογής εντός εικονικού περιβάλλοντος, για παράδειγμα η java με το JVM[12]. Ένα στιγμιότυπο εικονικοποίησης και προσομοίωσης ενός υπολογιστικού συστήματος ονομάζεται εικονική μηχανή (virtual machine ή VM). Η παρούσα διπλωματική εργασία ασχολείται με την εικονικοποίηση σε επίπεδο συστήματος, όπου το σύνολο του υλικού του συστήματος εικονικοποιείται με την βοήθεια κάποιου επόπτη (hypervisor). Τέτοιοι επόπτες είναι συνήθως το KVM, το Xen κ.α.

Ο επόπτης είναι ένα σύνολο λογισμικού, το οποίο επιτρέπει την εικονικοποίηση ενός συστήματος. Οι εποπτές μπορεί να είναι διεργασίες-μέρη ενός άλλου λειτουργικού συστήματος (host) ή να παρεμβάλλονται απευθείας ανάμεσα στο υλικό και το εικονικοποιημένο σύστημα (guest). Στην πρώτη περίπτωση ονομάζονται επόπτες τύπου II,



Σχήμα 1.1: Παραδοσιακό σύστημα και Εικονικοποιημένο σύστημα

ενώ στην δεύτερη επόπτες τύπου I[19].



Σχήμα 1.2: Επόπτες τύπου I και τύπου II

Ταυτόχρονα με την εικονικοποίηση, χρησιμοποιείται και η τεχνική της προσομοίωσης (emulation). Με την προσομοίωση, μπορεί να αναπαραχθεί πιστά η συμπεριφορά ενός κομματιού υλικού, το οποίο δεν έχει κάποιο φυσικό αντίστοιχο στο υφιστάμενο πραγματικό σύστημα. Τέτοια κομμάτια υλικού μπορεί να είναι, για παράδειγμα, μια κάρτα δικτύου. Η αναπαραγωγή της συμπεριφοράς γίνεται εξ ολοκλήρου σε επίπεδο λογισμικού (software). Ένα ευρέως διαδεδομένο πρόγραμμα προσομοίωσης είναι το QEMU (Quick EMUlator), το οποίο μπορεί να λειτουργήσει και ως επόπτης, ενώ ο συνδυασμός emulator-hypervisor QEMU/KVM είναι μια από τις αποτελεσματικότερες λύσεις για εικονικοποίηση ενός λειτουργικού συστήματος[8].

1.1 Είδη εικονικοποίησης

Με την πλήρη εικονικοποίηση, προσομοιώνεται η λειτουργία ενός πλήρους λειτουργικού συστήματος. Μια σημαντική ιδιότητα είναι πως δεν υπάρχει υποβοήθηση από το υλικό (hardware assisted), όπως εμφανίζεται σε άλλη περίπτωση στην συνέχεια.

Το λειτουργικό σύστημα που προσομοιώνεται (guest) δεν χρειάζεται να υποστεί καμία αλλαγή και προσφέρεται ως έχει. Ο hypervisor προσφέρει οτιδήποτε χρειάζεται ο guest σε επίπεδο υλικού, έτσι ώστε ο guest να αισθάνεται πως εκτελείται επάνω σε πραγματικό υλικό, όπως είναι σχεδιασμένος να κάνει. Ταυτόχρονα, προσομοιώνει και την εκτέλεση των εντολών του επεξεργαστή. Ειδική μνεία χρειάζεται για τις προνομιούχες εντολές του guest, οι οποίες δεν επιτρέπεται για λόγους ασφάλειας να εκτελεστούν απευθείας επάνω στην CPU, και για αυτό τον λόγο εφαρμόζονται διαφορετικές λύσεις όπως η trap and emulate. Μάλιστα, ειδικά για την x86 αρχιτεκτονική, η δυσκολία στο να προσομοιωθούν αυτές οι συγκεκριμένες εντολές έκαναν αρχικά την εικονικοποίηση να μοιάζει αδύνατη [13].

Η προηγούμενη τεχνική έχει το μειονέκτημα της μειωμένης ταχύτητας εκτέλεσης. Για τον λόγο αυτό έχει αναπτυχθεί η εικονικοποίηση με υποβοήθηση υλικού (hardware assisted virtualization), όπου έχουν προστεθεί εντολές στην CPU, οι οποίες βοηθούν την εκτέλεση των ευαίσθητων εντολών του guest, δίχως μείωση στην ταχύτητα. Και σε αυτήν την περίπτωση ο guest εκτελείται ως έχει, χωρίς κάποια τροποποίηση. Συνεπώς, υπάρχει πρόβλεψη από το υλικό ώστε να επιταχύνει την εκτελεσή του guest με ασφάλεια[13]. Αυτή η τεχνική είναι διαθέσιμη εφόσον υπάρχουν επεκτάσεις υλικού (virtualization extensions), όπως η τεχνολογία VT-x της Intel ή η AMD-V της AMD στους επεξεργαστές της εκάστοτε εταιρείας[17].

Επιπροσθέτως, η άγνοια του guest, ως προς την εκτέλεση του επάνω σε εικονικό σύστημα, μπορεί μεν να είναι μια πολύ επιθυμητή αφαίρεση, οδηγεί όμως συχνά σε μη βέλτιστη συμπεριφορά και χρήση των πόρων του πραγματικού συστήματος. Για παράδειγμα η διαχείριση της υφιστάμενης μνήμης RAM, δεν γίνεται με βέλτιστο τρόπο από μεριάς guest, καθώς αυτός δεν γνωρίζει πραγματικά πόση συνολικά διαθέσιμη υπάρχει στο σύστημα, παρά μόνο τόση όση έχει ανατεθεί από τον host στον guest. Με την παρα-εικονικοποίηση (paravirtualization), ο guest γνωρίζει πως δεν εκτελείται επάνω σε φυσικό σύστημα, και ζητά την συνεργασία του επόπτη του για την αποδοτικότερη αντιμετώπιση συγκεκριμένων σεναρίων χρήσης του. Επί παραδείγματι, ο guest μπορεί να μεταβάλει την συμπεριφορά του και να επιλέξει να μην εκτελέσει τις προαναφερθείσες ευαίσθητες εντολές, και στην θέση αυτών να ζητήσει την συνεργασία του host, ώστε να αναλάβει αυτός τις συγκεκριμένες ενέργειες[13].

1.2 Προβλήματα και λύσεις

1.2.1 Unikernels

Οι περισσότερες υπηρεσίες στο cloud computing μπορούν εύκολα να υλοποιηθούν ως απλές εφαρμογές ενός παραδοσιακού λειτουργικού συστήματος. Το workload αυτών είναι χαμηλών η μέτριων απαιτήσεων και εστιασμένο στην εκάστοτε απαίτηση του χρήστη-πελάτη. Όμως η δημιουργία μια εικονικής μηχανής, εντός της οποίας εκτελείται ένα συμβατικό λειτουργικό σύστημα, είναι κάθε άλλο παρά τέλεια λύση.

Τα συμβατικά λειτουργικά είναι σχεδιασμένα με γνώμονα την παράλληλη εκτέλεση πολλαπλών εφαρμογών, είναι υπερβολικά σύνθετα, και πολύ αργά κατά την εκκίνηση για τις απαιτήσεις των στιγμιαίων υπηρεσιών του cloud. Το κυριότερο μειονέκτημα όμως, είναι πως η πολυπλοκότητα του συστήματος αποτελεί αχίλλειο πτέρνα για την ασφάλεια και την σταθερότητα ολόκληρης της εικονικής μηχανής.

Λύση στο ανώτερο πρόβλημα προσφέρουν τα unikernels, δηλαδή μικρές εικονικές μηχανές ικανές να εξουδετερώσουν τις προαναφερθείσες αδυναμίες. Εστιασμένα στην εκτέλεση μιας συγκεκριμένης υπηρεσίας έχουν ελάχιστες απαιτήσεις σε υλικό, ενώ διατηρούν τάξεις μεγέθους μικρότερο μέγεθος από τα παραδοσιακά λειτουργικά. Επιπροσθέτως, η μείωση της πολυπλοκότητας και του μεγέθους αυτών, μειώνει την πιθανότητα δυσλειτουργίας καθώς και την επιφάνεια επίθεσης (attack surface) από κακόβουλους τρίτους χρήστες[16]. Υπάρχουν διάφορα unikernel frameworks, τα οποία αναφέροναι σε επόμενο κεφάλαιο, ενώ οι εικονικές μηχανές που προέρχονται από αυτά συνήθως εκτελούνται με την βοήθεια κάποιου επόπτη ή πιο σπάνια απευθείας επάνω στο υλικό (bare metal).

1.2.2 Transcedent memory

Ένα δεύτερο πρόβλημα που αναφέρθηκε είναι η υποχρησιμοποίηση των πόρων του συστήματος στην περίπτωση του virtualization, και ειδικά της μνήμης. Όταν εξαντληθεί η διαθέσιμη μνήμη, ένα παραδοσιακό λειτουργικό καταφεύγει στο να μεταφέρει (swapping) σελίδες μνήμης (memory pages), στον δίσκο ή σε κάποιο αντίστοιχο αποθηκευτικό μέσο, η ταχύτητα των οποίων είναι πολύ μικρότερη σε σχέση με αυτήν της φυσικής μνήμης. Κατά την εικονικοποίηση ενός VM το πιθανότερο όμως είναι να υπάρχει διαθέσιμη φυσική μνήμη, απλά όμως να ανήκει στον host, και να μην έχει ανατεθεί εξ αρχής στον guest. Άρα ένος μέρος της φυσικής μνήμης παραμένει ανεκμετάλευτο, γεγονός που θα μπορούσε να έχει αποφευχθεί εάν η μνήμη κατανέμονταν με αποδοτικότερους τρόπους.

Για να αντιμετωπιστεί αυτό το φαινόμενο έχουν εφαρμοστεί διάφορες λύσεις, κάθε μια με διαφορετικά πλεονεκτήματα και μειονεκτήματα. Μια ισορροπημένη τεχνική στηρίζεται στην εκμετάλλευση της επικοινωνίας host και guest, δηλαδή εφαρμογή τεχνικών paravirtualization, ώστε να μεταβάλλεται δυναμική η διαθέσιμη μνήμη του guest. Για παράδειγμα, ο μηχανισμός του ballooning αυξομειώνει κατά το runtime τον αριθμό σελίδων μνήμης που ανήκουν στον guest, σύμφωνα με τις ανάγκες του.

Η μηχανισμός διαχείρισης ελαστικής μνήμης transcendent memory ή tmem, ο οποίος αναπτύχθηκε από την Oracle το 2009, επεκτείνει την προηγούμενη σκέψη συνεργασίας guest-host ένα βήμα παραπέρα. Ο guest σε περίπτωση έλλειψης μνήμης δεν ζητά να του παραχωρηθεί επιπλέον από τον host. Αντίθετα ο guest ζητά από τον host να αναλάβει την αποθήκευση των δεδομένων της μνήμης του πρώτου, ωστέ στη συνέχεια ο guest να μπορεί να αποδεσμεύσει μέρος της μνήμης του γνωρίζοντας πως

ανά πάσα στιγμή μπορεί να την ανακτήσει από τον host. Η αναφορά σε αυτές τις περιοχές μνήμης γίνεται με ζεύγη κλειδιού-τιμής (key-value), ώστε host και guest να έχουν ένα κοινό κώδικα-πρωτόκολλο επικοινωνίας[27][19].

Στην tmem, η επικοινωνία γίνεται μεταξύ του πυρήνα (kernel space) του host και του guest. Σε προηγούμενη εργασία, όμως, αναπτύχθηκε μηχανισμός που επιτρέπει την πρόσβαση στην tmem από το χώρο χρήστη (user space) του guest, και άρα από τις εκτελούμενες εφαρμογές. Ο νέος μηχανισμός ονομάστηκε userspace transcendent memory ή utmem. Αποδείχθηκε δε, πως σε περιπτώσεις έλλειψης μνήμης μια εφαρμογή που χρησιμοποιεί αυτόν τον μηχανισμό παρουσιάζει καλύτερη συμπεριφορά ως προς την ταχύτητα και την διαχείριση της μνήμης[29].

1.3 Σκοπός της εργασίας

Ο σκοπός αυτής της διπλωματικής εργασίας είναι, κατ' αρχάς, η μελέτη των unikernel frameworks και του μηχανισμού utmem. Στην συνέχεια παρουσιάζουμε την ενσωμάτωση του μηχανισμού της utmem σε ένα συγκεκριμένο unikernel framework. Τέλος, παραθέτουμε την πειραματική αποτίμηση της νέας διαθέσιμης τεχνολογίας σε σύγκριση με τα υπάρχοντα δεδομένα, και σε σύγκριση με τον αρχικό μηχανισμό.

Με την παρούσα εργασία, αναπτύσσουμε ένα αξιόλογο εργαλείο για τον προγραμματιστή unikernel εφαρμογών, όσον αφορά την ρητή διαχείριση της μνήμης των εφαρμογών του. Ταυτόχρονα, όπως φαίνεται και στην συνέχεια, βελτιώνουμε την συμπεριφορά του unikernel VM σε σενάρια έλλειψης διαθέσιμης μνήμης.

Κεφάλαιο 2

Θεωρητικό Υπόβαθρο

2.1 Unikernels - Rumprun

2.1.1 Ορισμός

Ο όρος unikernel αναφέρεται σε μικρές, εξειδικευμένες εικονικές μηχανές, χωρίς διαχωρισμό user space και kernel space, οι οποίες κατασκευάζονται χρησιμοποιώντας κάποιο library operating system[16].

Αυτές οι εικονικές μηχανές είναι μικρές σε μέγεθος, καθώς συνήθως έχουν αφαιρεθεί πολλαπλά στρώματα-μέρη λογισμικού που υπάρχουν σε ένα παραδοσιακό λειτουργικό σύστημα. Εξειδικευμένες, επειδή εστιάζουν στην εκτέλεση μιας και μόνο λειτουργίας, και δεν προσφέρουν δυνατότητες παράλληλης εκτέλεσης πολλαπλών εφαρμογών όπως τα παραδοσιακά λειτουργικά συστήματα. Το πιο σημαντικό, όμως, χαρακτηριστικό είναι η ανάπτυξη τους χρησιμοποιώντας κάποιο library operating system, το οποίο και αξίζει να αναλυθεί στην συνέχεια.

2.1.2 Library Operating Systems

Τα library operating systems αποτελούν μια μορφή αρχιτεκτονικής όπου οι διάφορες υπηρεσίες που χρειάζεται ένα high level application, για παράδειγμα η ανταλλαγή πακέτων με χρήση κάποιου network protocol, προσφέρονται ως συναρτήσεις βιβλιοθήκης από το περιβάλλον στο οποίο αναπτύσσεται, οι οποίες ενσωματώνονται τελικά σε ένα μοναδικό επίπεδο λογισμικού. Για να μπορέσει να επιτευχθεί αυτό, τα library operating systems από σχεδιασμού τους προσφέρουν δύο πράγματα. Πρώτον τις βιβλιοθήκες που προσφέρουν πρόσβαση στο υλικό και στους πόρους, ουσιαστικά έναν κατάλληλο μηχανισμό. Δεύτερον, τις κατάλληλες πολιτικές με τις οποίες επιτυγχάνεται ορθός έλεγχος της πρόσβασης, και απομόνωση στο υψηλό επίπεδο της εφαρμογής. Ο έλεγχος, συνεπώς, και η προστασία του υλικού δεν εξασφαλίζεται πλέον μεταξύ του χώρου της εφαρμογής και του χώρου του πυρήνα του λειτουργικού, αλλά ακόμα

χαμηλότερα, στο επίπεδο του υφιστάμενου υλικού[21].

Οι πρώτες υλοποιήσεις τέτοιων συστημάτων-αρχιτεκτονικών, εμφανίστηκαν στα τέλη της δεκαετίας του 1990, όπως το Exokernel και το Nemesis[21]. Ένα εμφανές πλεονέκτημα αυτών των λιγότερων σύνθετων αρχιτεκτονικών, είναι η ταχύτερη πρόσβαση στους πόρους, καθώς δεν χρειάζεται η εναλλαγή μεταξύ privilege mode και μη. Κυρίως, όμως, η απλούστευση σε επίπεδου στοίβας λογισμικού προσφέρει προβλέψιμη συμπεριφορά όλου του συστήματος, και οδηγεί σε σταθερότερες και ασφαλέστερες εφαρμογές.

Κληρονομώντας, συνεπώς, αυτήν την φιλοσοφία τα unikernels στηρίζονται επάνω σε αυτή την απλούστευση της στοίβας του λογισμικού. Το ερώτημα, λοιπόν, που οδήγησε στην δημιουργία των unikernel, είναι το εξής: «τι θα γινόταν αν ολόκληρη η στοίβα του λογισμικού, από το ανώτερο επίπεδο, μέχρι και τον κώδικα assembly, μεταγλωτίζονταν ως ένα σώμα σε ένα ασφαλές, υψηλού επιπέδου γλωσσας framework;»

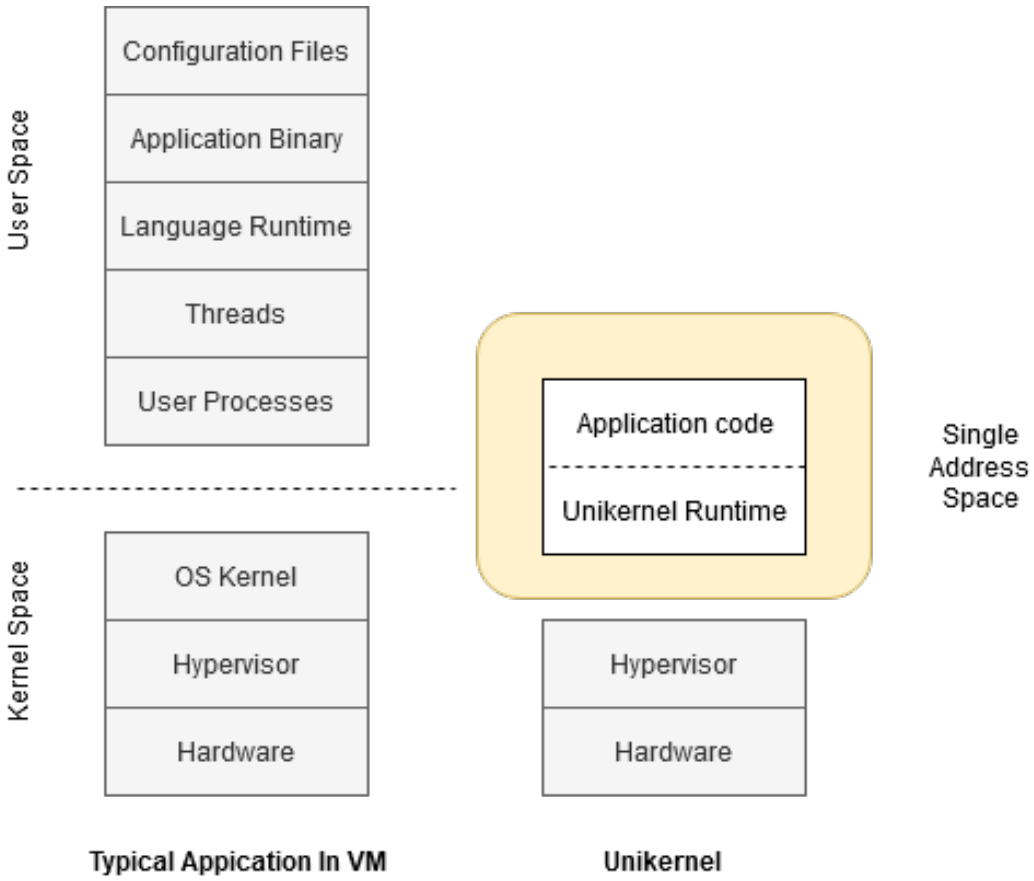
2.1.3 Χαρακτηριστικά

Για να γίνει καλύτερα κατανοητή η διαφορά και το πλεονέκτημα ενός unikernel σε σχέση με ένα συμβατικό λειτουργικό σύστημα, θα συγκριθεί στην συνέχεια η στοίβα λογισμικού σε δύο διαφορετικές περιπτώσεις.

Σε ένα παραδοσιακό λειτουργικό σύστημα όπως στο linux, ο προγραμματιστής αναπτύσσει τον κώδικα της εφαρμογής του στο υψηλότερο επίπεδο. Ο κώδικας αυτός εξαρτάται εν πολλοίς από διάφορες βιβλιοθήκες που προσφέρει το σύστημα. Αυτές οι βιβλιοθήκες ενώνονται δυναμικά ή στατικά με τον κώδικα του χρήστη και παράγεται το εκτελέσιμο της εφαρμογής. Τώρα, κατά την εκτέλεση η εφαρμογή επικοινωνεί με τον πυρήνα του λειτουργικού συστήματος ώστε αυτός να εκτελέσει με την σειρά του προνομιούχες εντολές-διαδικασίες οι οποίες σχετίζονται με λειτουργίες οι οποίες σε ένα multi-process περιβάλλον χρήζουν ασφαλείας και προσοχής. Χαρακτηριστικό παράδειγμα είναι ή πρόσβαση στο υλικό, όπως π.χ. μια ανάγνωση από αρχεία του δίσκου. Ο μηχανισμός που διεκπεραιώνει αυτήν την επικοινωνίας είναι οι κλήσεις συστήματος (system calls), σύμφωνα με τις οποίες η εκτελούμενη διεργασία υποχρεούται να αιτηθεί από τον πυρήνα (kernel) του λειτουργικού να αναλάβει αυτός την ενέργεια εκ μέρους της. Υποχρεωτικά εμφανίζεται αυτός ο διαχωρισμός μεταξύ του χώρου χρήστη και του χώρου πυρήνα, και για κάθε προνομιούχα δραστηριότητα πρέπει να γίνεται εναλλαγή (context switch) μεταξύ αυτών των δύο. Ακόμα πιο κάτω, υπάρχει ο κώδικας που τρέχει στον χώρο πυρήνα, από το σύστημα αρχείων, τις εικονικές συσκευές,τους drivers που επικοινωνούν με το πραγματικό υλικό έως και την διαχείριση μνήμης και την χρονοδρομολόγηση των διεργασιών.

