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

Μελέτη και αποτίμηση μεθόδων εκτέλεσης εφαρμογών ως *Unikernels* σε αρχιτεκτονικές **ARM**

Διπλωματικη Εργασια

του

ορεστη λαγκα νικολου

Επιβλέπων: Νεκτάριος Κοζύρης
Καθηγητής Ε.Μ.Π.

Εργαστηριο Υπολογιστικων Συστηματων
Αθήνα, Οκτώβριος 2018

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών
Εργαστήριο Υπολογιστικών Συστημάτων

Μελέτη και αποτίμηση μεθόδων εκτέλεσης εφαρμογών ως *Unikernels* σε αρχιτεκτονικές **ARM**

Διπλωματικη Εργασια

του

ορεστη λαγκα νικολου

Επιβλέπων: Νεκτάριος Κοζύρης
Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 22η Οκτωβρίου 2018.

(Υπογραφή)	*(Υπογραφή)*	*(Υπογραφή)*
.....	.....	.....
Νεκτάριος Κοζύρης	Γεώργιος Γκούμας	Νικόλαος Παπασπύρου
Καθηγητής Ε.Μ.Π.	Επίκουρος	Αναπληρωτής
	Καθηγητής Ε.Μ.Π.	Καθηγητής Ε.Μ.Π

Αθήνα, Οκτώβριος 2018

(Υπογραφή)

.....

Ορεστης Λαγκας Νικολος

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

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών
Εργαστήριο Υπολογιστικών Συστημάτων

Ευχαριστίες

Αρχικά θα ήθελα να ευχαριστήσω τον επιβλέποντα καθηγητή κ.Νεκτάριο Κοζύρη για την ευκαιρία που μου έδωσε να εκπονήσω την παρούσα εργασία στο Εργαστήριο Υπολογιστικών Συστημάτων του Ε.Μ.Π.

Για την ολοκλήρωση της οφείλω να ευχαριστήσω τον Ερευνητή Αναστάσιο Νάνο για τη στενή συνεργασία, την αμέριστη προθυμία και τις καθοριστικές ιδέες του καθώς και τον Κωνσταντίνο Παπαζαφειρόπουλο, υποψήφιο Διδάκτορα του Εργαστηρίου, για τη διαρκή συνεισφορά του, το χρόνο του και την καθημερινή υποστήριξη. Θα ήθελα επίσης να ευχαριστήσω τον υποψήφιο Διδάκτορα Στράτο Ψωμαδάκη για τις ενδιαφέρουσες συζητήσεις μας, καθώς και όλα τα μέλη του ΕΥΣ που είχα την ευκαιρία να έρθω σε επαφή στα πλαίσια εκπόνησης αυτής της Διπλωματικής εργασίας.

Καθώς αυτή η εργασία σηματοδοτεί και την ολοκλήρωση των σπουδών μου θα ήταν παράλειψη να μην ευχαριστήσω:

Τους γονείς μου, Σπύρο και Χαρά, για την υπομονή τους, τη στήριξη τους και πάνω από όλα την εμπιστοσύνη που νιώθω ότι μου δείχνουν όλα αυτά τα χρόνια αλλά και τον αδερφό μου, Γιάννη, που με την επίτευξη των στόχων του με γεμίζει θαυμασμό και αισιοδοξία.

Το Δημήτρη, τον Κωστή, τον Παναγιώτη για την ανοχή τους και τη φιλία τους. Όσους και όσες έχουν σταθεί δίπλα μου με ειλικρίνεια ανά τα χρόνια και θα καταλαβαινόμασταν και με μία ματιά.