Από την άλλη, σε ένα unikernel περιβάλλον εκτελείται μονάχα μία εφαρμογή χωρίς την πιθανή συνεκτέλεση τρίτων εφαρμογών. Για αυτό τον λόγο, απουσιάζει ο διαχωρι-

σμός ανάμεσα σε user space και kernel space, καθώς δεν υπάρχει ανάγκη προστασίας και ασφάλειας μεταξύ διεργασιών. Αυτό οδηγεί, σε ένα μοναδικό χώρο όπου ο κώδικας από το πιο υψηλό επίπεδο μέχρι και το πιο χαμηλό εκτελείται σε μοναδικό context και χώρο διευθύνσεων μνήμης. Ταυτόχρονα, κατά την μεταγλώττιση της εφαρμογής, το toolchain, δηλαδή τα εργαλεία δημιουργίας του unikernel, του εκάστοτε unikernel framework, αφαιρεί όλα τα περιττά συστατικά του συστήματος, ώστε να απομένουν τα απολύτως απαραίτητα που χρειάζονται για τον συγκεκριμένο σκοπό. Απουσιάζει, επίσης, το σύστημα που διαχειρίζεται την παράλληλη εκτέλεση των διεργασιών και συνήθως η εικονική μνήμη[22].



Σχήμα 2.1: Συμβατική εικονική μηχανή - Unikernel

Δημιουργείται, συνεπώς, μια ταχύτατη, μικρή και εστιασμένη εικονική μηχανή, η οποία μπορεί να εκτελεστεί αυτόνομα επάνω σε ένα hypervisor ή στο ίδιο το υλικό, δίχως την ανάγκη ενός υφιστάμενου λειτουργικού συστήματος. Η εκκίνηση (boot) είναι ταχύτατη, τάξεις μεγέθους ανώτερη από ένα συμβατικό λειτουργικό στο οποίο εκτελείται αντίστοιχη εφαρμογή. Ταυτόχρονα, η ασφάλεια είναι αυξημένη καθώς η εφαρμογή τρέχει στο δικό της περιβάλλον, αποκομμένη από την συνεκτέλεση άλλων

στο ίδιο χώρο χρήστη, όπως ισχύει στα συμβατικά λειτουργικά συστήματα. Τέλος, οι ανάγκες σε πόρους ελαχιστοποιούνται, καθώς μπορεί να ανατεθούν ακριβώς όσους χρειάζεται η συγκεκριμένη εφαρμογή, ενώ σε ένα συμβατικό λειτουργικό θα έπρεπε να μεριμνήσει ο σχεδιαστής ώστε να υπάρχουν διαθέσιμοι πόροι για όλες τις εφαρμογές που μπορεί να εκτελεστούν παράλληλα, ακόμα και αν δεν εκτελούνται πράγματι.

2.1.4 Frameworks

Ακολουθεί μια χρήσιμη αναφορά σε διάφορα unikernel frameworks που υπάρχουν την στιγμή που γράφονται αυτές οι γραμμές, καθώς και μια ανάλυση των χαρακτηριστικών του καθενός.

MirageOs

Το MirageOs αποτελεί ένα από τα παλαιότερα και πιο διαδεδομένα frameworks. Το MirageOs δημιουργήθηκε στο εργαστήριο υπολογιστών του πανεπιστημίου του Cambridge. Ο σκοπός των δημιουργών ήταν να λύσουν το πρόβλημα που προκύπτει κατά την εικονικοποίηση συστημάτων, πως προστίθεται ένα επιπλέον στρώμα λογισμικού, οδηγώντας σε μη βέλτιστη συμπεριφορά των εικονικών μηχανών[21]. Μάλιστα στο ίδιο εργαστήριο είχε αναπτυχθεί το 2003 ο hypervisor Xen[28]. Ο σκοπός του project, ήταν να ανακατασκευαστούν οι εικονικές μηχανές, από το επίπεδο της εφαρμογής μέχρι και το επίπεδο του πυρήνα ώστε να είναι καλύτερα αλληλοσυνδεδεμένα στην βάση ενός library operating system.

Για την δημιουργία του unikernel με βάση το MirageOS χρησιμοποιείται η γλώσσα OCaml, μια γλώσσα αρκετά υψηλού επιπέδου. Η επιλογή της συγκεκριμένης γλώσσας έχει γίνει διότι αυτή επιτρέπει την ανάπτυξη type safe εφαρμογών με μέλημα την προστασία της μνήμης από memory leaks. Τέτοια σφάλματα σχετίζονται με την μη ορθή χρήση των διαφόρων τύπων δεδομένων από μια γλώσσα. Η OCaml για να το πετύχει αυτό διαθέτει στατικό έλεγχο των τύπων (static type check), οπότε οι τυχόν ασυμφωνίες εντοπίζονται κατά το compile και όχι κατά την εκτέλεση, καθώς και ένα διακριτικό συλλέκτη σκουπιδιών (garbage collector), ο οποίος ανακυκλώνει την αποδεσμευμένη μνήμη.

Για να αξιοποιηθούν καλύτερα τα νέα χαρακτηριστικά του mirageOs, αναπτύχθηκαν από την αρχή σημαντικά συστατικά ενός παραδοσιακού λειτουργικού συστήματος, όπως το network και storage stack. Έτσι, γράφτηκαν βιβλιοθήκες για TCP/IP, HTTP, DNS κ.α. στην Ocaml. Το αποτέλεσμα είναι ένα ασφαλές και αρθρωτό framework, ιδανικό να υποστηρίξει web υπηρεσίες. Υπάρχουν διάφορα παραδείγματα self-hosted διαδικτυακών ιστοσελίδων που εκτελούνται ως MirageOs εφαρμογές επάνω σε κάποιον hypervisor, όπως το Xen[7].

Ένα μειονέκτημα είναι πως η εκτελούμενη εφαρμογή πρέπει να είναι γραμμένη σε OCaml για να μπορεί να εκτελεστεί με το MirageOs. Αυτό συνήθως δεν ισχύει για

τα περισσότερα προγράμματα, με αποτέλεσμα να απαιτείται ανάπτυξη της εφαρμογής από την αρχή. Τέλος, οι υποστηριζόμενες πλατφόρμες είναι το Xen, και το KVM αν χρησιμοποιηθεί το solo5 framework, που θα αναλυθεί στην συνέχεια.

IncludeOS

Το IncludeOS είναι ένα μινιμαλιστικό, ανοιχτού κώδικα, unikernel framework, το οποίο στοχεύει σε εφαρμογές και υπηρεσίες γραμμένες σε γλώσσα C++, και που απευθύνονται στο cloud. Πρόκειται για ένα από τα νεότερα unikernel frameworks, και στοχεύει στην ανάπτυξη τέτοιων εικονικών μηχανών με την ευκολία που έχει η συμπερίληψη μια βιβλιοθήκης κατά την ανάπτυξη των εφαρμογών. Γράφοντας απλά #include<os>, κυριολεκτικά κατά το linking της εφαρμογής μας θα συμπεριληφθεί ένα μικροσκοπικό λειτουργικό σύστημα και θα παραχθεί μια πλήρης εικόνα εικονικής μηχανής[2].

Σε αντίθεση με αρκετά άλλα frameworks, το IncludeOs δεν έχει κληρονομήσει από κάποιο άλλη πλατφόρμα αυτούσιες τις βιβλιοθήκες ή τα συστατικά στοιχεία που χρειάζονται οι εφαρμογές. Αντίθετα όλα αυτά έχουν γραφτεί από την αρχή σε C++, με έμφαση στα unikernel χαρακτηριστικά του. Με αυτόν τον τρόπο έχει επιτευχθεί αποδοτική διαχείριση των πόρων και των απαιτήσεων σε μνήμη, αποδοτική διαδικασία deployment, και ανεξαρτησία από την πλατφόρμα εικονικοποίησης. Αυτή την στιγμή τα unikernel που παράγονται από το IncludeOs μπορούν να εκτελεστούν επάνω στους περισσότερους δημοφιλείς επόπτες, καθώς και στο openStack[18].

Το IncludeOs χαρακτηρίζεται από χαμηλό αποτύπωμα μνήμης, υποστήριξη για C++11/17, τις κλασσικές βιβλιοθήκες C++ κ.α.. Επίσης, υπάρχει υποστήριξη και για την βιβλιοθήκη της C γλώσσας, δίνοντας μια συμβατότητα με κάποιες POSIX εφαρμογές[2]. Τέλος, είναι αρκετά ενδιαφέρον το γεγονός πως υλοποιήθηκε από την αρχή μια αρθρωτή στοίβα δικτύου εστιασμένη στο project, που επιτρέπει ταχύτερη ανταλλαγή πακέτων και υψηλότερη απόδοση.

Μια εφαρμογή σε IncludeOs, δεν εκκινεί όπως μια παραδοσιακή εφαρμογή για συμβατικό λειτουργικό με μια main(), αλλά με την εντολή Service::start. Αυτό πρόκειται για την υπηρεσία (service) που οφείλει να υλοποιεί ο προγραμματιστής αν θέλει να εκκινήσει η εφαρμογή του. Οι εφαρμογές πρέπει να γράφονται υποχρεωτικά σε γλώσσα C++. Όπως και στα περισσότερα unikernel framework, δεν υπάρχει υποστήριξη για εικονική μνήμη ούτε ο διαχωρισμός μεταξύ user space και kernel space. Τέλος, εφόσον απουσιάζουν τα νήματα και εκτελείται κώδικας μόνο μιας διεργασίας, πολυνηματικές εφαρμογές ή εφαρμογές που στηρίζονται στην παράλληλη εκτέλεση πολλών εργασιών πρέπει να ξαναγραφούν ώστε να ταιριάζουν με τα χαρακτηριστικά του framework[20].

OSv

Το OSv αποτελεί ένα σύγχρονο λειτουργικό σύστημα το οποίο εστιάζει στην εκτέλεση των εφαρμογών στο cloud. Δημιουργήθηκε το 2013 από την Cloudious Systems με σκοπό να τρέχει επάνω σε μια πληθώρα από hypervisors, και να υποστηρίζει τις περισσοτερες από τις δημοφιλείς γλώσσες, C, C++, Java, Ruby, NodeJS κλπ. Πρόκειται, συνεπώς, για ένα unikernel framework ικανό να εκτελέσει single-process linux εφαρμογές[18].

Σε σχέση με τα υπόλοιπα unikernel frameworks, το OSv χαρακτηρίζεται ως ένα λειτουργικό γενικού σκοπού, που μπορεί να μετατρέψει σχεδόν κάθε εφαρμογή σε unikernel. Ταυτόχρονα, υποστηρίζει τους περισσότερους γνωστούς επόπτες, όπως Xen, KVM, QEMU, VMware, GCE. Αυτή η ιδιότητα το υποχρεώνει να είναι ένα από το πιο σύνθετα, και συνεπώς μεγάλα σε μέγεθος, unikernel, όπου η παραγόμενη εικονική μηχανή είναι της τάξης μερικών δεκάδων megabytes. Εν συγκρίσει, τα περισσότερα unikernels εμφανίζουν συνηθισμένο μέγεθος μερικών kilobytes[18].

Όπως και στα περισσότερα unikernel περιβάλλοντα, υπάρχει μόνο μια εκτελούμενη διεργασία ανά πάσα στιγμή και απουσιάζει ο διαχωρισμός user space και kernel space, οπότε όλος ο κώδικας τρέχει σε ένα μοναδικό χώρο μνήμης. Αυτό προφανώς οδηγεί στην μείωση της καθυστέρησης της εκτέλεσης που σχετίζεται με την αλλαγή του context.

Ένα αξιοσημείωτο χαρακτηριστικό του είναι η χρήση χρονοδρομολογητή (scheduler) για τον συγχρονισμό των εικονικών επεξεργαστών (virtual CPUs), χωρίς την χρήση spinlocks. Τα spinlocks χρησιμοποιούνται όταν τα νήματα που πρέπει να συγχρονιστούν ως προς την πρόσβαση σε ένα κοινόχρηστο πόρο, όπως π.χ. μια ευαίσθητη περιοχή μνήμης, δεν μπορούν να κοιμηθούν. Στα εικονικοποιημένα περιβάλλοντα είναι όμως πιθανόν μια εικονική CPU να πάψει προσωρινά να εκτελεί εντολές, με αποτέλεσμα να υποχρεώνει τις υπόλοιπες να αναμένουν άεργες σπαταλώντας πόρους. Επιλέχθηκε, συνεπώς, όλα τα νήματα στο OSv, εκτός από τον ίδιο τον scheduler, να είναι σε θέση να κοιμηθούν, και άρα να μην απαιτούνται spinlocks[24]. Επίσης, δεδομένου πως η εκτέλεση διαδικτυακών εφαρμογών στο cloud θα είναι ο κύριος στόχος τέτοιων frameworks, οι συσκευές δικτύου σχεδιάστηκαν από την αρχή. Με την τεχνική που ονομάζεται κανάλια δικτύου (network channels), τα πακέτα που έρχονται με διακοπές από το δίκτυο επεξεργάζονται και ταξινομούνται σε ξεχωριστά νήματα, μέσα στο ίδιο context με την εκτελούμενη εφαρμογή επιτυγχάνοντας αυξημένη επίδοση ταχύτητα, απλοποιώντας τον ανταγωνισμό για τους μηχανισμούς από τα διάφορα περιβάλλοντα επικοινωνίας που χρησιμοποιούν το δίκτυο. Τέλος, πολύ ενδιαφέρον είναι πως, σε αντίθεση με τα αρκετά άλλα unikernel frameworks, στο OSv υπάρχει υποστήριξη για εικονική μνήμη. Έτσι εφαρμογές που στηρίζονται σε αυτή, π.χ. με χρήση της mmap(), μπορούν να μετατραπούν πιο εύκολα σε unikernel[24].

Solo5

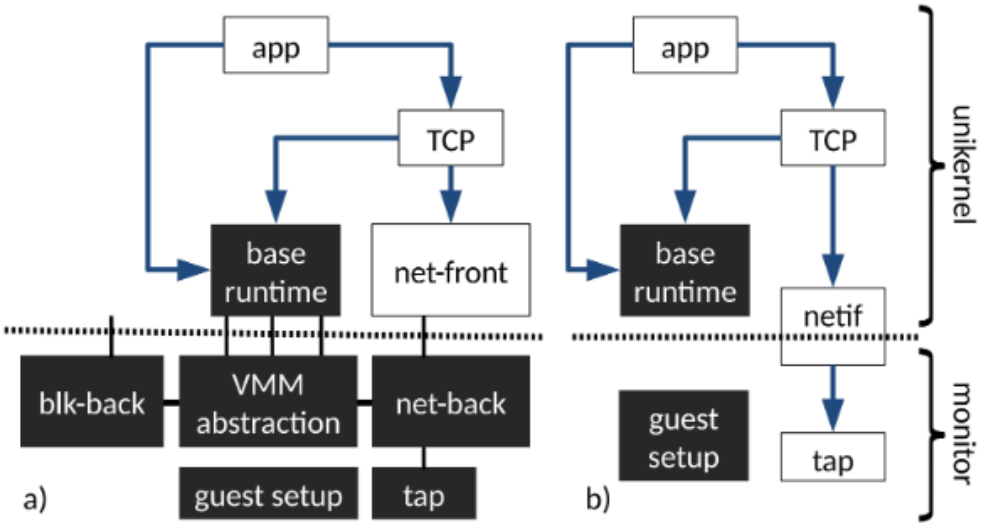
Το Solo5 δεν πρόκειται για ένα πλήρες unikernel framework, όπως τα υπόλοιπα που αναφέρονται στην παρούσα εργασία, αλλά ένα ενδιάμεσο στρώμα επάνω στο οποίο μπορεί να αναπτυχθεί ένα unikernel. Ξεκίνησε από τον Dan Williams της IBM Research, με σκοπό να επιτρέψει στο MirageOs να εκτελείται επάνω στον επόπτη linux-KVM. Πλέον έχει εξελιχθεί σε ένα sandbox περιβάλλον εκτέλεσης και απομόνωσης, επάνω στο οποίο μπορεί να εκτελεστεί ένα μεγάλος αριθμός από Unikernel ή γενικότερα από library operating systems. Μπορούμε να το φανταστούμε ως ένα στρώμα λογισμικού, το οποίο κάθεται ανάμεσα στον επόπτη και στο Unikernel[11].

Όπως έχει αναφερθεί, βασικό συστατικό της φιλοσοφία των Unikernel, και γενικώς των library operating systems, είναι η αφαίρεση των περιττών συστατικών που υπάρχουν σε ένα συμβατικό λειτουργικό. Το Solo5 πηγαίνει αυτή την ιδέα ένα βήμα παραπέρα, θεωρώντας τώρα όλο το unikernel ως μια διεργασία της οποίας η διεπαφή με το hypervisor ή και γενικά με όλο το λειτουργικό σύστημα επανασχεδιάζεται. Αρχικά, είναι επιθυμητό η διεπαφή να είναι lightweight και όσο πιο δυνατόν legacy-free. Για παράδειγμα το QEMU, ένα από τα πιο δημοφιλή hypervisors, εκθέτει στο unikernel μια πληθώρα από εικονικές συσκευές, άχρηστες για το unikernel, ακριβώς επειδή το QEMU είναι σχεδιασμένο να υποστηρίζει πλήρη λειτουργικά συστήματα και όχι αποκλειστικά unikernel[26]. Το Solo5 περιορίζει την διεπαφή μόνο στα συστατικά που χρειάζεται το unikernel. Το κέρδος είναι πως με την αυξημένη αφαίρεση μπορούμε να πετύχουμε μεγαλύτερη φορητότητα (portability) των unikernel ανάμεσα σε διάφορες πλατφόρμες[18]. Είναι εμφανής η αναλογία των σχέσεων μεταξύ διεργασίας-λειτουργικού και unikernel-hypervisor.

Επιπροσθέτως, το Solo5 εισάγει την έννοια του προγράμματος φύλακα (tender), το οποίο ονομάζει hvt (harware virtualized tender). Ο φύλακας είναι ένα εξειδικευμένα στρώμα διεπαφής για unikernels. Ο ρόλος του μοιάζει με αυτόν που έχει παραδοσιακά το QEMU, όμως όπως αναφέρθηκε, το QEMU είναι γενικού σκοπού, ενώ ο hvt στοχεύει σε ένα συγκεκριμένο unikernel. Το interface, λοιπόν, που 'βλέπει' το unikernel είναι αυτό ακριβώς που έχει το ίδιο ανάγκη, και τίποτα παραπάνω, μειώνοντας έτσι την επιφάνεια επίθεσης και το ενδεχόμενο κάτι να δυσλειτουργήσει.

Το κέρδος αυτής της μείωσης της επιφάνειας του interface, είναι πολύ σημαντικό. Η μη ανάγκη για αρχικοποίηση των περιττών στοιχείων επιτρέπει στα unikernel να εκκινούν (boot) αισθητά πιο γρήγορα από άλλα. Η ασφάλεια βρίσκεται σε υψηλότερα επίπεδα, καθώς δεν υπάρχουν περιττά components, με τα οποία θα μπορούσε κάποιος να επηρεάσει το unikernel. Τέλος, το σύνολο unikernel-monitor, μειώνεται δραματικά σε μέγεθος, καθώς δεν χρειάζεται η παρουσία ενός general-purposed και 'βαριού' monitor όπως το QEMU, αλλά ενός lightweight και γρήγορου[26][18].

Το Solo5 χρησιμοποιείται κυρίως από το MirageOS, που ήταν και ο αρχικός στόχος του, και από το IncludeOS. Επίσης υποστηριζόμενες πλατφόρμες-hypervisors



Σχήμα 2.2: a. Monitor γενικού σκοπού b. Monitor ειδικού σκοπού

είναι linux-KVM, FreeBSD-VMM ως hvt, και γενικά οποιοσδήποτε hypervisor σε x86 αρχιτεκτονική, αρκεί να υποστηρίζει με την σειρά του virtio συσκευές, όπως QEMU/KVM[18]. Σημειώνεται επίσης, πως το solo5 ούτε στοχεύει ούτε υποστηρίζει την απευθείας εκτέλεση στο υλικό (bare-metal).

Rumprun

Το Rumprun αποτελεί ένα unikernel framework, που έχει στηριχθεί επάνω στα rump kernels και στοχεύει στην εκτέλεση οποιαδήποτε single-process POSIX εφαρμογής. Αυτή είναι και η ιδέα που ώθησε την δημιουργία του, δηλαδή να μπορεί να μετατραπεί οποιαδήποτε POSIX εφαρμογή σε unikernel, με όσο τον δυνατόν ελάχιστες αλλαγές στον κώδικά της[18][6].

Το project των rump kernels αποτελεί εγχείρημα ώστε να προσφέρονται αυτούσιοι οι drivers και τα συστατικά στοιχεία ενός λειτουργικού συστήματος, ώστε να επιτρέπεται η ανάπτυξη ελαφρών, αρθρωτών και ασφαλών εικονικών μηχανών. Τα rump kernels, χρησιμοποιούν τους drivers του NetBSD λειτουργικού συστήματος, το οποίο με την σειρά του έχει σχεδιαστεί ώστε να είναι όσο το δυνατόν modular, με αποτέλεσμα οι drivers του να είναι αρκετά φορητοί και ανεξάρτητοι πλατφόρμας[23].

Μια δεύτερη ιδέα που οδήγησε στο Rumprun, είναι αυτή του anykernel. Παραδοσιακά, τα λειτουργικά συστήματα χαρακτηρίζοντας από την αρχιτεκτονική του μονολιθικού πυρήνα (monolithic kernel), στην οποία όλες οι προνομιούχες ενέργειες,

όπως η χρονοδρομολόγηση, η διαχείριση της εικονικής μνήμης, η εκτέλεση των οδηγών των συσκευών, γίνονταν στον χώρο του πυρήνα με αυξημένα προνόμια. Όσο μεγάλωνει η πολυπλοκότητα των συστημάτων, αυτή η προσέγγιση γίνεται όλο και πιο δυσκίνητη και ανελαστική. Σε απάντηση αυτού του προβλήματος, εμφανίζονται άλλες εναλλακτικές όπως τα hybrid kernels κ.α. Μια από αυτές είναι το anykernel, όπου περιγράφεται ένας κώδικας βάσης από τον οποίο μπορούν να εξαχθούν από τον πυρήνα οι drivers και να εκτελεστούν με μικρή ή και καθόλου τροποποίηση σε διαφορετικό περιβάλλον ή ακόμα και στον χώρο χρήστη. Αυτός ο διαχωρισμός μεταξύ των drivers και των πιο αυξημένης σημασίας στοιχείων του λειτουργικού δίνει μεγαλύτερη ευελιξία στο όλο σύστημα[23][3].

Αυτήν την στιγμή, τα unikernel από το Rumprun μπορούν να εκτελεστούν επάνω στο Xen, στο QEMU/KVM, αλλά και απευθείας επάνω στο υλικό (bare metal).

Για την υλοποίηση της παρούσας εργασίας επιλέγουμε το Rumprun ως unikernel framework, επάνω στο οποίο εργαζόμαστε, οπότε οφείλουμε να το αναλύσουμε διεξοδικά στην συνέχεια.

Αυτά ήταν κάποια από τα πιο γνωστά και σημαντικά unikernel frameworks. Υπάρχουν πολλά ακόμα όπως το ClickOS, Clive, LING, RuntimeJS, Nanos κ.α[14].

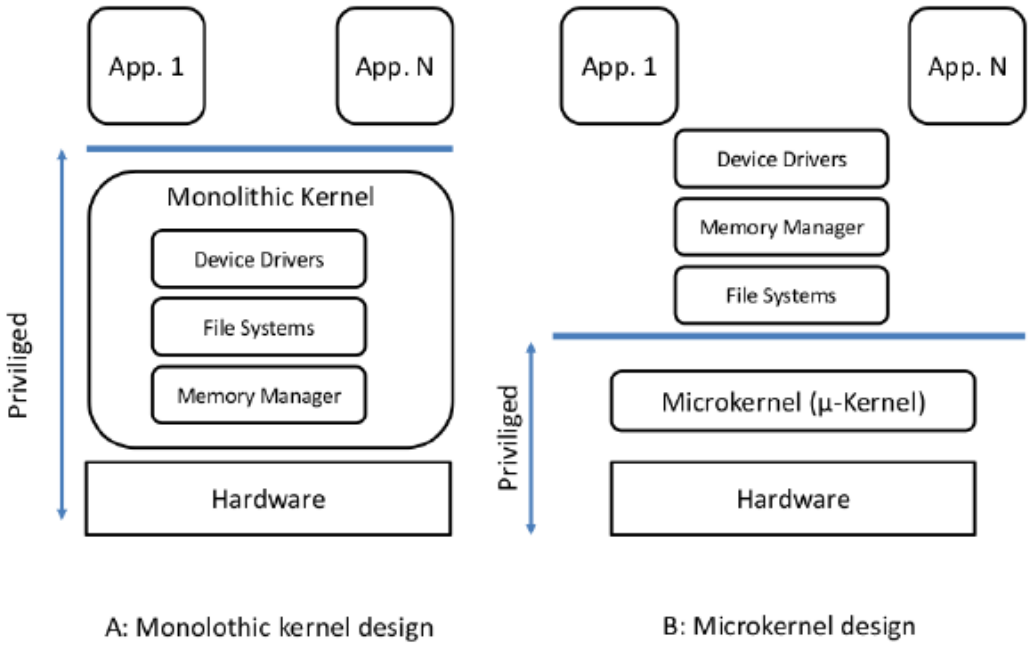
2.1.5 Rumprun

Ιστορία

Οι rump kernels και το Rumprun γεννήθηκαν αρχικά από την ανάγκη εκτέλεσης driver, και συγκεκριμένα του συστήματος αρχείων (file systems), του NetBSD σε χώρο χρήστη. Παρατηρήθηκε πως είναι πολύ πιο γρήγορο και αποτελεσματικό να αναπτύσσονται drivers με αυτόν τον τρόπο, παρά σε ένα εικονικοποιημένο περιβάλλον όπως ήταν το σύνηθες. Με την πάροδο του χρόνου, οι drivers γράφονταν ώστε να είναι όλο και πιο ανεξάρτητοι της πλατφόρμας και φορητοί, έτσι προέκυψαν οι rump kernels. Εν τέλει, προσφέρθηκε η δυνατότητα να εκτελούνται αυτοι οι drivers επάνω στον Xen hypervisor, το οποίο με την σειρά του γέννησε το Rumprun[6][3].

Όπως αναφέρθηκε, το Rumprun framework στηρίζεται στην ιδέα του anykernel, την οποία έννοια επινόησε ο Antii Kantee[3], δημιουργός των rump kernels και του Rumprun. Τα περισσότερα συμβατικά λειτουργικά συστήματα στηρίζονται στην μονολιθική (monolithic) αρχιτεκτονική, όπου όλες οι ενέργειες, οι οποίες απαιτούν αυξημένη δικαιοδοσία, όπως η διαχείριση μνήμης, ο scheduler, το σύστημα αρχείων αλλά και οι drivers για τις περιφερειακές συσκευές, εκτελούνται μέσα στον χώρο πυρήνα. Από τη μία αυτό προσφέρει ασφάλεια, από την άλλη όμως, δυσκολεύει την ανάπτυξη των συστατικών (components) και ειδικά των drivers.

Σε απάντηση του παραπάνω προβλήματος, έχουν εμφανιστεί διαφορετικές αρχιτεκτονικές λειτουργικών, όπως οι microkernels, exokernels κ.α.. Κύριο χαρακτηριστικό αυτών είναι ότι οι πολύ σημαντικές δραστηριότητες του πυρήνα, όπως η διαχείριση μνήμης και ο scheduler, παραμένουν στο χώρο του πυρήνα, ενώ λιγότερο σημαντικά συστατικά, όπως οι drivers των συσκευών και το σύστημα αρχείων, εκτελούνται στον χώρο χρήστη. Αυτό προσφέρει μεγαλύτερη ευελιξία και ασφάλεια κατά την ανάπτυξη των τελευταίων συστατικών.



Σχήμα 2.3: A.Μονολιθική και B.Microkernel αρχιτεκτονική πυρήνα

Το σκεπτικό πίσω από τον όρο anykernel επεκτείνει περαιτέρω τον ανώτερο διαχωρισμό των λειτουριών ενός συστήματος που παραδοσιακά επιτελούνται εντός του πυρηνα. Πλέον, οι drivers του συστήματος ανεξαρτητοποιούνται από το σύστημα στο οποίο εν τέλει θα εκτελεστούν. Αναφερόμαστε σε μια αρχιτεκτονικά αγνωστική προσέγγιση, όπου οι drivers μπορούν είτε να ενσωματώνονται σε κάποιο μονολιθικό πυρήνα, σε κάποιο microkernel, σε κάποιο exokernel, γενικά σε πυρήνες διαφορετικών αρχιτεκτονικών, ή να εκτελούνται ως διεργασία στο user space χωρίς κάποια αλλαγή στον κώδικά τους. Ως drivers δεν εννοούμε μόνο τους οδηγούς συσκευών, αλλά και το σύστημα αρχείων, την στοίβα δικτύου κ.α.. Υπάρχει, λοιπόν, μια εγγενής φορητότητα των drivers[23]. Το λειτουργικό σύστημα NetBSD ταιριάζει στον ορισμό του anykernel[3].

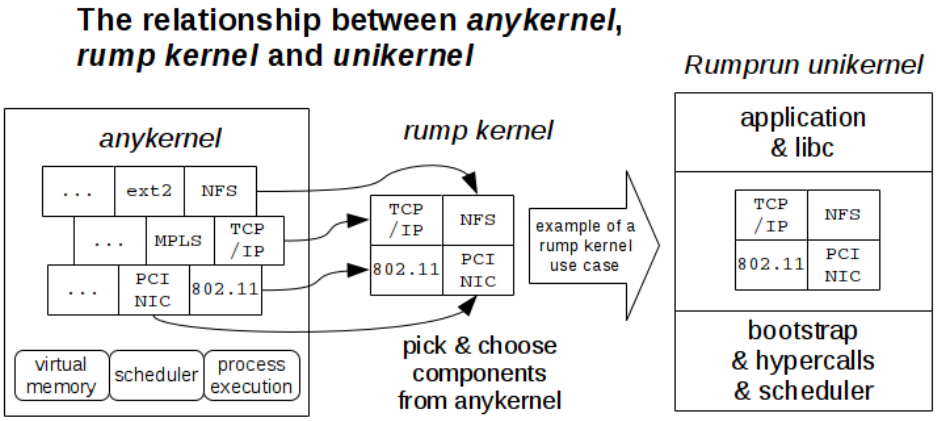
Σχετική έννοια είναι αυτή των rump kernels. Με τον όρο rump εννοούμε ένα κατάλοιπο ενός μεγαλύτερου αρχικού συνόλου. Rump kernel, λοιπόν, είναι ένα ει-

κονικοποιημένο στιγμιότυπο ενός συνόλου από drivers, οι οποίοι τρέχουν εκτός του μονολιθικού πυρήνα. Αυτό πρακτικά μεταφράζεται σε έναν πυρήνα ενός συμβατικού λειτουργικού συστήματος, από το οποίο έχουν αφαιρεθεί διάφορα κομμάτια, όπως η εικονική μνήμη και ο συγχρονισμός μεταξύ διεργασιών. Το υπόλειμμα (rump) αυτής της διαδικασίας είναι οι drivers και οι απολύτως απαραίτητες ρουτίνες, ώστε να μπορεί να εκτελεστεί μία διεργασία[18]. Μπορούμε να φανταστούμε τους rump kernels σαν πυρήνας-ως-υπηρεσία (kernel-as-a-service). Οι rump kernels που δημιουργήθηκαν από τον Antii Kantee εξάγουν τους drivers και τα στοιχεία από το λειτουργικό σύστημα NetBSD, δηλαδή από τον open-source πηγαίο κώδικά του[3].

Το rumprum unikernel είναι μια εφαρμογή των rump kernels, που έχει στόχο να εκτελούνται προγράμματα ως unikernel. Υπενθυμίζεται ότι ο αρχικός στόχος του εγχειρήματος των rump kernel ήταν να εκτελούνται αυτόνομα drivers στο user space, ώστε να διευκολύνεται η ανάπτυξη και η αποσφαλμάτωσή του. Όμως, με την πάροδο του χρόνου, αποδείχθηκε πως οι rump kernels μπορούν να εκτελεστούν ως εικονικοποιημένος guests επάνω στον Xen hypervisor. Η πρώτη εφαρμογή αυτού του «παντρέματος» ήταν η εκτέλεση ελαφρών router ή firewalls ως guests, λειτουργίες που παραδοσιακά βρίσκονταν στον χώρο ενός μονολιθικού πυρήνα. Σύντομα προστέθηκε και η ύπαρξη μια διεπαφής από τον χώρο χρήστη, δηλαδή ένας τρόπος να καλούνται οι κλήσεις συστήματος[6]. Με όλα αυτά τα επίπεδα λογισμικού διαθέσιμα, είναι προφανές ότι οι rump kernels έχουν την δυνατότητα να παράγουν unikernels, τα οποία ονομάστηκαν Rumprun unikernels. Η σχέση μεταξύ του Rumprun και των rump kernels, είναι πως για την ανάπτυξη του unikernels, το Rumprun επιλέγει από το σύνολο που προσφέρουν οι rump kernels μόνο τα στοιχεία που είναι απαραίτητα για την εκτέλεση μιας συγκεκριμένης εφαρμογής, το οποίο σημαίνει πως το Rumprun είναι μια από τις πολλές εφαρμογές (use cases) των rump kernels. Η εικόνα 2.4 παρουσιάζει καλύτερα την σχέση μεταξύ του anykernel των rump kernels και του Rumprun unikernel.

Δομή και Ανάπτυξη του Unikernel

Η στοίβα λογισμικού του Rumprun, θυμίζει σε ένα βαθμό την στοίβα εκτέλεσης μιας διεργασίας σε κάποιο συμβατικό λειτουργικό σύστημα. Στο ανώτερο επίπεδο βρίσκεται ο κώδικας της εφαρμογής, ο οποίος στηρίζεται σε βιβλιοθήκες του user space, όπως π.χ. η libc για την γλώσσα C. Από κάτω βρίσκονται οι κώδικας που αναλαμβάνει να καλέσεις τις κλήσεις συστήματος (system calls) προς τον πυρήνα. Μια διαφορά που υπάρχει στο Rumprun είναι πως οι κλήσεις αυτές δεν γίνονται με κάποια trap εντολή ώστε να αλλάξει το επίπεδο προνομίων με το οποίο εκτελείται ο κώδικας, αλλα καλείται απλά μια απλή συνάρτηση βιβλιοθήκης η οποία αναλαμβάνει να εξυπηρετήσει την αίτηση. Υπενθυμίζεται πως σε unikernel όλη η εκτέλεση γίνεται σε ένα μοναδικό ενιαίο χώρο μνήμης, δεν υπάρχει διάκριση μεταξύ user space και kernel space. Για να το πετύχει αυτό, το Rumprun αντικαθιστά όλες τις κλήσεις συστήματος (system calls) του NetBSD με δικές του κλήσεις πυρήνα (rump kernel calls), οι



Σχήμα 2.4: Σχέση μεταξύ anykernel-rump kernels-Rumprun

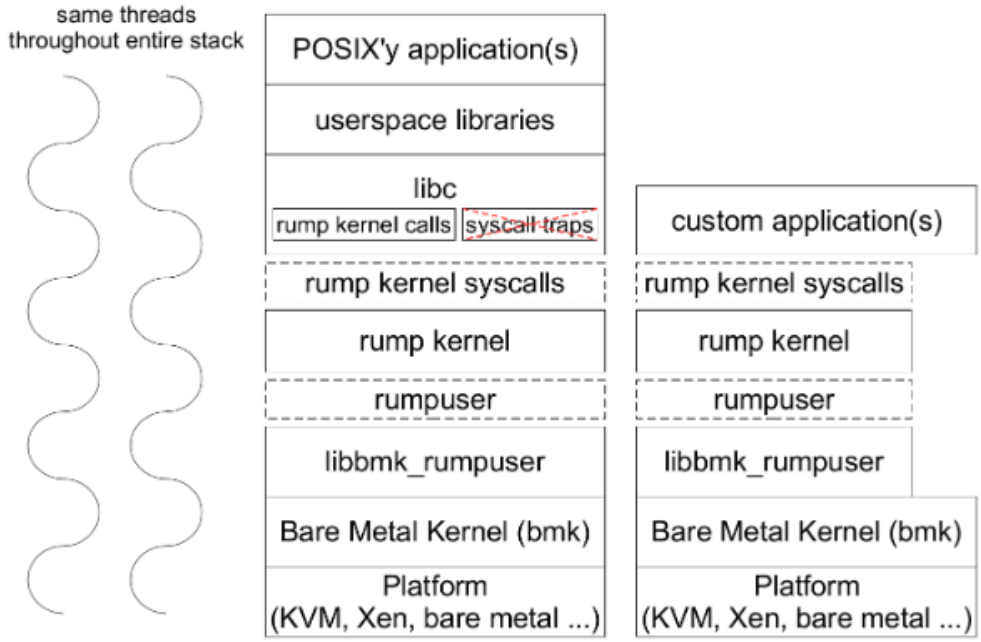
οποίες είναι στην ουσία κλήσεις συναρτήσεων (library calls). Από αυτό το επίπεδο και μέχρι το επίπεδο του επόπτη βρίσκεται ό,τι θα έπρεπε να βρίσκεται στον πυρήνα ενός συμβατικού λειτουργικού.

Εύλογα θα περίμενε κανείς, στα κατώτερα επίπεδα να βρίσκονται κομμάτια λογισμικού μόνο από τους rump kernels. Εντούτοις, οι rump kernels δεν σχεδιάστηκαν ώστε να τρέχουν επάνω σε κάποιο επόπτη ως εικονική μηχανή, οπότε τους λείπει η δυνατότητα να επικοινωνούν με το χαμηλότερο επίπεδο του επόπτη. Ταυτόχρονα απουσιάζουν και οι μέθοδοι εκκίνησης (bootstrap), χρονοδρομολόγησης, διακοπές (interrupt), και η διαχείριση των σελίδων της μνήμης. Για αυτόν ακριβώς τον λόγο, στην στοίβα του Rumprun κάτω από το κομμάτι των rump kernels έχει προστεθεί ένα στρώμα υλικού που ονομάζεται Bare Metal Kernel (bmk), και που ο ρόλος του είναι να επιτελεί όλες αυτές τις σημαντικότατες προαναφερθείσες λειτουργίες. Ο bmk αποτελείται τόσο από κώδικα που είναι κοινός για κάθε πλατφόρμα στην οποία στοχεύει το Rumprun, όσο και από κώδικα ειδικό για κάθε πλατφόρμα ξεχωριστά[18]. Παράλληλα, υπάρχει και ένας μηχανισμός επικοινωνίας των rump kernels με τον bmk, που ονομάζεται rumpuser[23]. Οι βασικοί στόχοι του Rumprun είναι είτε ο Xen hypervisor, είτε το KVM των Linux, είτε απευθείας το υλικό (bare metal).

Η δομή της στοίβας φαίνεται στην εικόνα 2.5

Αξίζει, επίσης, να περιγραφεί η διαδικασία ανάπτυξης ενός unikernel με το Rumprun framework. Η διαδικασία αποτελείται συνοπτικά από τέσσερα βήματα.

1. Αρχικά αναπτύσσεται ο κώδικας της εφαρμογής σε user space. Το Rumprun στοχεύει στην εκτέλεση POSIX εφαρμογών, οπότε η ανάπτυξη δεν διαφέρει από συμβατικά λειτουργικά, και συνήθως μπορεί να χρησιμοποιηθεί κώδικας



Σχήμα 2.5: Στοίβα λογισμικού rump kernels/Rumprun

από εφαρμογές ήδη γραμμένες για κάποια άλλη πλατφόρμα, όπως το linux. Ωστόσο, συγκεκριμένες λειτουργίες, όπως οι πολλές διεργασίες και η εικονική μνήμη δεν υποστηρίζονται, άρα εφαρμογές που στηρίζονται σε κλήσεις όπως fork() ή η mmap() πρέπει να προσαρμόζονται.

2. Στην συνέχεια ο κώδικας γίνεται compiled σε ένα ή περισσότερα object file, όπως κατά συμβατικές διαδικασίες μεταγλώττισης. Το Rumprun στηρίζεται σε cross-compilation, όποτε δεν υπάρχει ανάγκη ύπαρξης κάποιου unikernel το οποίο μπορεί να μεταγλωτίζει, αλλά αντί αυτού προσφέρονται έτοιμοι compilers.

3. Έπειτα έρχεται η σειρά του linking αυτών των αντικειμένων. Δεν πρόκειται για συμβατικό linking, αλλά για pseudo-linking, καθώς οι εξαρτήσεις από το λειτουργικό σύστημα, για παράδειγμα οι κλήσεις συστήματος, δεν γίνονται resolve αλλά γίνεται μόνο έλεγχος για την ορθή συντακτική χρήση τους. Συνεπώς, δεν επιτελείται κάποια σύνδεση με τα συστατικά του λειτουργικού συστήματος, που σε άλλη περίπτωση θα επέτρεπαν την επικοινωνίας κατά το runtime. Σε αυτή την φάση, παράγεται ένα αρχείο το οποίο θυμίζει εκτελέσιμο, όμως δεν είναι.

4. Τέλος, επέρχεται το λεγόμενο bake, δηλαδή οι προαναφερθείσες εξαρτήσεις συνδέονται με τις συναρτήσεις των rump kernels που αναλαμβάνουν τις λειτουργίες που ζητούνται. Από το σύνολο των drivers των rump kernels εξάγονται μόνο όσοι χρειάζονται πραγματικά από την εφαρμογή. Ταυτόχρονα, προστίθε-

νται και τα κομμάτια που χρειάζονται για την επικοινωνία με τον επόπτη στον οποίο στοχεύουμε.

Με αυτόν τον τρόπο παράγεται το τελικό unikernel, η μικρή, ασφαλής και γρήγορη εικονική μηχανή, που θα επιτελεί την λειτουργία για την οποία σχεδιάστηκε.

Γιατί Rumprun

Όπως αναφέρθηκε, επιλέγεται το Rumprun να είναι το framework επάνω στο οποίο εργαζόμαστε για την υλοποίηση της παρούσας εργασίας. Θυμίζεται πως επιθυμείται να ενσωματώθει η λειτουργία μνήμης utmem, η οποία αναλύεται εκτενέστερα στην συνέχεια, σε κάποιο unikernel framework. Το rump kernels - Rumprun επιλέχθηκε ως κατάλληλο εργαλείο για τους εξής λόγους:

- Ο μηχανισμός utmem, τουλάχιστον για το frontend που χρειάζεται να υλοποιηθεί από την αρχή, είναι ουσιαστικά ένας driver που επικοινωνεί με τον hypervisor. Αφού, λοιπόν, τα rump kernels επιτρέπουν την εξαγωγή drivers γραμμένους για το NetBSD, ένα well-documented λειτουργικό σύστημα, είναι εύλογο να ταιριάζουν στο use case μας και να διευκολύνουν το έργο μας.

- Το Rumprun στοχεύει στο να υποστηρίζει την μετατροπή των περισσότερων εφαρμογών οι οποίες υπακούν στο POSIX πρότυπο σε unikernel. Συνέπεια αυτού είναι πως δεν χρειάζεται ιδιαίτερη προσαρμογή επιλογών για να αναπτυχθεί user space κώδικας που να χρησιμοποιεί την utmem, καθώς η συντριπτική πλειοψηφία των προγραμματιστών γνωρίζει το POSIX πρότυπο,

- Στην στοίβα του Rumprun απουσιάζει από επιλογή των σχεδιαστών η εικονική μνήμη. Αυτή η επιλογή έγινε για να είναι όσο το δυνατόν ελαφρύτερα και απλούστερα τα τελικά unikernel. Ως εκ τούτου, το να ενσωματωθεί ο μηχανισμός της utmem σε ένα τέτοιο unikernel framework, στην πράξη προσφέρει λύση σε ένα υπαρκτό ενδεχόμενο, δηλαδή στην διαχείριση περιορισμένης μνήμης. Η συγκεκριμένη εργασία επικεντρώνεται στην αντιμετώπιση αυτού του προβλήματος. Όπως θα φανεί και στην συνέχεια, πράγματι έχει ουσιαστική αξία η υλοποίηση της εργασίας.