Τους ΑΝεξάρτητους Αριστερούς Φοιτητές Ηλεκτρολόγους γιατί πλάι τους έμαθα πολλά αλλά πάνω απ΄όλα έμαθα αυτό που, κάποιος έγραψε στον τοίχο, και ελπίζω να με συντροφεύει για καιρό· *Χαμένοι είναι οι αγώνες που δε δόθηκαν ποτέ.*

Ορέστης Λάγκας Νικολός

Περίληψη

Καθώς ο όγκος των δεδομένων προς επεξεργασία αυξάνεται συνεχώς, εμφανίζεται επιτακτικά η ανάγκη να εστιάσουμε στην αποδοτικότερη επεξεργασία τους με τη χρήση κατά το δυνατόν λιγότερων υπολογιστικών πόρων. Ένας σημαντικός παράγοντας που επιβαρύνει το περιβάλλον εκτέλεσης μίας εφαρμογής στο Cloud, είναι τα ίδια τα συμβατικά Λειτουργικά Συστήματα. Η εκτέλεση εφαρμογών πάνω από ΛΣ γενικού σκοπού στο Cloud περιλαμβάνει περιττές λειτουργίες που δε συνάδουν με την αρχιτεκτονική του Cloud. Σκοπός της εργασίας αυτής είναι η μελέτη μεθόδων απαλοιφής αυτής της επιβάρυνσης μέσω της δημιουργίας ενός λεπτότερου στρώματος εξαρτήσεων που θα καθιστά δυνατή την αυτόνομη εκτέλεση μίας εφαρμογής στο υλικό, ως ένα ενιαίο εκτελέσιμο αρχείο. Στα πλαίσια αυτά μελετάμε τις δομές των Unikernel ως μία εναλλακτική προσέγγιση για την αναπροσαρμογή της αρχιτεκτονικής εκτέλεσης των εφαρμογών στο Cloud. Αποτιμούμε τις επιδόσεις τους σε βασικές λειτουργίες (δίκτυο, πρόσβαση στη μνήμη του οικοδεσπότη) που εκτελούνται συχνά στα πλαίσια μίας εφαρμογής και αποτελούν βασικό παράγοντα επιβάρυνσης του χρόνου εκτέλεσής της. Συγκρίνουμε τις μεθόδους αυτές με τις επικρατούσες προσεγγίσεις στο Cloud Computing, εντοπίζουμε αξιόλογες προοπτικές και προτείνουμε μελλοντικές επεκτάσεις.

Λέξεις Κλειδιά

Τεχνολογία Νέφους, Εικονικοποίηση, Εικονικές Μηχανές, Πυρήνας Λειτουργικού Συστήματος, Διαδίκτυο Πραγμάτων, Ενιαίος Χώρος Διευθύνσεων

Abstract

The continuously growing amount of data to be processed makes it really important to maximize the efficiency of the processing by also minimizing the underlying resources. An important downside in the execution environment of an application on the Cloud are the traditional Operating Systems. The purpose of this thesis is to perform a feasibility study of the possible methods to limit the burden of the Operating System by creating a lightweight layer of dependencies for the application; whilst keeping it possible to run it directly on the hardware (baremetal or hypervisor) as an single autonomous image. To-that-end, we focus on Unikernels as an alternative approach for the modernization of the execution architecture for Applications running on the Cloud. We are evaluating the performance of common operations (network, block device), that are usually adding a significant overhead on the execution time of an application. Finally by comparing this new approach with the traditional ones standing, we find out that the results are remarkable and we propose further extensions for future work.

Keywords

Unikernels, Cloud, Virtualization, Virtual Machines, Operating System Kernel, Internet of Things, Single Address Space

Περιεχόμενα

Κατάλογος Σχημάτων

Κατάλογος Πινάκων

Κεφάλαιο 1

Εισαγωγή

Τα τελευταία χρόνια ο όγκος των δεδομένων προς επεξεργασία έχει μεγαλώσει εκθετικά ενώ οι τρόποι επεξεργασίας τους παραμένουν απαιτητικοί τόσο σε υπολογιστική ισχύ, όσο και σε ενέργεια. Η εποχή που το Personal Computing διαδόθηκε και επικράτησε έχει αρχίσει να αντικαθίσταται από την εποχή του Cloud Computing, όσον αφορά την αποθήκευση των δεδομένων σε απομακρυσμένα data center όσο και την απομακρυσμένη παροχή υπηρεσιών. Σε αυτό συνηγορεί πλέον και η ραγδαία εξάπλωση του Internet of Things με την έννοια ότι τα δεδομένα πλέον παράγονται και καταγράφονται ως ψηφιακές απεικονίσεις πληροφοριών του πραγματικού κόσμου, από χιλιάδες συσκευές (smartphones, αισθητήρες κλπ), και στη συνέχεια διαχέονται στο δίκτυο με σκοπό την περαιτέρω επεξεργασία τους.