2.2 Μνήμη - utmem

2.2.1 Ιστορία

Η διαχείριση της μνήμης ενός υπολογιστικού συστήματος είναι ένα πολύ σημαντικό πρόβλημα που καλείται να λύσει ο μηχανικός υπολογιστικών συστημάτων. Η μνήμη αποτελεί σημαντικότατο πόρο, αν όχι τον σημαντικότερο. Στα συμβατικά λειτουργικά συστήματα, κάθε διεργασία που εκτελείται νομίζει πως βρίσκεται σε έναν ενιαίο χώρο μνήμης, τον οποίον δεν μοιράζεται με άλλες εργασίες, οι οποίες μπορεί να βρίσκονται

στο σύστημα την στιγμή της εκτέλεσης. Το λειτουργικό σύστημα από την άλλη, οφείλει να αντιστοιχεί αυτόν τον χώρο διευθύνσεων που «βλέπει» η διεργασία (virtual address space), σε κάποιο κομμάτι της φυσικής μνήμης του συστήματος (physical address space), χρησιμοποιώντας μηχανισμούς και πολιτικές τόσο σε επίπεδο υλικού όσο και λογισμικού.

Πρόβλημα εμφανίζεται όταν η φυσική μνήμη του συστήματος γεμίσει από εγγραφές και πλέον το λειτουργικό αδυνατεί να κατανείμει περισσότερη στα προγράμματα. Αν δεν αντιμετωπιστεί αυτό το πρόβλημα, και απλά αρνηθεί το λειτουργικό σύστημα να δώσει νέα μνήμη, τότε προκύπτει ένα ασταθές, δυσλειτουργικό και αναποτελεσματικό σύστημα. Η λύση, που πλέον ενσωματώνεται σε όλα τα σύγχρονα λειτουργικά συστήματα όπως Linux, Windows κλπ, είναι η ύπαρξη της εικονικής μνήμης (virtual memory), δηλαδή ενός μηχανισμού που επιτρέπει στο σύστημα να δίνει στα προγράμματα την ψευδαίσθηση πως υπάρχει περισσότερη διαθέσιμη μνήμη από την φυσική. Όταν η μνήμη γεμίσει, σελίδες (memory pages) αυτής εναλλάσσονται (swapping) σε κάποιο άλλο αποθηκευτικό μέσο, διαθέσιμο στο σύστημα, το οποίο συνήθως είναι κάποιος δίσκος. Όταν χρειαστεί να προσπελαστούν οι σελίδες που έχουν μεταφερθεί στον δίσκο τότε αντιγράφονται πίσω στην κύρια μνήμη, στην θέση κάποιων άλλων οι οποίες με την σειρά τους θα μετακινηθούν στον δίσκο. Οι αποθηκευτικοί δίσκοι έχουν χωρητικότητα αρκετές φορές μεγαλύτερη από αυτήν της φυσικής μνήμης, έτσι λοιπόν επιτυγχάνεται η ψευδαίσθηση της περισσότερης διαθέσιμης μνήμης[15]. Το μεγάλο μειονέκτημα αυτής της τεχνικής είναι πως, επειδή η ταχύτητα ανάγνωσης και εγγραφής του δίσκου είναι τάξεις μεγέθους πιο αργή από της φυσικής μνήμης, η επίδοση και ταχύτητα εκτέλεσης των προγραμμάτων μειώνονται δραματικά. Για εφαρμογές που μας ενδιαφέρει η υψηλή ταχύτητα εκτέλεσης και χαμηλή καθυστέρηση, όπως π.χ. μια διαδικτυακή εφαρμογή που προσφέρει περιεχόμενο σε πελάτες, αυτό είναι καταστροφικό.

Σε περιβάλλον εικονικοποίησης και ειδικά σε cloud virtualization, το ανώτερο πρόβλημα γίνεται ακόμα πιο σύνθετο. Ένα λειτουργικό που εκτελείται ως guest, αγνοεί το πραγματικό μέγεθος της φυσικής μνήμης του μηχανήματος επάνω στο οποίο εκτελείται, με αποτέλεσμα σε αρκετές περιπτώσεις η μνήμη να υποχρησιμοποιείται. Επίσης, και ο host αγνοεί την οποιαδήποτε ύπαρξη μηχανισμών διαχείρισης της μνήμης από τον guest[29]. Έστω το εξής σενάριο παράλληλης εικονικοποίησης δύο guest λειτουργικών συστημάτων επάνω στο ίδιο μηχάνημα: ο πρώτος guest έχει εξαντλήσει όλη την μνήμη η οποία του έχει ανατεθεί, και άρα έχει οδηγηθεί σε λιγότερο αποδοτικές λύσεις όπως το swapping, ενώ ο δεύτερος έχει ακόμα ένα μεγάλο ποσοστό της μνήμης του ελεύθερο. Έτσι, παρόλο που η συνολική μνήμη θεωρητικά αρκεί για τις ανάγκες όλων, εντούτοις δεν χρησιμοποιείται σωστά, και μέρος αυτής παραμένει αναξιοποίητο λόγω κακής κατανομής.

Έχουν προταθεί διάφοροι τρόποι αντιμετώπισης αυτού του γενικού προβλήματος, οι οποίοι συνοψίζονται σε

1. Υπερανάθεση πόρων (Overprovisioning). Σε αυτήν την περίπτωση οι σχεδιαστές δρουν σκεπτόμενοι πάντα το χειρότερο δυνατό σενάριο. Δηλαδή, αναθέτουν στους guests αρκετή μνήμη ώστε πρακτικά να εξαλείφεται το ενδεχόμενο να μην τους αρκέσει και να στραφούν στην εικονική μνήμη. Το μεγάλο μειονέκτημα, είναι πως πρώτον χρειαζόμαστε να υπάρχει διαθέσιμη ως φυσική μνήμη όλη αυτή που παραχωρείται, οπότε μπορούν να υποστηριχθούν λιγότεροι guests ανά μονάδα μνήμης, και δεύτερον πως προφανώς ο πόρος της μνήμης πάλι υποχρησιμοποιείται.

2. Εναλλαγή στον δίσκο (swapping). Αυτό είναι ή άλλη όψη του νομίσματος, δηλαδή να επιτρέπεται οι guests να καταφεύγουν στην εικονική τους μνήμη σε περίπτωση που τελειώσει η διαθέσιμή του. Μειώνεται η μνήμη που ανατίθεται σε κάθε γυεστς, αδιαφορώντας για πιθανούς εσωτερικούς μηχανισμούς διαχείρισής της. Μπορούν να υποστηριχθούν περισσότεροι guests ανά μονάδα μνήμης, αλλά η επίδοση κάποιων εξ αυτών χάνεται.

3. Ballooning. Η πιο ισορροπημένη και δημοφιλής μέθοδος. Ουσιαστικά αυξομειώνεται δυναμικά η διαθέσιμη μνήμη του guest κατά το runtime, ανάλογα με τις ανάγκες του. Για να λειτουργήσει το ballooning πρέπει host και guest να συνεργαστούν, δηλαδή αποτελεί εφαρμογή του paravirtualization. Αρχικά, τονίζεται πως host και guest «βλέπουν» διαφορετικά την μνήμη. Ο guest νομίζει πως διαχειρίζεται διευθύνσεις φυσικής μνήμης, όπως αν εκτελούνταν απευθείας επάνω στο υλικό, όμως στην ουσία αποτελεί μια διεργασία του host λειτουργικού, ή πιο γενικά του hypervisor. Άρα οι διευθύνσεις του είναι ουσιαστικά εικονικές, τις οποίες οφείλει ο host να αντιστοιχεί σε φυσική μνήμη. Με το ballooning, ο guest εναποθέτει τις σελίδες μνήμης (memory pages) του τις οποίες δεν χρησιμοποιεί σε μια ειδική συσκευή στον πυρήνα του λειτουργικού συστήματος, σαν να δηλώνει πως παραιτείται από την κυριότητά του επάνω σε αυτές. Με την σειρά της, η συσκευή επικοινωνεί με τον hypervisor, ο οποίος πλέον γνωρίζει πως δεν χρειάζεται να αντιστοιχεί αυτές τις εικονικές διευθύνσεις μνήμης που του έδωσε ο guest σε φυσική μνήμη, και άρα η πλέον απελευθερωμένη μνήμη μπορεί να χρησιμοποιηθεί για άλλο σκοπό, και όχι να παραμένει ανενεργή. Όταν πάλι ο guest χρειαστεί περισσότερη μνήμη, δηλώνει στην συσκευή πως ξαναπαίρνει την κυριότητα αυτών των σελίδων, οπότε με την σειρά του ο host αντιστοιχεί πάλι φυσική μνήμη σε αυτές τις σελίδες. Το μειονέκτημα της συγκεκριμένης τεχνικής πηγάζει από το γεγονός πως τα σύγχρονα λειτουργικά συστήματα είναι σχεδιασμένα ώστε να χρησιμοποιούν όλη την διαθέσιμη μνήμη του συστήματος, θεωρώντας πως είναι η μόνη οντότητα που έχει ανάγκη από την υφιστάμενη φυσική μνήμη. Για παράδειγμα, το page cache υποσύστημα στο linux, αντιστοιχεί σελίδες δεδομένων του δίσκου σε αχρησιμοποίητες από διεργασίες σελίδες της κύριας μνήμης. Έτσι ένας guest δεν έχει λόγο να αποδεσμεύσει τις σελίδες του χρησιμοποιώντας την συσκευή ballooning. Προκύπτει, λοιπόν, αυτομάτως μια σχέση ανταγωνισμού, και όχι συνεργασίας για την μνήμη ανάμεσα στον host και τον guest[29].

2.2.2 Transcedent memory - tmem

Μια διαφορετική λύση, στο πρόβλημα της διαχείρισης της μνήμης σε virtualized περιβάλλοντα, είναι αυτή του μηχανισμού ελαστικής μνήμης (transcendent memory ή tmem). Η tmem πρόκειται για ένα linux kernel module το οποίο αναπτύχθηκε από την εταιρία Oracle το 2009, για τον Xen hypervisor, και ενσωματώθηκε στον πυρήνα του linux, σχεδόν άμεσα[27].

Η βασική φιλοσοφία πίσω από την ιδέα αυτή, είναι να ανταλλάσσεται μνήμη, ή μάλλον δεδομένα, μεταξύ guest και host χρησιμοποιώντας μια βάση δεδομένων η οποία λειτουργεί με πρότυπο αναφοράς κλειδιού-τιμής (key-value)[19]. Τα δεδομένα σε αυτήν την βάση θα είναι σελίδες μνήμης του guest, όχι κενές αλλά με εγγραφές εντός αυτών. Διαφέρει με την τεχνική του ballooning, στο γεγονός πως η μνήμη που ανατίθεται με την tmem δεν είναι προσπελάσιμη απλά με αναφορά σε κάποια διεύθυνση μνήμης με κάποιο απλό load-store. Η πρόσβαση στην μνήμη αυτή, γίνεται μόνο μέσω ειδικών κλήσεων επόπτη (hypercalls), χρησιμοποιώντας το κατάλληλο κλειδί κάθε φορά. Αυτό, εκ πρώτης, ακούγεται αποθαρρυντικό, ίσως και λάθος σχεδιαστικά, όμως επιτρέπει μεγαλύτερη ευελιξία στην μορφή που εν τέλει αποθηκεύονται τα δεδομένα. Ο host, ή εν γένει οποιοδήποτε σύστημα καλείται να αποθηκεύσει την μνήμη αυτή, μπορεί να την μετασχηματίσει, να την τροποποιήσει και να την μεταφέρει οπουδήποτε κρίνει αυτός κατάλληλο, εφόσον δεσμεύεται πως μπορεί να την ανακτήσει και να την επιστρέψει πίσω στον guest, όταν και αν αυτός το ζητήσει[25].

Η μηχανισμός της tmem, ακουλουθεί μια απλή και προσεκτικά σχεδιασμένη διεπαφή, ικανή να καλύψει ένα ευρύ φάσμα περιπτώσεων. Η διεπαφή αυτή έχει σχεδιαστεί με γνώμονα να είναι ευέλικτη και να έχει ελάχιστο αποτύπωμα και κόστος στον πυρήνα που την χρησιμοποιεί. Όπως αναφέρθηκε, η tmem δεν είναι άμεσα προσπελάσιμη από τον guest, αλλά έμμεσα μέσα από ειδικές αιτήσεις στον επόπτη , ή εν γένει σε όποιον την διαχειρίζεται. Δημιουργείται στην αρχή μια δεξαμενή μνήμης (pool), με ξεχωριστό pool id. Επίσης, δημιουργούνται μοναδικά κλειδιά (handles), τα οποία αποτελούν αντίστοιχο ρόλο με τις διευθύνσεις των σελίδων μια συμβατικής μνήμης. Τα κλειδιά αυτά είναι μοναδικά για κάθε κομμάτι μνήμης που αποθηκεύεται στο pool, αλλά όχι μεταξύ διαφορετικών pools. Οι βασικές ενέργειες επάνω στην tmem είναι η λήψη (Get) και η τοποθέτηση (Put). Όταν ο guest, επιθυμεί να αποθηκεύσει δεδομένα τότε εκτελεί μια αίτηση Put στον hypervisor, αναφέροντας το σωστό pool id και τα σωστά κλειδιά, καθώς και την περιοχή μνήμης του στην οποία βρίσκονται τα δεδομένα που θέλει να αποθηκευτούν[27]. Αντίστοιχα, όταν θέλει να ανακτήσει τα δεδομένα, εκτελεί μια αίτηση Get, με τις αντίστοιχες παραμέτρους. Η μηχανισμός tmem, σε αντίθεση με I/O μηχανισμούς, είναι σύγχρονος όσον αφορά την αντιγραφή των δεδομένων, και ως εκ τούτου προκύπτει η ανάγκη ύπαρξης δικλείδων ασφαλείας για να αποφεύγονται deadlocks[25].

Όπως φαίνεται μέχρι στιγμής, η μηχανισμός tmem στηρίζεται στην επικοινωνία δύο μορφών χρηστών, συγκεκριμένα μεταξύ του frontend, δηλαδή του μηχανισμού-

χρήστη που επιθυμεί να αποθηκεύει κομμάτια μνήμης του, και του backend, δηλαδή του μηχανισμού-χρήστη ο οποίος αναλαμβάνει την αποθήκευση. Μηχανισμοί backend μπορούν να είναι διάφοροι, ενδεικτικά:

1. Transcendent memory for Xen. Ο Xen hypervisor ήταν ο αρχικός στόχος του εγχειρήματος, οπότε είναι ο πιο ώριμος από τα backend. Αποθηκεύει στην μνήμη του hypervisor τα δεδομένα, ενώ υποστηρίζει πολλαπλούς guests, συμπίεση και μοναδική αποθήκευση διπλότυπων δεδομένων, αν εμφανίζονται από διαφορετικούς guests, για εξοικονόμηση χώρου.

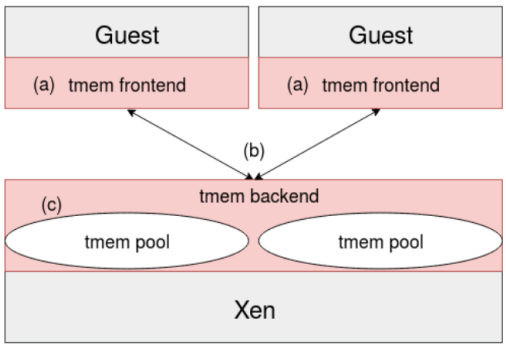
2. Zcache. Η Zcache επιτρέπει την αποθήκευση πολλαπλάσιων σελίδων μνήμης μέσω της tmem όταν η μνήμη δεν είναι διαθέσιμη. Το καταφέρνει αυτό χρησιμοποιώντας συμπίεση δεδομένων μέσα στον πυρήνα. Δεν αφορά απαραίτητα κάποιον hypervisor, αφού η ανταλλαγή δεδομένων μπορεί να γίνεται ανάμεσα σε δύο μέρη του ίδιου ίδιου πυρήνα, οδηγεί όμως σε καλύτερη αξιοποίηση των πόρων[25].

Μεγαλύτερο ενδιαφέρον παρουσιάζουν οι χρήστες frontend, δηλαδή ποια συστήματα ενός λειτουργικού συστήματος αξίζει να χρησιμοποιούν την tmem. Αυτά είναι:

1. Το frontswap subsystem. Η λειτουργία του στον πυρήνα του linux είναι να εναλλάσσει (swap) σελίδες μνήμης από την κύρια μνήμη σε κάποιο δευτερεύον αποθηκευτικό μέσο. Χρησιμοποιεί μόνιμα σύνολα, που σημαίνει για οποιοδήποτε σελίδα μνήμης αποθηκευτεί στο frontswap, υπάρχει εγγύηση πως θα βρίσκεται εκεί ανά πάσα στιγμή το αναζητήσει πίσω αυτός που το τοποθέτησε. Ως χρήστης του tmem μηχανισμού, το frontswap επιλέγει να προωθεί τις σελίδες μνήμης στα backend της tmem, και να μην τις στέλνει απευθείας στο δίσκο που είναι τάξεις μεγέθους πιο αργός από την κύρια μνήμη. Υπάρχει, βέβαια, πάντα το ενδεχόμενο το backend να απορρίψει την αίτηση tmem για διάφορους λόγους που επιλέγει το ίδιο, και τότε να πρέπει το frontswap να καταφύγει στον δίσκο. Γενικώς, επιτυγχάνεται αυξημένη ταχύτητα πρόσβασης στις σελίδες οι οποίες δίνονται στο frontswap από την κύρια μνήμη.

2. Το cleancache subsystem. Κατά την εκτέλεση των προγραμμάτων, είναι πολύ συχνό να μεταφέρονται δεδομένα από τον δίσκο στην κύρια μνήμη και τούμπαλιν. Στο Linux, η πρόσβαση στις συσκευές και άρα και στον δίσκο γίνεται μόνο μέσα από έγκριση του πυρήνα, και καμία διεργασία δεν μπορεί να τις χρησιμοποιήσει απευθείας. Έτσι, αυτές οι συναλλαγές περνάν από τον πυρήνα πρώτα. Η cleancache συμπεριφέρεται κατά αντιστοιχία με την cache στις CPUs, αλλά αντί να διατηρεί εφήμερα δεδομένα ανάμεσα στον επεξεργαστή και την κύρια μνήμη, διατηρεί σελίδες δεδομένων ανάμεσα στην μνήμη και στον δίσκο. Θυμίζεται πως αυτά τα δεδομένα βρίσκονται και ως αντίγραφα σε άλλες θέσεις, είτε στον δίσκο είτε στην μνήμη, και πως είναι και εφήμερα, που σημαίνει πως μπορεί στην θέση του να γραφτούν άλλα και άρα δεν υπάρχει εγγύηση για την

ύπαρξη τους ανά πάσα στιγμή. Με την tmem, αυτά τα δεδομένα μπορούν να αποθηκευτούν σε κάποιο tmem pool, και άρα η απουσία τους από τη clean-cache να μην οδηγεί απαραίτητα σε ανάκτηση (fetch) από τον δίσκο, που είναι και αρκετά αργή διαδικασία[19].



Σχήμα 2.6: Tmem επάνω στον Xen hypervisor

2.2.3 User space transcedent memory - utmem

Ο μηχανισμός tmem, που αναλύθηκε προηγουμένω, αφορά περιπτώσεις χρήσης από υποσυστήματα που τρέχουν «σιωπηλά» εντός του πυρήνα. Ένας προγραμματιστής, όχι μόνο αγνοεί την ύπαρξη ενός τέτοιου συστήματος, αλλά ούτε είναι σε θέση να αναπτύξει προγράμματα ικανά να χρησιμοπούν ρητά τον μηχανισμό. Δημιουργείται, συνεπώς, η εύλογη απορία αν θα ήταν δυνατόν ένας τέτοιος μηχανισμός να είναι διαθέσιμος στο user space.

Η απάντηση δόθηκε σε μια προηγούμενη διπλωματική εργασία του εργαστηρίου υπολογιστικών συστημάτων του Εθνικού Μετσοβίου Πολυτεχνείου, όπου υλοποιήθηκε ένας τέτοιος μηχανισμός από τον Αιμίλιο Τσαλαπάτη[19]. Ονομάστηκε utmem (από το user space tmem). Αποδείχθηκε, μάλιστα, πειραματικά πως σε συγκεκριμένες περιπτώσεις πίεσης μνήμης οι διεργασίες που χρησιμοποιούν αυτόν τον μηχανισμό utmem συμπεριφέρονται καλύτερα από διεργασίες που στηρίζονται σε tmem μόνο έμμεσα με χρήση του frontswap subsystem[19][29].

Η δομή του νέου μηχανισμού είναι πολύ απλή, και αφορά την επικοινωνία ενός linux guest, ο οποίος τρέχει επάνω στο KVM, και ενός linux host. Αυτή τη στιγμή, η utmem συνεργάζεται μόνο με το KVM ως hypervisor.

Αρχικά σχεδιάστηκε μια εικονική συσκευή (virtual device), που ονομάστηκε dev/utmem, η οποία εκθέτει τον μηχανισμό σε διεργασίες του user space. Για να τον χρησιμοποιήσει μια διεργασία, πρέπει αρχικά να ανοίξει την συσκευή (open), και

ύστερα με κλήση συστήματος τύπου ioctl, να στέλνει αιτήματα στον πυρήνα του λειτουργικού να αναλάβει να διαχειριστεί κομμάτια από την μνήμη της. Εσωτερικά, η διεπαφή έρχεται από την αυθεντική tmem που σχεδίασε η Oracle. Υπάρχουν ενέργειες Put και Get, για ανταλλαγή δεδομένων, καθώς και Invalidate ώστε να ακυρώνονται κλειδιά που αντιστοιχούν σε δεδομένα τα οποία πλέον δεν χρειάζεται η εφαρμογή.

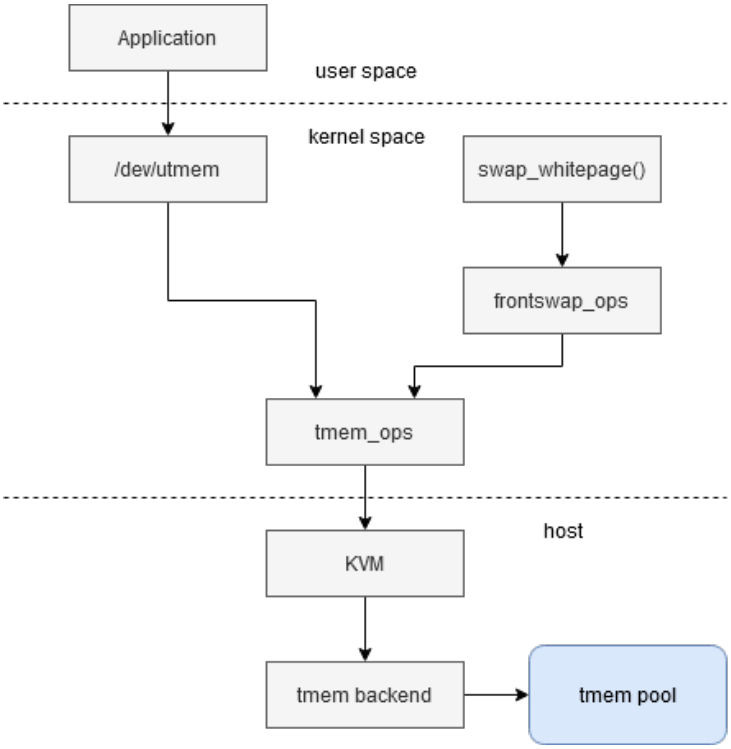
Στην συνέχεια, η εσωτερική δομή της συσκευής δέχεται το αίτημα και τα δεδομένα της διεργασίας και ελέγχει πως όλα είναι σωστά. Αυτό γίνεται πλέον στον χώρο πυρήνα. Αν ο έλεγχος είναι επιτυχής, τότε ζητά μέσω κλήση επόπτη (hypercall), στο backend να αναλάβει το αίτημα. Η συσκευή, παρέχει ταυτόχρονα και δευτερεύουσες λειτουργίες οι οποίες σχετίζονται με την συμπεριφορά της, με όρια (quota) που μπορούν να τεθούν στις διεργασίες, και με καταγραφή στατιστικών για την χρήση του μηχανισμού.

Τέλος, το backend, το οποίο έχει ενσωματωθεί στον KVM επόπτη, αναλαμβάνει τελικά τα αιτήματα και τοποθετεί δεδομένα σε ένα pool στον πυρήνα του host λειτουργικού.

Παράλληλα, τροποποιήθηκε το frontswap υποσύστημα του linux, ώστε να χρησιμοποιεί τον ίδιο μηχανισμό και backend με την utmem συσκευή, ώστε να είναι εφικτή η αξιοποίηση του νέου backend, καθώς και η εκτίμηση της αξίας ολόκληρου του μηχανισμού της utmem.

2.3 Συμπεράσματα - Στόχος της εργασίας

Πλέον είναι σαφές ότι η utmem είναι κατάλληλη λύση για περιπτώσεις virtualization όπου υπάρχει πίεση μνήμης, ή έχει δοθεί ελάχιστη στο guest σύστημα. Η εκτέλεση υπηρεσιών ως unikernels, είναι ένα συνηθισμένο παράδειγμα ελάχιστης διαθέσιμης μνήμης, όπου θα ταίριαζε ένας μηχανισμός ανταλλαγής μνήμης με τον host, όπως η utmem. Σκοπός αυτής της εργασίας είναι να ενσωματώσουμε τον μηχανισμό utmem στο Rumprun unikernel framework. Επίσης, όπως θα φανεί και στην συνέχεια, εκτιμάται η χρηστική αξία και οι επιδόσεις του συνδυασμού των προαναφερθέντων τεχνολογιών.



Σχήμα 2.7: Δομή του utmem μηχανισμού

Κεφάλαιο 3

Σχεδιασμός και υλοποίηση

Προς αποφυγή παρεξηγήσεων ή παρερμηνειών, εις το εξής θα αναφερόμαστε στον μηχανισμό utmem που παρουσιάστηκε στο προηγούμενο κεφάλαιο ως αυθεντικό μηχανισμό ή αυθεντική utmem. Στην έκδοση που υλοποιήσαμε και παρουσιάζουμε στην παρούσα εργασία θα αναφερόμαστε ως unikernel utmem ή απλά unikernel έκδοση.

Η αυθεντική υλοποίηση του μηχανισμού utmem διαχωρίζει τον μηχανισμό σε τρία αρθρωτά μέρη. Την συσκευή (device) /dev/utmem, τον χρήστη utmem και το πίσω μέρος (backend)[29]. Από αυτά, τα δύο πρώτα βρίσκονται στο περιβάλλον του guest και το τρίτο στο περιβάλλον του host. Επειδή τα μέρη αυτά είναι εκ κατασκευής σε αρκετά υψηλό βαθμό ανεξάρτητα μεταξύ τους, και επειδή ο host θα παραμένει ένα παραδοσιακό POSIX λειτουργικό σύστημα (linux) διατηρούμε αυτούσιο το πίσω μέρος ως έχει, ενώ επανασχεδιάζουμε τα άλλα δύο μέρη στο unikernel περιβάλλον. Επίσης, αγνοούμε μερικά άλλα τοπικά backend, που προσφέρονται στο repository της αυθεντικής utmem καθώς χρησιμεύουν μόνο ως δοκιμές και δεν συνεργάζονται με τον host. Ο επόπτης (hypervisor) παραμένει το KVM όπως και στην αυθεντική υλοποίηση.

Θυμίζουμε πως η συσκευή utmem εκτός των βασικών λειτουργιών, προσφέρει μέσω της ioctl κλήσης μια πληθώρα βοηθητικών λειτουργιών οι οποίες έχουν να κάνουν με δευτερεύουσες επιλογές και αξιοποίηση μετρητικών ενδείξεων. Επειδή μας ενδιαφέρει η περίπτωση της επικοινωνίας host και guest, η υλοποίηση για unikernel περιβάλλον υποστηρίζει μόνο τις βασικές, αλλά συνάμα απαραίτητες, λειτουργίες επάνω στην tmem (Get, Put, Invalidate) και συνεπώς επιλέγουμε να απουσιάζουν οι προαναφερθείσες δευτερεύουσες λειτουργίες.

Η ενσωμάτωση του μηχανισμού utmem στο Rumprun έγινε σε δύο φάσεις, με δυο διαφορετικές μορφές. Αμφότερες αναλύονται στην συνέχεια, ενώ και οι δύο μπορούν αξιόπιστα να επιτελέσουν τις τρεις βασικές λειτουργίες utmem που μας ενδιαφέρουν.

Επίσης, εις το εξής θα αναφερόμαστε στα user space και kernel space με εισαγωγικά («user space» - «kernel space»), θέλοντας να δείξουμε την αντιστοιχία, ως προς

το επίπεδο της στοίβας λογισμικού του Rumprun, με τους αντίστοιχους χώρους που υπάρχουν στα συμβατικά λειτουργικά.

Τέλος, αναφέρουμε πως η μεταφορά δεδομένων στην utmem γίνεται χρησιμοποιώντας το struct tmem_request . Αυτό πρόκειται για ένα datatype, με το οποίο μπορούμε να αντιγράψουμε μεταξύ host και guest μνήμη (value) αυθαίρετου μεγέθους, η οποία αναφέρεται με κλειδί (key) επίσης αυθαίρετου μεγέθους. Για να έχουμε συμβατότητα με τον αυθεντικό μηχανισμό διατηρούμε και εμείς τα ίδια semantics και data types.

```
1  struct tmem_request {
2    union {
3      struct tmem_put_request put;
4      struct tmem_get_request get;
5      struct tmem_invalidate_request inval;
6    };
7    unsigned long flags;
8  };
9
10 struct tmem_put_request {
11     void *key;
12     size_t key_len;
13     void *value;
14     size_t value_len;
15 };
16
17 struct tmem_get_request {
18     void *key;
19     size_t key_len;
20     void *value;
21     size_t *value_lenp;
22 };
23
24 struct tmem_invalidate_request {
25     void *key;
26     size_t key_len;
27 };
```

Σχήμα 3.1: Το struct tmem_request καθώς και δευτερεύοντα datatypes που χρησιμοποιούνται για την μεταφορά δεδομένων

3.1 Πρώτη φάση - system call

Στο Rumprun απουσιάζει ο διαχωρισμός των χώρων χρήστη και πυρήνα , ωστόσο κατά την αρχική υλοποίηση διατηρούμε αυτόν τον , τουλάχιστον δομικό, διαχωρισμό ώστε η υλοποίηση μας να μοιάζει περισσότερο με την αυθεντική υλοποίηση της utmem σε linux. Προς αποφυγή οποιασδήποτε παρεξήγησης, δεν εννοούμε πως δημιουργούμε ξεχωριστό χώρο διευθύνσεων, αλλά πως εισάγουμε την utmem χαμηλά στην στοίβα λογισμικού, εκεί που παραδοσιακά βρίσκεται ο χώρο του πυρήνα. Συνεπώς, με βάση αυτήν μας την επιλογή μια «user space» εφαρμογή στο Rumprun, εφόσον επιθυμεί να χρησιμοποιήσει τις utmem λειτουργίες, οφείλει να ζητήσει από τον πυρήνα, δηλαδή από τα rump kernels, να εκτελέσει τις λειτουργίες αυτές εκ μέρους της, θεωρώντας πως δεν έχει το δικαίωμα να την επιτελέσει η ίδια στο «user space».

Στην αυθεντική υλοποίηση του utmem μηχανισμού προστέθηκε μια virtual device στο λειτουργικό, την οποία οι εφαρμογές χρησιμοποιούσαν μέσω ioctl κλήσεις συστήματος. Στην Rumprun υλοποίηση δημιουργήσαμε απλά μια νέα κλήση συστήματος (system call) του NetBSD , την οποία ονομάσαμε tmem με αριθμό 483, η οποία υποστηρίζει τις τρεις βασικές λειτουργίες (operations) της utmem με βάση το KVM, δηλαδή Put, Get, Invalidate. Αυτή η κλήση συστήματος αντικαθιστά την χρήση της ioctl από την εφαρμογή. Εν τέλει, η κλήση συστήματος θα μεταφραστεί σε κλήση συνάρτησης από τα rump kernels, όταν δημιουργηθεί το τελικό unikernel. Για να γίνει αυτό, φροντίσαμε να ενημερώσουμε τα rump kernels για την ύπαρξη αυτής της νέας κλήσης συστήματος, έτσι ώστε κατά το build του toolchain να συμπεριληφθεί και στην συνέχεια να είναι διαθέσιμη κατά το bake των unikernel.

Επιλέξαμε την εισαγωγή της νέας κλήσης συστήματος καθώς, από πλευράς έκτασης κώδικα και χρήσης από μια «user space» εφαρμογή, είναι απλούστερο σε σχέση με το να εισάγουμε μια εικονική συσκευή όπως γίνεται στην αυθεντική utmem. Μην ξεχνάμε πως βασική φιλοσοφία των unikernels είναι η ελαχιστοποίηση του αποτυπώματος των εφαρμογών που εκτελούνται. Άλλωστε, δεν αποτελεί την τελική μορφή όπως θα φανεί και στην συνέχεια.

Η ροή των ενεργειών που θα ακολουθηθεί όταν μια εφαρμογή-unikernel επιθυμεί να αποθηκεύσει στον host ένα κομμάτι της μνήμης του στον guest, μέσω του operation Put, είναι η εξής:

1. Η εφαρμογή γεμίζει μια περιοχή μνήμης της με δεδομένα. Αυτά τα δεδομένα σκοπεύει να τα δώσει στον host, ώστε να μπορεί στη συνέχεια να αποδεσμεύσει αυτήν την περιοχή μνήμης.

2. Εκτελεί την κλήση συστήματος tmem, όπου δηλώνει την διεύθυνση μνήμης της περιοχής, το όνομα του κλειδιού με το οποίο θα αναφερόμαστε σε αυτή, και το μέγεθος της. Η ροή περνά στο «kernel space».

3. Ο πυρήνας αντιγράφει τα δεδομένα σε δικό του χώρο, ελέγχει πως όλα είναι

σωστά, και εκτελεί το KVM hypercall που αφορά την tmem και συγκεκριμένα το Put operation.

4. Το backend λαμβάνει το hypercall request και αναλαμβάνει να αποθηκεύσει την μνήμη που ζητήθηκε. Η συμπεριφορά του είναι ίδια με την περίπτωση που είχαμε virtualized guest ως κανονικό λειτουργικό. Το backend δεν είναι σε θέση να γνωρίζει το είδος της εικονικής μηχανής που έστειλε το αίτημα.

5. Εφόσον όλα πήγαν καλά, επιστρέφεται μήνυμα επιτυχίας στον guest μέχρι και το «user space» επίπεδο. Πλέον η εφαρμογή μπορεί να αποδεσμεύσει την μνήμη.

Η ροή για τα Put και Invalidate operation είναι αντίστοιχη με τα παραπάνω.

3.1.1 Τεχνικές δυσκολίες

Όπως ήταν αναμενόμενο, για να υλοποιηθεί το παραπάνω εμφανίστηκαν μερικές τεχνικές δυσκολίες.

Η πρώτη τεχνική δυσκολία στην υλοποίηση του μηχανισμού επικοινωνίας με τον KVM hypervisor, και άρα και με τις paravirtualization ευκολίες που προσφέρει, είναι πως στο NetBSD δεν υπάρχει έτοιμος μηχανισμός (API) ώστε να εκτελούνται hypercalls με βάση το KVM ως hypervisor. Αντίθετα στο linux αυτό το task είναι πιο απλό. Εκεί υπάρχουν έτοιμες συναρτήσεις του χώρου πυρήνα[5], ονόματι kvm_hypercallX, όπου X+1 ο αριθμός των argument που πρέπει να γνωρίζει ο host για την εξυπηρέτηση της hypercall. Ο tmem μηχανισμός για hypercall χρειάζεται τρια arguments, τον κωδικό της tmem, τον κωδικό του tmem operation, και το tmem request structure το οποίο εκθέτουμε στον host ως ανταλλακτήριο πληροφοριών.

Η λύση στο πρόβλημα αυτό είναι να εξάγουμε από το source tree του linux τις απαραίτητες γραμμές και να τις προσθέσουμε στην κλήση συστήματός μας. Αυτό αυτομάτως περιορίζει την υλοποίησή μας σε αρχιτεκτονική x86, η οποία, όμως, είναι η πλέον δημοφιλής και άρα δεν αποτελεί μεγάλο μειονέκτημα. Ακολουθήσαμε λοιπόν την ροή της kvm_hypercall2 στο linux, και καταλήγουμε σε περίπου 60 γραμμές x86 assembly. Το πιο σημαντικό εντός αυτών είναι πως εκτελείται ένα X86_FEATURE_VMMCALL με νούμερο (8*32+15) το οποίο όπως αναμενόταν αφορά το KVM hypervisor στην x86 αρχιτεκτονική. Παίρνοντας αυτές τις γραμμές από τον πηγαίο κώδικα του linux και τοποθετώντας τες σε ένα header file, η system call που υλοποιούμε είναι σε θέση να καλεί KVM hypercalls ακόμα και ως Rumprun unikernel.

Το επόμενο βήμα είναι να καλέσουμε KVM hypercall για χρήση συγκεκριμενα της utmem. Τα operation codes και type code της utmem είναι γνωστά από την αυθεντική υλοποίηση και μπορούν να μεταφερθούν αυτούσια.

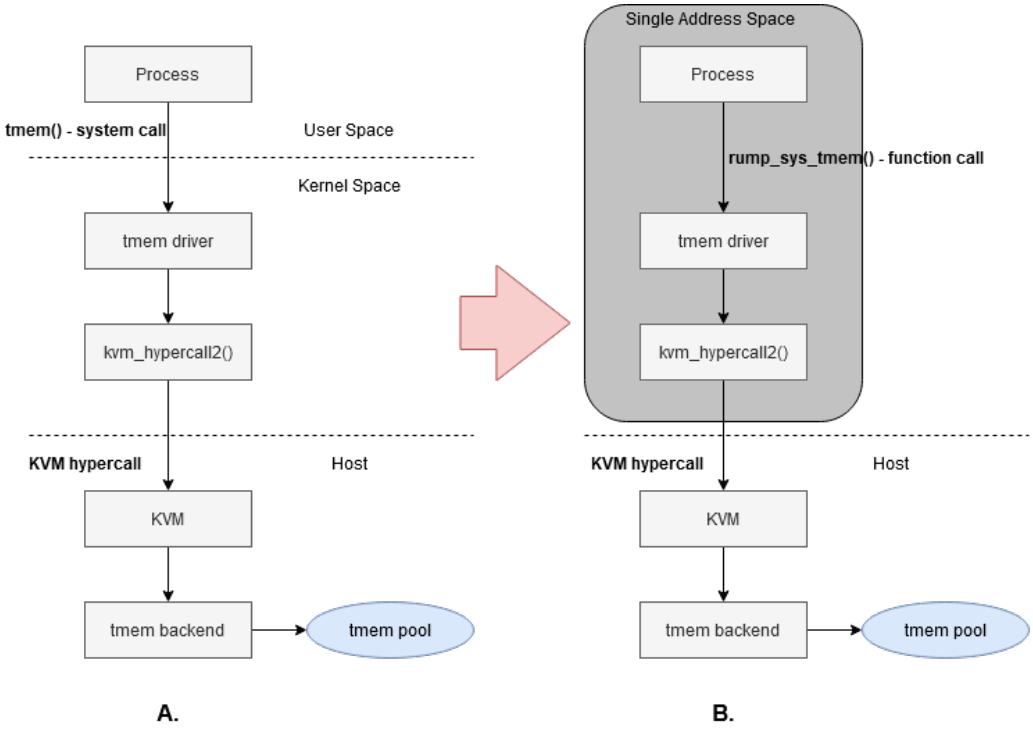
Εδώ εμφανίζεται και η δεύτερη τεχνική δυσκολία, η οποία είναι η έκθεση (expose) των πληροφοριών μας στον host που αναλαμβάνει την εξυπηρέτηση των hypercall μας. Ο μηχανισμός utmem χρησιμοποιεί instances του tmem_request struct ώστε να μεταφέρεται η πληροφορία από τον host στον guest ή ανάποδα. Το συγκεκριμένο struct φέρει την τιμή του ζευγαριού key-value καθώς και βοηθητικές πληροφορίες που έχουν να κάνουν με το μέγεθος της εκάστοτε τιμής. Δυστυχώς ο host και ο guest «ζουν» σε διαφορετικούς χώρους μνήμης (memory space) και χρειάζεται μετάφραση σε φυσικό χώρο διευθύνσεων (physical address space), τον οποίο ο host είναι σε θέση να προσπελάσει. Η αυθεντική υλοποίηση της utmem μεταφράζει page-aligned virtual addresses σε physical address. Η ευθυγράμμιση σε επίπεδο σελίδας (page alignment) δεν είναι απαραίτητη και φαίνεται πως είναι κατάλοιπο από τα αρχικά στάδια της υλοποίησης του μηχανισμού της utmem, οπότε επιλέξαμε να την αγνοήσουμε στην δική μας υλοποίηση. Τονίζουμε πως αυτό δεν επηρεάζει την ορθή επικοινωνία μεταξύ των δύο πλευρών. Για να επιτύχουμε την μετάφραση από virtual address σε physical address χρησιμοποιούμε την συνάρτηση vtophys() που προσφέρει το NetBSD σε χώρο πυρήνα, ενώ παράλληλα την υποστηρίζουν και τα rump kernels.

Υλοποιήσαμε συνεπώς ό,τι χρειάζεται η system call μας και πλέον έχουμε ένα μηχανισμό αντίστοιχο με της αυθεντικής υλοποίησης της utmem, ο οποίος μπορεί να χρησιμοποιηθεί σε unikernel περιβάλλον. Όλος ο κώδικας δεν ξεπερνά τις 450 γραμμές, αρκετές από τις οποίες είναι αντίγραφα κώδικα του πυρήνα του Linux, και άρα δεν έχουν αναπτυχθεί από το μηδέν.

3.2 Δεύτερη φάση - function call

Όπως αναφέραμε, στο Rumprun υπάρχει μόνο ένας χώρος διευθύνσεων καθώς εφαρμογή και πυρήνας γίνονται bake σε ένα σώμα, το unikernel. Ως εκ τούτου, η αρχική μας προσέγγιση διαχωρισμού είναι κενή σημασίας, απλά βοηθά στην καλύτερη κατανόηση της ροής αφού θυμίζει περισσότερο παραδοσιακό περιβάλλον. Θα ήταν σαφώς καλύτερο να εκμεταλλευτούμε την χαρακτηριστική αυτή ιδιότητα ώστε η υλοποίησή μας να έχει ελάχιστο αποτύπωμα στο περιβάλλον ανάπτυξης.