Η τάση αυτή έχει στρέψει την προσοχή της επιστημονικής κοινότητας αλλά και της βιομηχανίας στην αποδοτικότερη αξιοποίηση των υπολογιστικών πόρων και στην προσπάθεια δραστικής μείωσης τόσο του χρόνου επεξεργασίας των δεδομένων (execution time) όσο και της καθυστέρησης λόγω του δικτύου (latency). Σε αυτό το πλαίσιο, η αναζήτηση της αύξησης των επιδόσεων για τα συστήματα επόμενης γενιάς και μεγάλης κλίμακας περιλαμβάνει τη διαρκή προσπάθεια βελτιστοποίησης της αρχιτεκτονικής αλλά και του λειτουργικού συστήματος στο περιβάλλον των οποίων εκτελούνται οι διάφορες εφαρμογές. Ένας σημαντικός παράγοντας σε αυτή την αναζήτηση είναι η επιβάρυνση που προσθέτει το λειτουργικό σύστημα στο οποίο εκτελείται μία εφαρμογή και η διερεύνηση μεθόδων για την αποφυγή σπατάλης υπολογιστικών πόρων (cpu time, μνήμη) σε άσκοπες, αναφορικά με την κύρια εφαρμογή, διεργασίες και component του λειτουργικού.

1.1 Αντικείμενο της διπλωματικής

Μια ενδιαφέρουσα προσέγγιση στη μείωση του θορύβου του Λειτουργικού Συστήματος και των περιττών εξαρτήσεων στο περιβάλλον εκτέλεσης μιας εφαρμογής είναι η δημιουργία ενός λεπτού στρώματος εξαρτήσεων (βιβλιοθήκες, Λειτουργικό Σύστημα) και η σύνθεση ενός ενιαίου εκτελέσιμου αρχείου της εφαρμογής (unikernel), που θα μπορεί να εκτελεστεί αυτόνομα, όπως σε ένα κοινό λειτουργικό σύστημα. Η απαλοιφή των εξαρτήσεων αυτών μπορεί να οδηγήσει σε άμεσα οφέλη όπως η μείωση της μνήμης που καταναλώνει μια εφαρμογή, η μείωση του

χρόνου εκτέλεσής της και η μείωση της συνολικής επιβάρυνσης στο δίκτυο εξοικονομώντας έτσι άμεσα πόρους για το σύστημα.

Στην εργασία αυτή διερευνούμε τη δυνατότητα μείωσης της επιβάρυνσης στην εκτέλεση των εφαρμογών μέσω της εκτέλεσής τους ως Unikernel. Μελετάμε τις υπάρχουσες λύσεις στην προσέγγιση αυτή και αποτιμούμε τις επιδόσεις αιτημάτων Εισόδου / Εξόδου για συσκευές χαμηλής ενεργειακής κατανάλωσης αρχιτεκτονικής ARM.

Το γεγονός ότι η πλειοψηφία των εφαρμογών που εκτελούνται σε περιβάλλοντα cloud (AWS, Azure κλπ) φιλοξενούνται σε εικονικές μηχανές, και με τις δυνατότητες που προσφέρει το Virtualization, είναι εύλογο να αναμένουμε ότι τα οφέλη της παραπάνω μελέτης, αναλογιζόμενοι και το δυναμικό χαρακτήρα της ζήτησης για υπηρεσίες, θα είναι επίσης μεγάλα. Η απαλοιφή των περιττών εξαρτήσεων που προσθέτει το Λειτουργικό Σύστημα οδηγεί (i) στην αρχικοποίηση λιγότερων διεργασιών και διεπαφών κατά την εκκίνηση του, βελτιώνοντας την ελαστικότητα των πόρων του συστήματος, καθώς σε συνδυασμό με την (ii) αναμενόμενη μείωση του χρόνου εκτέλεσης μπορούμε να πετύχουμε (iii) αμεσότερη αποδέσμευση πόρων, οδηγούμενοι τελικά σε (iv) υψηλότερη ή αντίστοιχη εξυπηρεσιμότητα, χρησιμοποιώντας όμως (v) λιγότερους πόρους. Τέλος ένας σημαντικός παράγοντας στην όλη διαδικασία είναι η (vi) αύξηση της ασφάλειας των συστημάτων με τον περιορισμό της συνολικής έκθεσης σε κινδύνους που μπορεί να προσθέτονταν από άσχετους, με την εκτέλεση μιας εφαρμογής, παράγοντες καθώς και η (vii) απομόνωση του περιβάλλοντος εκτέλεσης που προσφέρει η τεχνολογία του Virtualization.