Με μια προσεχτική ματία στον κώδικα, η μόνη εξάρτηση από συνάρτηση του πυρήνα, αντίστοιχη της οποίας δεν υπάρχει σε χώρο χρήστη, είναι για την μετάφραση από virtual σε physical addresses μέσω της συνάρτησης vtophys(). Η υλοποίηση αυτής στα rump kernels είναι εξαιρετικά απλή, το οποίο μας επιτρέπει να την αντιγράψουμε σε «user space» επίπεδο, δηλαδή στα ανώτερα στρώματα της στοίβας του λογισμικού. Παίρνουμε λοιπόν ό,τι χρειάζεται και για την εκτέλεση του hypercall και πλέον μπορούμε να έχουμε όλη τη λειτουργικότητα της utmem δίχως την ανάγκη μιας system call. Δημιουργήσαμε μια βιβλιοθήκη, έτοιμη να συμπεριληφθεί στον κώδικα από οποιοδήποτε πρόγραμμα το επιθυμεί. Ό,τι κάναμε πριν εντός της system call το κάνουμε τώρα σε ένα «user space» .c file, το οποίο ως σχεδίαση είναι σαφώς απλούστερο από πριν. Αυτή η βιβλιοθήκη αποτελεί ουσαστικά τον driver της utmem για

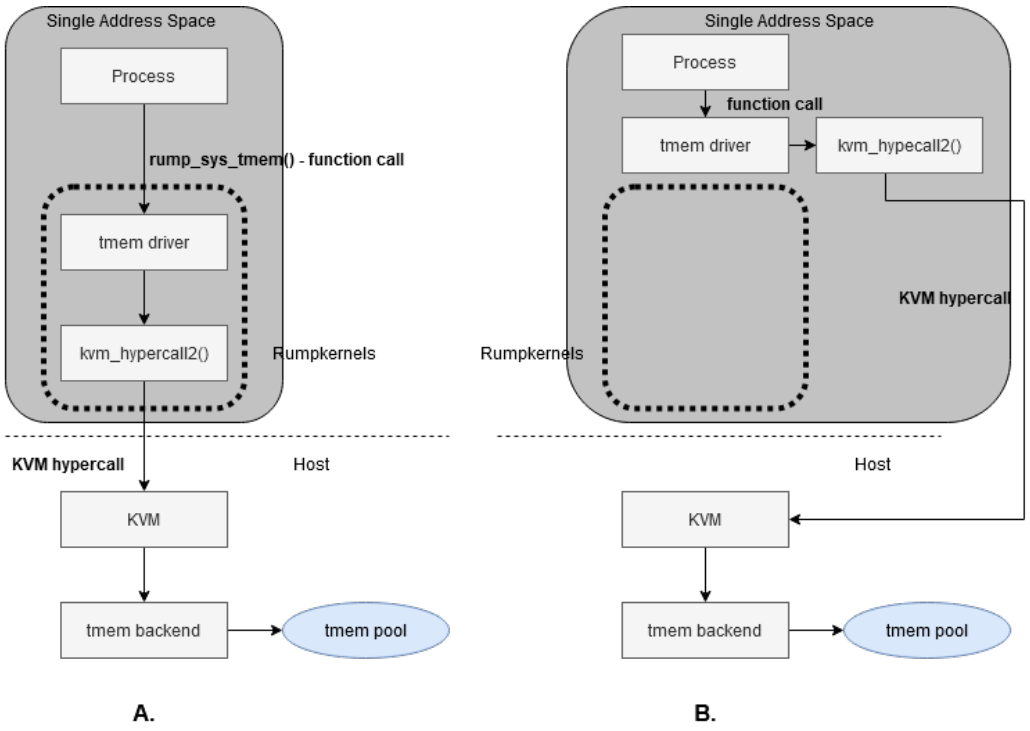


Σχήμα 3.2: A.Ροή εκτέλεσης της system call tmem() στο NetBSD B. Η μετατροπή της σε function call από τα rump kernels

το Rumprun.

Από την μια, η ανάπτυξη εφαρμογών γίνεται ελαφρώς πιο σύνθετη, καθώς πρέπει να προστεθεί κατά το linking αυτών και ο κώδικας που εκτελεί τα hypercall και προσφέρει τα utmem operations, σε αντίθεση με την system call όπου η libc αναλαμβάνει την αντιστοιχία. Από την άλλη, όμως, το κέρδος είναι πως πλέον δεν απαιτούνται αλλαγές στο Rumprun-rump kernels. Η τελική μορφή των δύο εκδόσεων διαφέρει ελάχιστα, καθώς η system call tmem της πρώτης φάσης εν τέλει μεταφράζεται σε ένα απλό function call από το Rumprun, με αποτέλεσμα το overhead να είναι ίδιο στις δύο περιπτώσεις. Μια μικρή διαφορά είναι πως η function call απαιτεί μια αντιγραφή λιγότερη των δεδομένων.

Συντονιζόμαστε, συνεπώς, με την φιλοσοφία πίσω από τα rump kernels, δηλαδή την ανάπτυξη και εκτέλεση drivers από τον χώρο χρήστη, δίχως να απαιτείται η προσαρμογή του υφιστάμενου πυρήνα. Ο driver μας είναι όλος ο κώδικας που αναλαμβάνει να εκτελέσει την κλήση επόπτη, και να ανταλλάξει με ασφάλεια τα κομμάτια μνήμης μεταξύ του guest και του host.



Σχήμα 3.3: A.Utmem στο Rumprun ως system call B. Utmem στο Rumprun ως function call

Κεφάλαιο 4

Αξιολόγηση μηχανισμού

4.1 Ανάλυση καθυστέρησης αιτημάτων της unikernel utmem

Θέλοντας να εκτιμήσουμε την επίδοση του μηχανισμού utmem ως Unikernel, θεωρούμε χρήσιμο να μετρήσουμε την χρονική καθυστέρηση εντός του driver, που αναπτύξαμε ,μεταξύ των δυο σημαντικών operations δηλαδή του Put και του Get, καθώς και να αναλύσουμε την ανά στάδιο χρονική διάρκεια. Με απλά λόγια, θέλουμε να δούμε πόσο διαρκεί η εξυπηρέτηση ενός αιτήματος που αποστέλνει η εφαρμογή, η οποία χρησιμοποιεί την utmem.

Επιλέγουμε να μελετήσουμε την έκδοση με το function call, καθώς είναι σχετικά απλούστερη, ενώ τα αποτελέσματα δεν επηρεάζονται από το NetBSD ή τα rump kernels. Την έκδοση αυτή επιλέγουμε και στα επόμενα στάδια της εργασίας, όπως θα φανεί και στην συνέχεια.

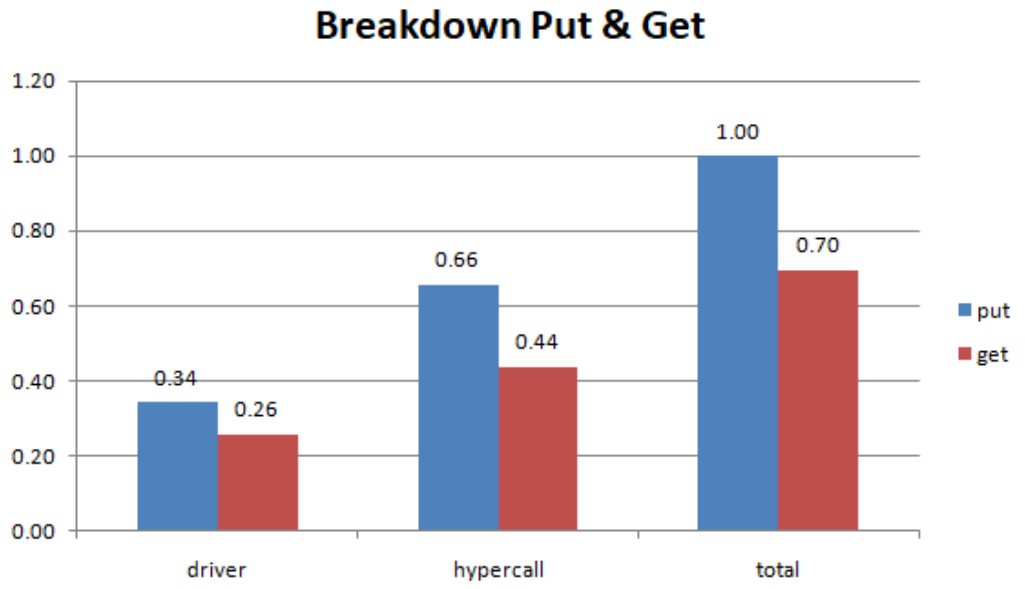
Διαιρούμε την διαδικασία εξυπηρέτησης του αιτήματος σε δύο στάδια. Τον χρόνο που η ροή του κώδικα βρίσκεται εντός του hypercall, δηλαδή τον χρόνο κατά τον οποίο αναλαμβάνει το backend και γενικά ο επόπτης, και τον χρόνο που η ροή βρίσκεται εντός του driver, αλλά μόνο στις γραμμές που προστέθηκαν από εμάς. Το πρώτο μέγεθος το ονομάζουμε hypercall (time), ενώ το δεύτερο driver (time). Ταυτόχρονα, υπολογίζουμε και τον συνολικό (total) χρόνο που χρειάζεται η εξυπηρέτηση του αιτήματος, ο οποίος προφανώς είναι το άθροισμα των δύο προηγούμενων. Άρα κάνουμε το λεγόμενο breakdown της καθυστέρησης.

Για να πάρουμε τις μετρήσεις, αναπτύξαμε ένα unikernel, το οποίο αποστέλνει αίτημα Get και Put επάνω στο ίδιο κλειδί και στο ίδιο value. Το μέγεθος του value επιλέχθηκε να είναι 1 megabyte, καθώς αυτό είναι το ίδιο μέγεθος κατά την αντίστοιχη διαδικασία η οποία περιγράφεται στην εργασία της αυθεντικής utmem[19]. Ταυτόχρονα, προστέθηκαν γραμμές εντός του function call, οι οποίες καταγράφουν τον χρόνο

του συστήματος, ώστε να είναι δυνατός ο υπολογισμός της καθυστέρησης ανά στάδιο. Το συγκεκριμένο unikernel, λοιπόν, αποστέλνει το ίδιο αίτημα Put και Get, για προκαθορισμένο αριθμό φορών, και καταγράφει τις καθυστερήσεις κάθε φοράς. Τέλος, υπολογίζει και μας τυπώνει τον μέσο όρο αυτών των μετρήσεων.

Οι μετρήσεις έγιναν μέσα σε εικονική μηχανή QEMU/KVM με Ubuntu Linux 4.9.39, στον πυρήνα του οποίου έχουν προστεθεί τα patch της αυθεντικής utmem για το back-end. Ο επεξεργαστής είναι intel core i5-8250u από τον οποίο αναθέτουμε 2 φυσικούς πυρήνες (4 λογικούς), ενώ η διαθέσιμη μνήμη RAM του VM είναι 4GB.

Τα αποτελέσματα δείχνουν πως για κάθε request απαιτείται κατα μέσο όρο συνολικός χρόνος της τάξης εκατοντάδων microsecond (μs). Συγκεκριμένα 178 μs για την Put και 124 μs για την Get. Αυτή η ασυμμετρία στους χρόνους των δύο αιτημάτων συμπίπτει με την ασυμμετρία που παρατηρήθηκε και στην αυθεντική utmem. Ταυτόχρονα παρατηρούμε πως η αναλογία του σταδίου driver προς hypercall είναι της τάξης του 65-75 %. Τα αποτελέσματα φαίνονται αναλυτικά στον πίνακα 4.1. Στο σχήμα 4.1 φαίνεται η σχέση ανάμεσα στα δύο αιτήματα σύμφωνα με τις παραπάνω μετρήσεις.



Σχήμα 4.1: Χρονική ανάλυση των σταδίων των αιτημάτων utmem Put και Get κανονικοποιημένη ως προς τον συνολικό χρόνο του Put

	Put	Get
driver	61	46
hypercall	117	78
total	178	124

Πίνακας 4.1: Αποτελέσματα ανάλυσης καθυστέρησης αιτημάτων Put και Get. Οι τιμές είναι σε μs.

4.2 Σύγκριση με υπάρχουσες unikernel λύσεις

Επιθυμούμε επίσης να εκτιμήσουμε την αξία της χρήση της utmem στο Rumprun, δηλαδή αν και σε ποιές περιπτώσεις αξίζει ένας προγραμματιστής να στραφεί στην utmem. Για αυτόν τον σκοπό χρησιμοποιούμε ως καταλληλότερο το μετροπρόγραμμα (benchmark) Redis (Remote Dictionary Server). Το Redis είναι ιδανικό για αυτόν το σκοπο για δύο λόγους. Πρώτον πρόκειται, όπως λέει και το ίδιο [4], για ένα data structure store, δηλαδή ένα εξυπηρετητή αποθήκευσης αφηρημένων δεδομένων, όπου οι βασικές λειτουργίες είναι get και set με αναφορά σε κλειδιά, και άρα υπάρχει προφανής ομοιότητα με τα utmem operations. Δεύτερον, υπάρχει ήδη έκδοση unikernel του Redis από τα προσφερόμενα Rumprun-packages στο επίσημο repository, οπότε δεν χρειάζεται να το μετατρέψουμε εμείς σε unikernel[9].

Η γενική δομή του Redis μοντέλου, είναι η ύπαρξη ενός back-end server, ο οποίος είναι υπεύθυνος για την αποθήκευση αφηρημένης μορφής δεδομένων, και την εξυπηρέτηση αιτημάτων από τρίτες εφαρμογές επάνω σε αυτά τα δεδομένα. Ταυτόχρονα, υπάρχουν διάφοροι front-end clients, οι οποίοι αποστέλνουν αιτήματα στον server με το TCP/IP πρωτόκολλο. Ως unikernel προσφέρεται μόνο ο Redis server, και ο client μπορεί να είναι οποιαδήποτε εφαρμογή ικανή να διαχειρίζεται TCP αιτήματα, π.χ. nc.

Αποφασίζουμε να προσθέσουμε τρεις νέες εντολές στο Redis, τις tmemPut, tmemGet, tmemInval, οι οποίες όπως δείχνει το όνομά τους αντιστοιχούν στα τρία utmem operation που υλοποιήσαμε. Στα εξής θα αναφερόμαστε σε αυτές τις εντολές ως utmem commands, για να τις ξεχωρίζουμε από τα utmem operations. Ο προτεινόμενος τρόπος εισαγωγής εντολών στο Redis είναι η δημιουργία ενός Redis module το οποίο προσθέτει τις εντολές κατά το runtime του Redis server. Όμως η έκδοση του Redis, που υπάρχει ως unikernel, είναι σχετικά παλιά και δεν υποστηρίζει τα modules, οπότε προσθέσαμε τις εντολές απευθείας στον πηγαίο κώδικα του Redis server.

Επίσης επιλέγουμε να χρησιμοποιηθεί η έκδοση utmem μέσω function call, και όχι system call. Αυτό εξαλείφει την πιθανότητα τα αποτελέσματά μας να επηρεάζονται από τις αλλαγές στα rump kernels, μιας και τα χρησιμοποιούμε χωρίς αλλαγή. Πρέπει μόνο να ενσωματώσουμε την μεταγλώττιση του driver της utmem στο Redis, το οποίο σημαίνει να προστεθούν flags για assembly compilation.

Οι μετρήσεις έγιναν εντός της ίδια εικονικής μηχανής που χρησιμοποιήθηκε στο προηγούμενο κεφάλαιο.

Τέλος, οφείλουμε να διευκρινίσουμε πως εκτελούμε το Redis χωρίς AOF persistence. Το Redis προσφέρει δυνατότητα διατήρησης των δεδομένων της μνήμης τους στο δίσκο ώστε αυτά να μπορούν να ανακτηθούν σε περίπτωση μη προβλεπόμενου τερματισμού του Redis. Διατηρεί, συνεπώς ένα αρχείο με αυτές τις εγγραφές στον δίσκο. Το μειονέκτημα, όπως είναι πως κάθε προσθετοαφαίρεση δεδομένου, π.χ. μια set, μπλοκάρει μέχρι να γραφούν τα δεδομένα στο δίσκο[10]. Ως γνωστόν, οι δίσκοι είναι τάξεις μεγέθους πιο αργοί από την κύρια μνήμη, συνεπώς το να διατηρούσαμε το AOF persistance του Redis θα έδινε ένα άδικο πλεονέκτημα στα utmem commands μας, τα οποία δεν διατηρούν τέτοια αντίγραφα. Συγκρίνουμε, λοιπόν μόνο in-memory ενέργειες του Redis.

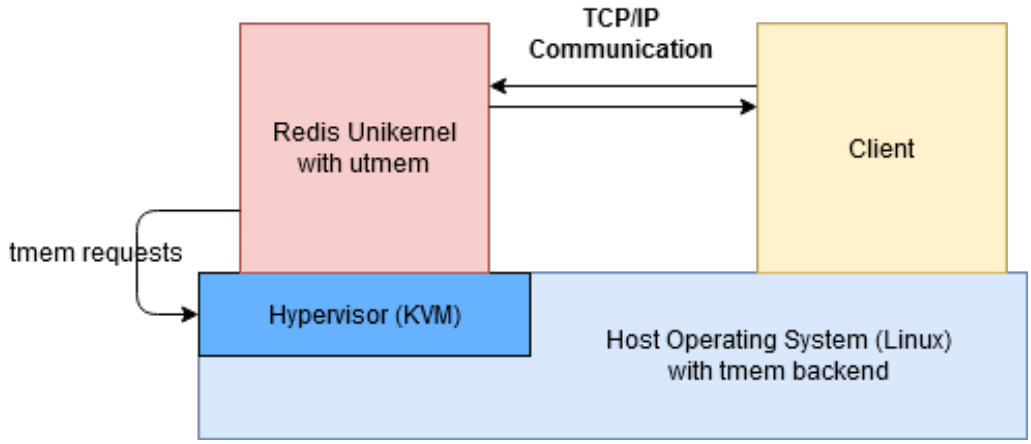
4.2.1 Σενάριο 1: με άφθονη διαθέσιμη μνήμη

Για μετρήσουμε την συμπεριφορά του Redis με τα νέα commands, δημιουργήσαμε ένα custom μετροπρόγραμμα, το οποίο ονομάζουμε client. Το πρόγραμμα αποστέλλει μέσω TCP/IP network requests, προς το Redis. Ταυτόχρονα, το client καταγράφει δεδομένα για την επίδοση της επικοινωνίας με τον server. Ο client τρέχει στον host ως απλή διεργασία ώστε να μην επηρεάζεται η εκτέλεση του Redis, το οποίο με την σειρά του τρέχει ως unikernel. Θέλουμε να συγκρίνουμε δύο commands, το απλό set και το tmemPut, οπότε για διάφορους συνδυασμούς μεγέθους value και αριθμό requests λαμβάνουμε μετρήσεις για το πόσο χρόνο χρειάστηκε να ικανοποιηθούν όλα τα requests, και στην συνέχεια παίρνουμε ένα μέσο όρο. Συνεπώς, η μετρική που μας ενδιαφέρει είναι τα commands/δευτερόλεπτο. Λογίζουμε ως παράγοντα και τον αριθμό των request, ώστε να ελέξουμε αν και κατά πόσον το συνολικό μέγεθος των αποθηκευμένων δεδομένων επηρεάζει τον ρυθμό εξυπηρέτησης.

Αναλυτικότερα, επιλέγουμε ένα μέγεθος τιμής (value size) και ένα αριθμό από τιμές (number of values) που θα καταχωρήσουμε στο Redis. Για παράδειγμα ο συνδυασμός 512-1024 σημαίνει πως θα καταχωρηθούν 1024 τιμές, η κάθε μια από τις οποίες αντιστοιχεί σε ένα ξεχωριστό command, όπου κάθε τιμή θα έχει μέγεθος 512 bytes. Αυτή η διαδικασία γίνεται για πολλούς συνδυασμούς value size και number of values. Προφανώς, ανάμεσα σε κάθε επανάληψη του πειράματος φροντίζουμε να αδειάζουμε την μνήμη του Redis ή το tmem pool, για το set ή το tmemPut αντίστοιχα.

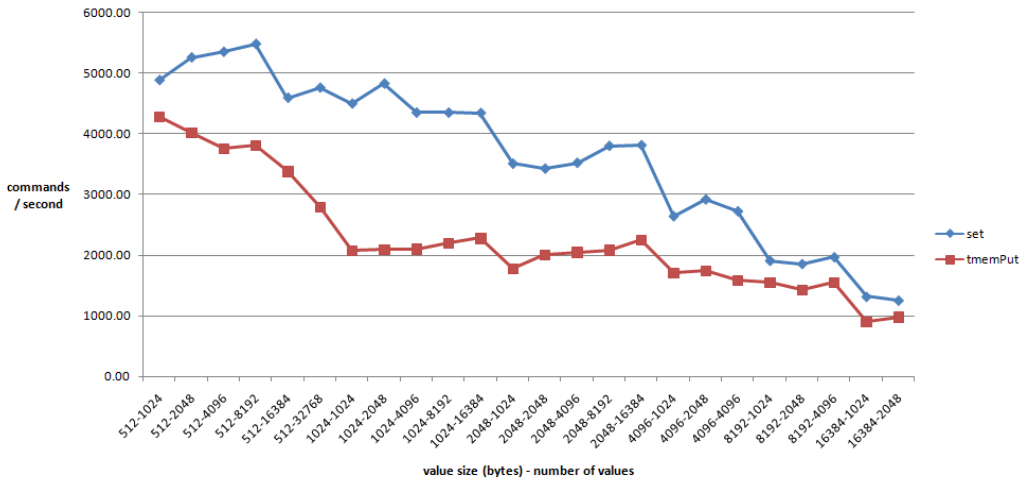
Η δομή του συστήματος client και Redis unikernel φαίνεται στην εικόνα 4.2.