Στην παραπάνω μελέτη διαπιστώνουμε ωστόσο τεχνικές δυσκολίες που αφορούν τόσο την υποστήριξη των εναλλακτικών αρχιτεκτονικών χαμηλής κατανάλωσης (ARM) όσο και τη δυνατότητα ικανοποίησης των πραγματικών εξαρτήσεων μίας εφαρμογής από τις διάφορες κλασικές βιβλιοθήκες ενός παραδοσιακού λειτουργικού συστήματος που συνήθως είναι αλληλένδετες μεταξύ τους. Ωστόσο θεωρούμε ότι τα ωφέλη αποδεικνύονται κυρίως στο χαμηλότερο επίπεδο του Application Stack μέσω της απαλοιφής των συνεχόμενων εναλλαγών του σεναρίου εκτέλεσης (mode switches από user σε kernel) και ότι τα υψηλότερα επίπεδα που αφορούν την υλοποίηση πρωτοκόλλων και μεθόδων θα είναι ούτως ή άλλως πάντοτε αντικείμενο βελτίωσης.

1.1.1 Συνεισφορά

Η συνεισφορά της διπλωματικής συνοψίζεται ως εξής:

1. Μελετάμε τις υπάρχουσες και επικρατέστερες Unikernel λύσεις.

2. Μελετάμε την υποστήριξη που παρέχουν για αρχιτεκτονικές ARM.

3. Αξιολογούμε την επίδοση τους σε απαιτητικά αιτήματα Εισόδου / Εξόδου

4. Μελετάμε και αξιολογούμε την αντίστοιχη επίδοση παραδοσιακών λύσεων πλήρους εικονικοποίησης (Linux Guest QEMU/KVM)

5. Συγκρίνουμε τα αποτελέσματα καθόλη τη διάρκεια της μελέτης αναζητώντας πιθανά περιθώρια βελτίωσης στα σενάρια εκτέλεσης εφαρμογών σε εικονικοποιημένα περιβάλλοντα.

1.2 Οργάνωση του τόμου

Στο Κεφάλαιο 2 περιγράφουμε τις υπάρχουσες προσεγγίσεις όσον αφορά τα σενάρια εκτέλεσης των εφαρμογών, δίνοντας έμφαση στις διαφορετικές τεχνολογίες εικονικοποίησης όπως αυτές εχουν διαμορφωθεί σύμφωνα με τις σύγχονες ανάγκες. Στο Κεφάλαιο 3 παρουσιάζουμε τις δομές Unikernel που μελετάμε στα πλαίσια αυτής της εργασίας παραθέτοντας διάφορες πηγές από τη σχετική βιβλιογραφία. Στο Κεφάλαιο 4 εστιάζουμε στην δομή και τη συμβατότητα ορισμένων εξ'αυτών με τις αρχιτεκτονικές ARM όπου και αποφασίσαμε να τις αξιολογήσουμε. Στο Κεφάλαιο 5 παρουσιάζουμε τα αποτελέσματα της αξιολόγησης που κάναμε παραθέτοντας μετρήσεις και συγκριτικά με άλλες τεχνολογίες εικονικοποίησης, καθώς και τα επιμέρους συμπεράσματα για κάθε πείραμα. Τέλος στο Κεφάλαιο 6 εκφράζουμε τα συνολικά συμπεράσματα που αφορούν την αξιολόγηση που επιχειρήσαμε καθώς και πτυχές βελτίωσης που προέκυψαν στην πορεία της εργασίας αυτής, προτείνοντας την περαιτέρω διερεύνησή τους.