Από τα αποτελέσματα των μετρήσεων παρατηρείται πως, όταν υπάρχει αρκετή μνήμη τα utmem commands είναι σχετικά πιο αργά από τις set εντολές του Redis. Αυτό είναι λογικό, καθώς πρώτον πρέπει για κάθε tmem command να εκτελείται

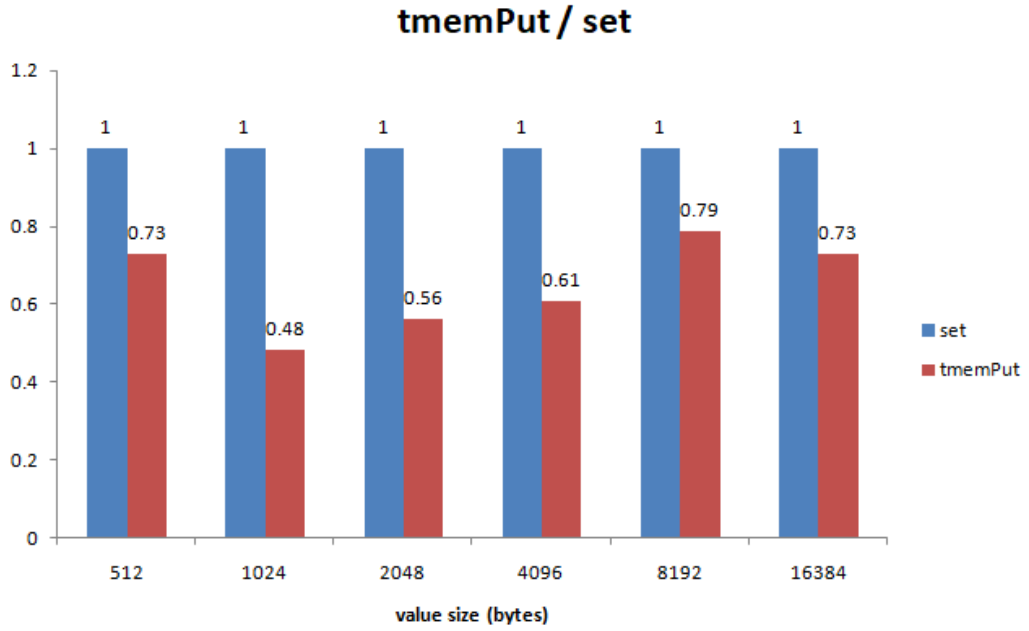


Σχήμα 4.2: Επικοινωνία και δομή client - Redis unikernel

ένα hypercall, το οποίο είναι σύγχρονος μηχανισμός και άρα οδηγεί σε μπλοκάρισμα του unikernel. Δεύτερον, πρέπει να αντιγράφεται το σύνολο των δεδομένων από τον χώρο του unikernel στον χώρο πυρήνα του host, αυξάνοντας τις αντιγραφές που εκτελούνται ανά command. Στην εικόνα 4.3 φαίνεται η σχετική συμπεριφορά των δύο ειδών commands για διάφορους συνδυασμούς μεγέθους value και αριθμού requests, και πως πράγματι η tmemPut είναι σχετικά πιο αργή από την set μετρώντας commands/second. Στην εικόνα 4.4 παρουσιάζεται η ποσοστιαία διαφορά ταχύτητας των δύο εντολών παίρνοντας μέσο όρο ανά value size, ενώ στον πίνακα 4.2 φαίνονται οι απόλυτες τιμές της παραπάνω μέτρησης



Σχήμα 4.3: Σύγκριση tmemPut και set commands στο unikernel Redis.



Σχήμα 4.4: Ποσοστιαία σχέση tmemPut και set κανονικοποιημένη ως προς set ανά value size

value size	512	1024	2048	4096	8192	16384
set	5062.7	4480.9	3621.1	2768.2	1916.2	1288.5
tmemPut	3676.2	2148.3	2032.1	1678.6	1506.0	938.6

Πίνακας 4.2: Επίδοση μετρημένη σε commands / second για set και tmemPut ανά value size.

4.2.2 Σενάριο 2: με περιορισμένη διαθέσιμη μνήμη

Υποχρεούμαστε να μελετήσουμε την συμπεριφορά του Redis και σε περιπτώσεις όπου ο πόρος της μνήμης είναι σημαντικά περιορισμένος. Τρέχουμε πάλι τον ίδιο Unikerel server, αλλά τώρα περιορίζουμε την διαθέσιμη μνήμη όλης της εικονικής μηχανής σε μερικές δεκάδες megabyte.

Όταν η μνήμη είναι περιορισμένη, τα utmem commands γίνονται η μόνη λειτουργική λύση για αποθήκευση δεδομένων. Θυμίζουμε πως στο Rumprun απουσιάζει η εικονική μνήμη, συνεπώς όταν γεμίσει η διαθέσιμη δεν υπάρχει κάποιος μηχανισμός να απελευθερώσει αυτήν την πίεση μνήμης. Στην περίπτωση του set, μετά από ένα αριθμό από requests, είναι αδύνατον να κατανεμηθεί νέα μνήμη, έστω εικονική, στο Redis, το οποίο πλέον αδυνατεί να ικανοποιήσει τα requests που του έρχονται από τον client. Σε αυτό το σενάριο, εμφανίζει μήνυμα σφάλματος και τερματίζει την λειτουρ-

γία του. Καταστροφική συμπεριφορά για ένα εξωτερικό χρήστη που επικοινωνεί με το unikernel. Αντίθετα, το tmemPut ικανοποιείται κανονικά, καθώς η απαίτηση σε χώρο είναι ελάχιστη, αρκεί δηλαδή να ικανοποιείται ένα request, για να ικανοποιηθούν όλα, μιας και τα δεδομένα αποθηκεύονται στην μνήμη του host. Μάλιστα η συμπεριφορά είναι ταυτόσημη με την περίπτωση αφθονίας μνήμης, δεν παρατηρείται καμία επιπλέον καθυστέρηση λόγω της περιορισμένης μνήμης.

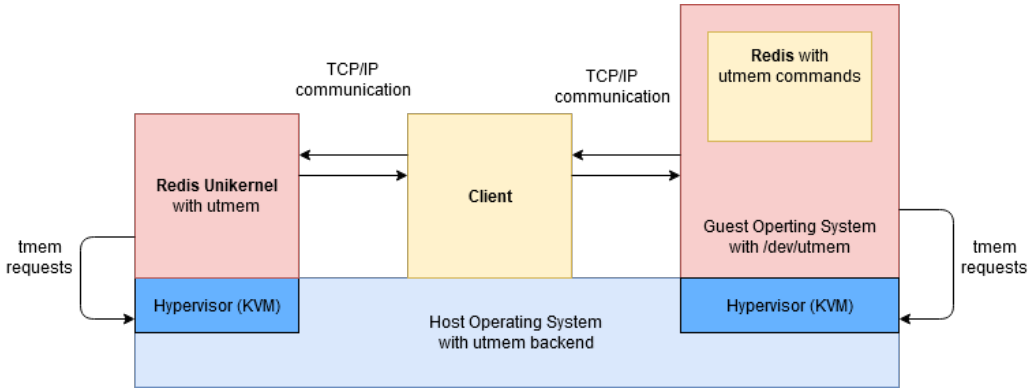
4.3 Σύγκριση με την αυθεντική utmem

Τέλος, για να εξαχθεί μια πληρέστερη εικόνα των επιδόσεων του μηχανισμού, θεωρούμε σκόπιμο, να συγκρίνουμε την utmem υλοποίηση σε unikernel, σε σχέση με τον αυθεντικό (original) μηχανισμό της utmem.

Χρησιμοποιούμε πάλι το μετροπρόγραμμα Redis, για το οποίο υπάρχει Redis-module με το οποίο το Redis τίθεται ικανό να ικανοποιήσει utmem αιτήματα χρησιμοποιώντας τον αυθεντικό μηχανισμό. Αναφερόμαστε εδώ στο Redis ως κανονική διεργασία ενός linux συστήματος, για αυτό και εις το εξής θα αναφερόμαστε στον αυθεντικό μηχανισμό και ως linux utmem. Θυμίζουμε, πως ο αυθεντικός μηχανισμός χρησιμοποιεί μια εικονική συσκευή (/dev/utmem) του linux, ούτως ώστε οι διεργασίες να μπορούν να ζητούν από τον πυρήνα να εκτελέσει tmem αιτήματα προς το backend εκ μέρους αυτών. Επιπροσθέτως, επιθυμούμε ο τρόπος μέτρησης να είναι όσο το δυνατόν όμοιος με τον τρόπο μέτρησης που περιγράφεται στην εργασία της αυθεντικής utmem [19], συνεπώς δημιουργήσαμε ένα διαφορετίκό πρόγραμμα client που αποστέλει αιτήματα. Αυτή τη φορά, δεν γεμίζουμε την tmem pool με δεδομένα, αλλά αποστέλλουμε συνεχώς το ίδιο αίτημα, δηλαδή το ίδιο key, για προκαθορισμένο χρονικό διάστημα και εν τέλει υπολογίζουμε ένα μέσο όρο αιτημάτων. Αποστέλουμε ακριβώς τα ίδια αιτήματα και στο unikernel Redis, ώστε να μπορούμε να έχουμε μια όμοια σύγκριση μεταξύ των δύο εκδόσεων. Ελέγχουμε διάφορες τιμές του μεγέθους της τιμής (value size), ενώ το utmem command είναι τύπου Put. Όπως και προηγουμένως, η μετρική που μας ενδιαφέρει είναι τα commands / second που εξυπηρετούνται.

Η τοπολογία του συστήματος μετρήσεων είναι η εξής: όταν μετράμε την αυθεντική utmem, το Redis εκτελείται ως μια κανονική διεργασία εντός εικονικής μηχανής (guest) με 2 GB RAM και ίδιου πυρήνα με τον host, δηλαδή τον πυρήνα με τα utmem patches. Για την utmem ως unikernel, εικονικοποιούμε ένα unikernel εντός του οποίου εκτελείται το Redis επάνω στον ίδιο host. Ο host παραμένει η ίδια εικονική μηχανή που περιγράφεται και προηγουμένως. Τέλος, ο νέος client εκτελείται επίσης ως απλή διεργασία επάνω στον host. Η διάταξη φαίνεται καλύτερα στο σχήμα 4.5.

Από όσα έχουν παρουσιαστεί στην θεωρία, είναι εμφανές πως υπάρχει μια εγγενής ασυμμετρία στα δυο συστήματα που μελετάμε, καθώς πρόκειται για δύο διαφορετικής σχεδίασης και πολυπλοκότητας εικονικές μηχανές. Η εικονική μηχανή linux μέσα στην οποία εκτελείται η αυθεντική utmem διαθέτει διαφορετικό υποσύστημα δικτύου,



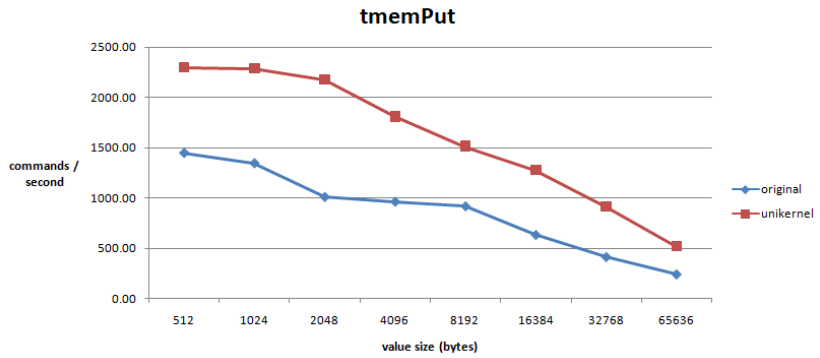
Σχήμα 4.5: Τοπολογία περιβάλλοντος σύγκρισης αυθεντικής utmem και Unikernel utmem.

διαφορετικά modules πυρήνα, ενώ εκτελεί παράλληλα και άλλες διεργασίες εντός της. Αποτελεί, γενικά, μια σύνθετη εικονική μηχανή. Από την άλλη το Rumprun διαθέτει διαφορετικό υποσύστημα δικτύου, καθώς το έχει κληρονομήσει από το NetBSD, και εν γένει είναι πιο απλό. Τέλος, αναμένεται να εμφανιστεί διαφορά στα αποτελέσματα και λόγω του γεγονότος πως η έκδοση unikernel απαιτεί μια αντιγραφή λιγότερη των δεδομένων, κατά την ροή του κώδικα από το υψηλότερο στάδιο μέχρι και το επίπεδο του backend.

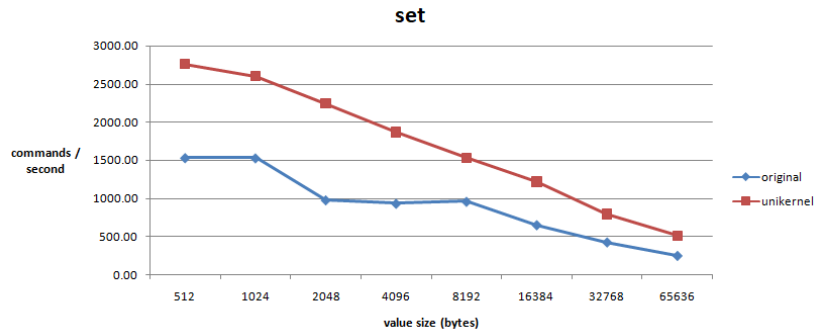
Οφείλουμε συνεπώς να διευρευνήσουμε αν και κατά πόσον η διαφορά των περιβάλλοντων επηρεάζει την τελική εικόνα στις μετρούμενες επιδόσεις ανάμεσα στην αυθεντική και την unikernel έκδοση. Για αυτόν τον λόγο αφού μετρήσαμε τον ρυθμό εξυπηρέτησης των Put requests, επαναλάβαμε το ίδιο πείραμα και για την set εντολή του Redis, της οποίας ο εσωτερικός μηχανισμός εξυπηρέτησης είναι κοινός στις δύο εκδόσεις του Redis. Αν το περιβάλλον εκτέλεσης δεν επηρεάζει τα αποτελέσματα, τότε αναμένουμε να βλέπουμε την ίδια συμπεριφορά της set και στις δύο περιπτώσεις.

Τα αποτελέσματα των μετρήσεων φαίνονται στην εικόνα 4.6. Παρατηρούμε πως τα utmem commands είναι πιο γρήγορα για την unikernel έκδοση σε σχέση με την αυθεντική. Κατά μέσο όρο περίπου δύο φορές πιο γρήγορα. Η ίδια συμπεριφορά, όμως, παρατηρείται και για την set εντολή. Οι απόλυτες τιμές των μετρήσεων παρουσιάζονται στον πίνακα 4.3 για τα Put request και στον πίνακα 4.4 για τα set requests.

Σύμφωνα με τα παραπάνω, οδηγούμαστε στο γενικό συμπέρασμα πως το Rumprun προσφέρει εγγενώς μια επιτάχυνση στα tasks που εκτελούνται. Δεν ξέρουμε, όμως, και πόσο επηρεάζει το τελικό αποτελέσμα η διαφορετική έκδοση utmem. Η αποσαφήνιση της ανώτερης συμπεριφοράς χρίζει περαιτέρω ανάλυσης. Για να επιτευχθεί αυτή, τοποθετούμε εντός της ροής του προηγούμενου πειράματος γραμμές κώδικα, ο οποίες καταγράφουν του χρόνους σε κάθε στάδιο. Ασχολούμαστε μόνο με την Put



(α′) utmem Put



(β′) set

Σχήμα 4.6: Σύγκριση επιδόσεων ανάμεσα σε Rumprun και linux περιβάλλον για utmem Put και set commands.

command της utmem, η οποία μας ενδιέφερε εξ αρχής.

Όσον αφορά την unikernel utmem, ξεχωρίζουμε τους εξής χρόνους:

1. Τον χρόνο κατά τον οποίο ο ροή του request βρίσκεται από το σημείο όπου αποστέλλει ο client το request, μέχρι να αναλάβει η συνάρτηση χειρισμού της εντολής εντός του Redis, τον οποίο ονομάζουμε network (time).

2. Τον χρόνο που η ροή βρίσκεται εντός της συνάρτησης χειρισμού στο Redis, ονόματι redis.

3. Τον χρόνο που βρισκόμαστε εντός του driver utmem που αναπτύξαμε, αλλά όχι εντός του hypecall. Αυτόν τον ονομάζουμε driver χρόνο.

4. Τον χρόνο εντός του hypercall, τον οποίο ονομάζουμε απλά hypercall χρόνο.

Οι δύο τελευταίοι χρόνοι είναι οι αντίστοιχοι που ορίσαμε και στο κεφάλαιο 4.1.

value size	512	1024	2048	4096	8192	16384	32768	65636
orginal	1445.8	1343.8	1009.6	962.4	917.0	633.7	414.8	243.3
unikernel	2297.6	2292.4	2176.1	1808.9	1512.4	1277.2	908.4	516.9

Πίνακας 4.3: Επίδοση της utmem Put μετρημένη σε commands / second για original (linux) και unikernel (Rumprun) περιβάλλον ανά value size.

value size	512	1024	2048	4096	8192	16384	32768	65636
orginal	1531.8	1528.8	977.8	932.1	958.3	651.3	425.3	248.0
unikernel	2754.3	2597.4	2241.7	1865.0	1532.3	1222.9	799.4	517.7

Πίνακας 4.4: Επίδοση της set μετρημένη σε commands / second για original (linux) και unikernel (Rumprun) περιβάλλον ανά value size.

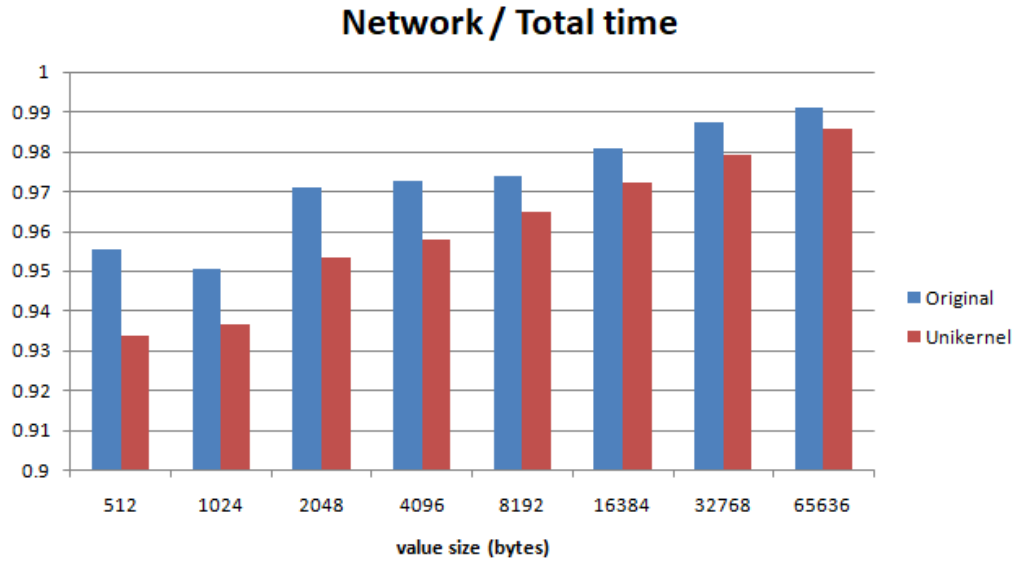
Για την αυθεντική utmem, οι χρόνοι network και redis ορίζονται με τον ίδιο τρόπο. Επειδή η μέτρηση του χρόνου εντός του hypercall είναι σχετικά δύσκολο έργο, λόγω της αυξημένης πολυπλοκότητας όλου του μηχανισμού, και επειδή αυτή η ανάλυση έχει γίνει στην αυθεντική εργασία, οι δύο τελευταίοι χρόνοι, δηλαδή driver και hypercall, μετρώνται ως ένας. Τον ονομάζουμε ioctl, καθώς ουσιαστικά είναι ο χρόνος που χρειάζεται να ικανοποιηθεί η ioctl system call του Put request. Η αντιστοιχία του χρόνου αυτού με το άθροισα driver + hypercall της unikernel utmem είναι αρμονικότατη, καθώς η function call που υλοποιήσαμε ουσιαστικά αντικαθιστά την χρήση της ioctl από την εφαρμογή που χρησιμοποιεί utmem.

Πλέον καταγράφουμε χρονικές διάρκειες, και όχι ρυθμό εξυπηρέτησης των αιτημάτων. Ως εκ τούτου, χαμηλότερες τιμές αποτελεσμάτων μεταφράζονται σε υψηλότερες επιδόσεις.

Έχοντας τα αποτελέσματα, αυτό που παρατηρούμε αμέσως είναι πως η συντριπτική χρονική καθυστέρηση αφορά τον χρόνο network, ο οποίος είναι της τάξης του millisecond (ms). Αυτό ισχύει και για την unikernel και την αυθεντική utmem. Αντίθετα οι υπόλοιποι χρόνοι δεν ξεπερνούν τις δεκάδες microsecond (μs). Στο σχήμα 4.7 φαίνεται αυτή η επικράτηση του network time στον συνολικό χρόνο. Επιπροσθέτως, αν συγκρίνουμε το network time μεταξύ unikernel και linux περιβάλλοντος, βλέπουμε πως το network του Rumprun είναι περίπου το μισό από του linux, επιβεβαιώνοντας την αρχική εκτίμηση πως το Rumprun προσφέρει από μόνο του επιτάχυνση στις εφαρμογές. Επιπλέον, παρατηρούμε πως ο χρόνος redis τελικά μοιάζει ασήμαντος και στα δύο περιβάλλοντα. Τόσο σε σχέση με την συνολική καθυστέρηση όσο σε σχέση και με τα υπόλοιπα στάδια ξεχωριστά, η τιμή του είναι πολύ μικρή, οπότε επιλέγουμε να μην προχωρήσουμε σε περαιτέρω ανάλυση του. Τέλος, αν συγκρίνουμε τον ioctl χρόνο της αυθεντικής utmem, με το άθροισμα driver+hypercall της unikernel utmem, βλέπουμε πως η unikernel έκδοση είναι σχετικά πιο γρήγορη (εικόνα 4.8). Αυτό οφείλεται στην απλούστερη εσωτερική δομή της unikernel έκδοσης, και στο ότι

η unikernel έκδοση επιτελεί μια αντιγραφή λιγότερη των δεδομένων.

Οι απόλυτες τιμές των αποτελεσμάτων των παραπάνω μετρήσεων φαίνονται στούς πίνακες 4.5 και 4.6, για την unikernel utmem και την αυθεντική utmem αντίστοιχα.



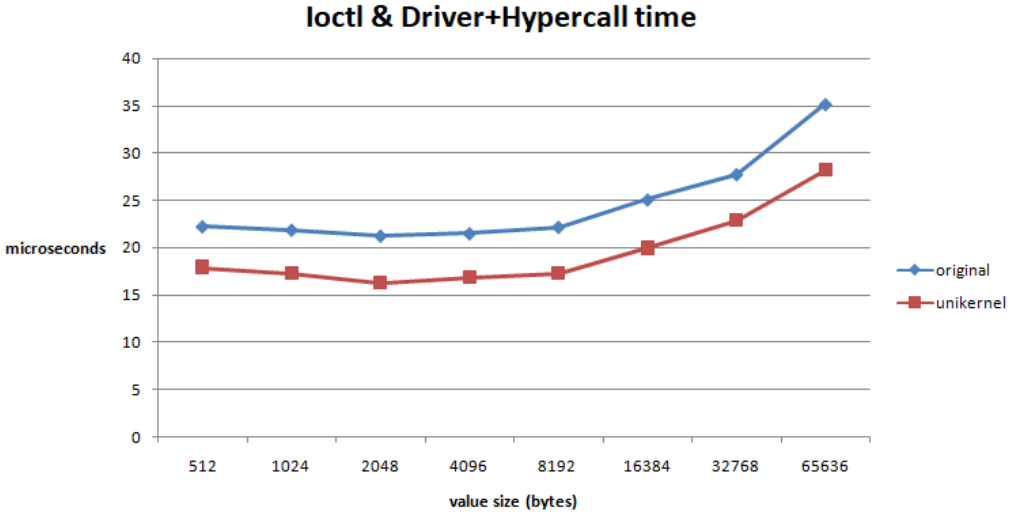
Σχήμα 4.7: Πόσοστό του network time ως πρός τον συνολικό χρόνο καθυστέρησης ανά value size.

value size	512	1024	2048	4096	8192	16384	32768	65636
network	256.7	260.1	340.7	391.1	487.7	722.5	1147.6	2159.6
redis	0.4	0.3	0.4	0.5	0.6	0.8	1.4	3.0
driver	0.5	0.6	0.6	0.7	0.7	1.0	1.6	3.3
hypercall	17.4	16.7	15.7	16.2	16.6	19.0	21.3	24.9
total	274.9	277.8	357.4	408.4	505.6	743.3	1172.0	2190.8

Πίνακας 4.5: Ανάλυση καθυστέρησης σταδίων της Put για unikernel (Rumprun) περιβάλλον. Τιμές σε μs.

4.4 Σχολιασμός

Από το παραπάνω προκύπτει το εξής ενδιαφέρον συμπέρασμα. Με χρήση της utmem, και του Rumprun unikernel framework, είναι δυνατόν να εκτελούνται σε εικονικοποιημένο περιβάλλον unikernels, στα οποία αρχικά δίνεται ελάχιστη μνήμη. Στην



Σχήμα 4.8: Σύγκριση του ioctl time (original) με το driver+hypercall time (unikernel) ανά value size.

value size	512	1024	2048	4096	8192	16384	32768	65636
network	486.9	429.5	725.0	777.0	847.7	1294.0	2189.7	4024.9
redis	0.6	0.5	0.6	0.6	0.6	0.6	0.7	0.8
ioctl	22.2	21.9	21.3	21.5	22.2	25.1	27.7	35.1
total	509.7	452.0	746.8	799.1	870.4	1319.7	2218.2	4060.9

Πίνακας 4.6: Ανάλυση καθυστέρησης σταδίων της Put για original (linux) περιβάλλον. Τιμές σε μs.

περίπτωση του Redis αρκούν μόνο 64 megabytes. Ο φαινομενικός αυτός περιορισμός, δεν απαγορεύει να εκτελούνται memory intensive διεργασίες ως unikernel, καθώς τα tmem pools αποθηκεύουν τα δεδομένα που κανονικά θα βρίσκονταν στην μνήμη εκτός αυτής. Ουσιαστικά εξασφαλίζεται αδιάκοπη εκτέλεση των unikernels, και αποφεύγεται η σπατάλη μνήμης που εν τέλει δεν χρησιμοποιείται από το unikernel. Η πίεση μνήμης μεταφέρεται από τα εικονικοποιημένα περιβάλλοντα στον υφιστάμενο επόπτη, ο οποίος εμφανίζει ανώτερη ικανότητα διαχείρισης αυτής. Παράλληλα, μεγιστοποιείται ο αριθμός των unikernel, και γενικά των εικονικών μηχανών, που μπορούν να εκτελούνται παράλληλα σε ένα φυσικό σύστημα ανά μονάδα μνήμης.

Αυτό που πέτυχε στη αυθεντική εργασία η utmem[19], ήταν να δώσει στον προγραμματιστή ένα χρησιμότατο εργαλείο. Προσαρμόζοντας ελαφρώς την συμπεριφορά της εφαρμογής τους ως προς την διαχείριση της μνήμης κέρδισε σε επιδόσεις και ταχύτητα, σε σχέση με το να εμπιστεύονταν αποκλειστικά τα υποσυστήματα διαχείρισης μνήμης (frontswap). Τώρα, αυτό το εργαλείο προσφέρεται και στο Rumprun

unikernel framework, στο οποίο το σενάριο έλλειψης μνήμης οδηγεί σε καταστροφικές συμπεριφορές. Με την utmem εξαλείφεται η παραπάνω αδυναμία αποτελεσματικά. Ταυτόχρονα, επιβεβαιώσαμε πως η εκτέλεση απλών εφαρμογών σε unikerel περιβάλλοντα οδηγεί σε αυξημένες επιδόσεις, λόγω της αφαίρεσης των περιττών συστατικών ενός συμβατικού λειτουργικού συστήματος.

Σημαντικό, επίσης, είναι το γεγονός πως η έκδοση utmem ως unikernel έχει επιδόσεις ανώτερες από τον αυθεντικό μηχανισμό, τουλάχιστον για ένα εξωτερικό χρήστη που επικοινωνεί με το εκάστοτε εικονικό περιβάλλον. Οι περισσότερες εφαρμογές των unikernel αφορούν υπηρεσίες που συμμετέχουν σε δίκτυα υπολογιστών, άλλωστε εφαλτήρτιο της εργασίας είναι το cloud computing. Θα ήταν, συνεπώς, λάθος να αποτιμίσουμε τις επιδόσεις της unikernel utmem χωρίς να λάβουμε υπ΄ όψιν την networking πτυχή των εφαρμογών. Η επιτάχυνση που προσφέρει το Rumprun κρίνεται ζωτικής σημασίας για εφαρμογές ευαίσθητες στον χρόνο απόδοσης, όπως δείξαμε για μια αποθήκη δεδομένων όπως το Redis. Τα οφέλη, συνεπώς, πολλαπλασιάζονται, όχι μόνο ελαχιστοποιούνται οι ανάγκες μας σε μνήμη, αλλά η απόκριση των εικονικών μηχανών, σε σχέση με αντίστοιχους μηχανισμούς, μεταβάλλεται θετικά.

Το τελικό συμπέρασαμα όλων των παραπάνω είναι πως ο συνδυασμός utmem και Rumprun είναι κάθε άλλο από ασύμφορος ή άκαρπος.

Κεφάλαιο 5

Επίλογος

5.1 Σύνοψη

Συνοψίζουμε τα όσα έχουμε παρουσιάσει έως τώρα στην παρούσα εργασία στον αναγνώστη.

Αρχικά, έγινε αναφορά στις νέες τάσεις του τρόπου να εκτελούμε computational tasks, και συγκεκριμένα στο cloud computing, το οποίο είναι το μοντέλο που χαρακτηρίζει την σύγχρονη εποχή στην ιστορία των υπολογιστών. Αναλύθηκε η εικονικοποίηση, η τεχνολογία που επιτρέπει την ύπαρξη του cloud computing, καθώς και τα διάφορα είδη της που έχουν προκύψει λόγο της δυσκολίας του να εικονικοποιηθεί ένα πλήρες υπολογιστικό σύστημα. Κατέστη σαφές ότι η φιλοσοφία των δημοφιλών λειτουργικών συστημάτων γεννήθηκε σε μια εποχή απλούστερων συστημάτων και περιορισμένων πόρων και άρα ότι δεν ταιριάζει τέλεια στο cloud computing. Παρουσιάστηκαν, λοιπόν, τα εξής προβλήματα με την εως τώρα εικονικοποίηση. Το πρώτο που είδαμε, είναι η υπερβολική πολυπλοκότητα των συμβατικών λειτουργικών συστημάτων και οι υψηλές απαιτήσεις σε πόρους αναλογικά με τις πλέον απλές εφαρμογές που φιλοξενούν. Το δεύτερο είναι η μη βέλτιστη διαχείριση των πόρων, και συγκεκριμένα της υφιστάμενης μνήμης από τα συμβατικά λειτουργικά συστήματα, όταν αυτά βρίσκονται εντός εικονικοποιημένων περιβαλλόντων.

Στη συνέχεια, αναλύθηκαν διάφορες τεχνικές αντιμετώπισης των ανωτέρων προβλημάτων. Το πρώτο πρόβλημα λύνεται με τα unikernels, αυτές τις ευέλικτες και απλές εικονικές μηχανές. Κρίθηκε σκόπιμο να αναλυθούν τα χαρακτηριστικά αυτών και να αναφερθούν τα διάφορα frameworks που υπάρχουν. Ειδικά για το Rumprun framework, με το οποίο εργαστήκαμε, η ανάλυση επάνω σε αυτό ήταν εις βάθος, ως έπρεπε. Για το δεύτερο πρόβλημα προσφέρονται διάφορες λύσεις, ενώ η προσοχή δόθηκε στη μνήμη utmem, καθώς αναλύθηκε η ιστορία της και τα χαρακτηριστικά της.

Τελικά, αποδείχθηκε πως αυτές οι δύο τεχνολογίες μπορούν να συνδυαστούν άρ-

τια, δημιουργώντας unikernels με utmem δυνατότητες. Με σεβασμό στην φιλοσοφία του Rumprun και στην σχεδίαση της utmem, αυτές οι δύο τεχνολογίες εναρμονίστηκαν άψογα. Μάλιστα, με βάση τις μετρήσεις που παρατέθηκαν, η ένωση αυτή δεν είναι άνευ αξίας, αλλά έχει τέλεια εφαρμογή σε συγκεκριμένα ενδεχόμενα εκτέλεσης εικονικοποιημένων συστημάτων. Διαπιστώθηκε, δε, πως η επίδοση του νέου μηχανισμού χαρακτηρίζεται ανώτερη από αυτή της αυθεντικής υλοποίησης, όσον αφορά unikernels που επικοινωνούν με εξωτερικές οντότητες και χρησιμοποιούν τον utmem μηχανισμό.

5.2 Μελλοντικές κατευθύνσεις

Μέχρι στιγμής, η αδυναμία της έκδοσης utmem για το Rumprun είναι πως ως μηχανισμός αποθήκευσης δεδομένων είναι σχετικά αργός σε σχέση με την αποθήκευση στη μνήμη της εικονικής μηχανής. Θα ήταν δυνατόν να βελτιστοποιήσουμε περαιτέρω με διάφορες τεχνικές την χρήση της utmem ώστε οι ταχύτητες να πλησιάζουν την in-memory αποθήκευση των δεδομένων. Κύρια αδυναμία είναι πως, με βάση την μορφή του μηχανισμού, απαιτείται ένα hypercall για κάθε αίτηση utmem. Αναμένουμε πως αν μειώνονταν ο αριθμός των hypercalls που απαιτούνται θα αυξάνονταν οι επιδόσεις του μηχανισμού.

Μια λύση θα ήταν να κρατάμε ένα buffer μνήμης με δεδομένα, τα οποία θα μεταφέρονταν στην tmem pool όταν περνούσαν ένα προκαθορισμένο μέγεθος. Έτσι δεν θα χρειάζονταν να κάνουμε ένα hypercall ανά request, αλλά θα εκμεταλλευόμασταν bulk-insertions με μόνο ένα hypercall. Επί παραδείγματι, ας φανταστούμε πως έρχονται χίλια αιτήματα utmem όπου το καθένα επιθυμεί να αποθηκεύσει δεδομένα μεγέθους ενός kilobyte. Θα μπορούσαμε να κάνουμε μαζική εισαγωγή όλων αυτών, όταν συμπληρωθεί και το π.χ. χιλιοστό αίτημα με μόνο ένα hypercall με δεδομένα μεγέθους χίλια επί ένα kilobyte συν ό,τι χρειάζεται για τα κλειδιά, αντί για χίλια «μικρά» hypercalls.

Άλλη κατεύθυνση θα ήταν τα δεδομένα να συμπιέζονται εντός του unikernel, πριν αποσταλούν στο backend, ώστε πάλι να μειώνεται ο αριθμός των απαιτούμενων hypercalls. Ωστόσο, επειδή η συμπίεση δεδομένων καταναλώνει σημαντική επεξεργαστική ισχύ, αυτό θα αφορούσε σενάρια εκτέλεσης όπου ο επεξεργαστής είναι πόρος σε αφθονία, ενώ η μνήμη σε σχετική έλλειψη. Μάλιστα, θα μπορούσε η εκάστοτε εφαρμογή που τρέχει ως unikernel, να εξειδικεύει τον αλγόριθμο συμπίεσης ώστε να ταιριάζει στο είδος των δεδομένων της καλύτερα. Η εξειδίκευση αυτή είναι πιο εύκολη από την πλευρά του unikernel, καθώς θεωρητικά ο προγραμματιστής γνωρίζει την φύση των δεδομένων, ενώ το backend εστιάζει στην αποθήκευση αφηρημένης μορφής δεδομένων.

Όσον αφορά use case με πολλές προσβάσεις στην μνήμη επάνω στα ίδια δεδομένα, η αποθήκευση και ανάκτηση των δεδομένων με ένα tmem pool από μόνη της μειώνει την απόδοση του συστήματος. Γνωστή και αποδοτική πλέον λύση τέτοιων περιπτώσε-

ων είναι η χρήση κρυφών μνημών. Στην περίπτωση μας θα ήταν η ύπαρξη ενός μικρού memory area μέσα στο unikernel, είτε στο ανώτερο επίπεδο είτε στο επίπεδο του driver, στο οποίο να βρίσκονται τα περισσότερα συχνά ανταλλάξιμα δεδομένα μεταξύ του frontend και του backend.

Αναθεωρώντας ακόμα περισσότερο την φιλοσοφία της utmem, η χρήση κάποιου ασύχρονου μηχανισμού επικοινωνίας host και guest, και όχι του σύγχρονου hypercall, θα έλυνε τα προβλήματα που προέρχονται από το blocking των αιτημάτων. Βέβαια, αυτό απαιτεί σημαντικά περισσότερη προσοχή ως προς την ασφάλεια, την αποφυγή αδιεξόδων (deadlock), και την εγγύηση πως τα δεδομένα μας θα είναι πάντα διαθέσιμα και ασφαλή.

Επιπροσθέτως, το γεγονός πως επιτρέψαμε την χρήση της utmem από unikernels στηριζόμενοι στο Rumprun, δεν σημαίνει πως τα άλλα frameworks δεν είναι συμβατά. Ανάλογα με το πόσο εύκολο ή δύσκολο είναι σε έναν προγραμματιστή να προσθέσει μια κλήση επόπτη στο framework που τον ενδιαφέρει, θεωρούμε πως τόσο εύκολο ή δύσκολο είναι και το porting της utmem. Για παράδειγμα, το mirageOS αν και σχεδιάστηκε για το Xen hypervisor, υποστηρίζει και το KVM πλέον, οπότε πιθανότατα η ενσωμάτωση του μηχανισμού της utmem να είναι σχετικά εύκολη διαδικασία. Το ίδιο ισχύει για τα περισσότερα frameworks που υποστηρίζουν το KVM ως επόπτη.

Τέλος, ένα θέμα που σίγουρα αξίζει να ερευνηθεί, καθώς θα καθιστούσε την tmem κατεξοχήν πολύτιμο εργαλείο, είναι κατά πόσον θα ήταν δυνατόν να υλοποιηθεί ένα υποσύστημα εικονικής μνήμης στο Rumprun το οποίο να χρησιμοποιεί την tmem. Όπως αναφέραμε και στο υποκεφάλαιο του Rumprun, τα rump kernels δεν υποστηρίζουν την εικονική μνήμη. Μπορούμε, συνεπώς, να φανταστούμε ένα σύστημα σαν το frontswap του linux, το οποίο όμως δεν θα χρησιμοποιεί περιφεριακές συσκευές για την αποθήκευση των σελίδων μνήμης, αλλά θα στρέφονταν αυτόματα στην tmem pools του επόπτη. Αυτομάτως ο προγραμματιστής θα εποφελούνταν από όλα τα οφέλη της utmem, όσον αφορά την απελευθέρωση της πίεσης μνήμης, δίχως να χρειάζεται να προσαρμόζει τον κώδικα των εφαρμογών του. Θα χρειαζόταν, λοιπόν, να τροποποιηθεί ο Bare Metal Kernel της στοίβας του Rumprun, καθώς αυτός είναι υπεύθυνος για την διαχείριση των σελίδων την μνήμης. Εντούτοις, θεωρούμε πως θα ήταν από τα δύσκολα μελλοντικά εγχειρήματα που προτείνουμε, καθώς η διαδικασία διαχείρισης της μνήμης είναι μια από τις πιο βασικές και ευαίσθητες διαδικασίες που οφείλει να εκτελεί ένα λειτουργικό σύστημα, ενώ μάλιστα ένα τέτοιο υποσύστημα σύμφωνα με τα υπάρχοντα δεδομένα θα περιόριζε το Rumprun αποκλειστικά στο KVM.

Βιβλιογραφία

[1] Cloud computing. https://en.wikipedia.org/wiki/Cloud_computing, Last accessed on 12/2/2021.

[2] IncludeOS on github. https://github.com/includeos/IncludeOS, Last accessed on 12/2/2021.

[3] Interview: Antti Kantee: The Anykernel and Rump Kernels. https://archive.fosdem.org/2013/interviews/2013-antii-kantee/, *FOSDEM 2013*, Last accessed on 12/2/2021.

[4] Introduction to Redis. https://redis.io/topics/introduction, Last accessed on 12/2/2021.

[5] Linux KVM Hypercall. https://www.kernel.org/doc/html/latest/virt/kvm/hypercalls.html, Last accessed on 12/2/2021.

[6] On rump kernels and the Rumprun unikernel. https://xenproject.org/2015/08/06/on-rump-kernels-and-the-rumprun-unikernel/, Last accessed on 12/2/2021.

[7] Overview of MirageOS. https://mirage.io/wiki/overview-of-mirage, Last accessed on 12/2/2021.

[8] QEMU wiki main page. https://wiki.qemu.org/Main_Page, Last accessed on 12/2/2021.

[9] Redis as Unikernel. https://github.com/rumpkernel/rumprun-packages/tree/master/redis, Last accessed on 12/2/2021.

[10] Redis persistence. https://redis.io/topics/persistence, Last accessed on 12/2/2021.

[11] Solo5 on github. https://github.com/Solo5/solo5, Last accessed on 12/2/2021.

[12] The Java Virtual Machine . https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-1.html#jvms-1.2, Last accessed on 12/2/2021.

[13] Understanding Full Virtualization ,Paravirtualization, and Hardware Assist. Technical Report, *VMware, Inc.* `https://www.vmware.com/content/dam/digitalmarketing/vmware/en/pdf/techpaper/VMware_paravirtualization.pdf`, Last accessed on 12/2/2021.

[14] Unikernel. `https://en.wikipedia.org/wiki/Unikernel`, Last accessed on 12/2/2021.

[15] Virtual Memory. `https://en.wikipedia.org/wiki/Virtual_memory`, Last accessed on 12/2/2021.

[16] What are Unikernels? `http://unikernel.org/`, Last accessed on 12/2/2021.

[17] x86 virtualization. `https://en.wikipedia.org/wiki/X86_virtualization#Intel_virtualization_(VT-x)`, Last accessed on 12/2/2021.

[18] Λάγκας Νικολός Ορέστης. Μελέτη και αποτίμηση μεθόδων εκτέλεσης εφαρμογων ως Unikernel σε αρχιτεκτονικές ARM. Master thesis, *Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών, Εθνικό Μετσόβιο Πολυτεχνείο*, 2018.

[19] Τσαλαπάτης Αιμίλιος. Σχεδίαση και Υλοποίηση Μηχανισμού Διαχείρησης Ελαστικής Μνήμης σε Εικονικά Περιβάλλοντα. Master thesis, *Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών, Εθνικό Μετσόβιο Πολυτεχνείο*, 2018.

[20] Μάινας Χαράλαμπος. Σχεδιασμός και υλοποίηση μηχανισμών fork και pipe σε Unikernel. Master thesis, *Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών, Εθνικό Μετσόβιο Πολυτεχνείο*, 2019.

[21] Anil Madhavapeddy, David J. Scott. unikernels: The Rise of the Virtual Library operating System. *Communications of the ACM*, 57(1), 2014. pp. 61-69.

[22] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott , Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand and John Crowcroft. Unikernels: Library Operating System for the Cloud. *ASPLOS '13*, 2013. pp. 461–472.

[23] Antti Kantee and Justin Cormack. Rump Kernels No OS? No Problem! *;login:*, 39(5), 2014. pp. 11-17.

[24] Avi Kivity, Dor Laor, Glauber Costa, Pekka Enberg , Nadav Har'El, Don Marti and Vlad Zolotarov. OSV-Optimizing the Operating System for Virtual Machines. *USENIX ATC '14'*, 2014.

[25] Dan Magenheimer. Transcendent memory in a nutshell. *lwn.net*, 2011. `https://lwn.net/Articles/454795/`, Last accessed on 12/2/2021.

[26] Dan Williams and Ricardo Koller. Unikernel Monitors: Extending Minimalism Outside of the Box. *USENIX Workshop on HotTopics in Cloud Computing (HotCloud '16)*, 2016.

[27] Megenheimer, Dan et al. Transcedent Memory on Xen. *Xen Summit*, 2009. pp. 1-3.

[28] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand , Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt and Andrew Warfield. Xen and the Art of Virtualization. *SOSP 2003*, 2003. pp. 164–177.

[29] Tsalapatis Aimilios, Gerangelos Stefanos, Psomadakis Stratos, Papazafeiropoulos Konstantinos, Koziris Nectarios. utmem:Towards Memory Elasticity in Cloud Workloads. *ISC High Performance 2018 International Workshops*, 2018. pp. 173-183.