Κεφάλαιο 2

Θεωρητικό Υπόβαθρο - Βιβλιογραφία

2.1 Εισαγωγή

Τα τελευταία χρόνια η ευρεία εξάπλωση και εμπλοκή του διαδικτύου με τον ένα ή τον άλλο τρόπο σχεδόν σε κάθε πτυχή της τεχνολογικής προόδου έχει *γεννήσει* μια σειρά εννοιών (Cloud, In-house, Internet of Things κλπ) που περιγράφουν τις διαφορετικές αρχιτεκτονικές / τεχνολογίες οργάνωσης της υπολογιστικής ισχύς και των δεδομένων. Οι έννοιες αυτές ακριβώς επειδή επιχειρούν να περιγράψουν με έναν πιο αφαιρετικό τρόπο την τοπολογία για την εκτέλεση μιας εφαρμογής ή μάλλον το περιβάλλον/concept εκτέλεσης της, χρησιμοποιούνται συνήθως στα πλαίσια μίας ήδη προχωρημένου επιπέδου κουβέντας ως δεδομένες. Πριν προχωρήσουμε λοιπόν σε μια πιο αναλυτική μελέτη - παρουσίαση του ειδικού θέματος αυτής της εργασίας θα περιγράψουμε κάποιες από αυτές τις έννοιες που θέτουν και το (πολύ) ευρύτερο πλαίσιο της: *πού και πώς επεξεργαζόμαστε τον αυξανόμενο όγκο δεδομένων, ποιες από τις κλασικές λειτουργίες ενός Λειτουργικού Συστήματος (ΛΣ) χρειαζόμαστε για την επεξεργασία αυτή και ποιο μπορεί να είναι ένα μελλοντικό μονοπάτι προς μια αποδοτική εξοικονόμηση και ταυτόχρονα πλήρη αξιοποίηση υπολογιστικών πόρων.*

2.2 Σενάρια Εκτέλεσης Εφαρμογών

2.2.1 In-House Servers

Η ραγδαία ανάπτυξη της τεχνολογίας μικροεπεξεργαστών τη δεκαετία του 1980 οδήγησε στην ευρεία επικράτηση του μοντέλου των προσωπικών υπολογιστών σε αντίθεση με τα προηγούμενα χρόνια που η κατοχή και χρήση υπολογιστικών πόρων είχε απαγορευτικό κόστος, με αποτέλεσμα όσοι είχαν ανάγκη υπολογιστικής δύναμης να *δανείζονται* χρόνο και πρόσβαση σε κάποια υπολογιστική μονάδα. Με την επικράτηση του Personal Computing τόσο οι εταιρίες όσο και οι απλοί χρήστες είχαν τη δυνατότητα να έχουν τις δικές τους υπολογιστικές υποδομές. Ωστόσο καθώς το διαδίκτυο απέκτησε όλο και υψηλότερες ταχύτητες

η μόδα τείνει να ξαναλλάξει προς το προηγούμενο μοντέλο, με την έννοια της *ενοικίασης* πόρων (Cloud Computing). Σε αυτή την αλλαγή συνηγορεί το γεγονός ότι ο όγκος των δεδομένων προς επεξεργασία μεγάλωσε με εκθετικό τρόπο ενώ οι τρόποι επεξεργασίας τους παραμένουν απαιτητικοί τόσο σε υπολογιστική ισχύ όσο και σε ενέργεια. Παρόλα αυτά ακόμα και σήμερα υπάρχουν οργανισμοί και απλοί χρήστες που προτιμούν να έχουν τη δική τους In-House υποδομή είτε διότι απαιτούν πλήρη έλεγχο πάνω στην ασφάλεια και ιδιωτικότητα των δεδομένων τους είτε διότι εκτελούν εφαρμογές που απαιτούν μικρό latency ή μικρή ανοχή αστοχίας πρόσβασης στο δίκτυο, είτε για άλλους λόγους.

Στην περίπτωση του In-House Server ο ίδιος ο χρήστης / πελάτης είναι και ο *πάροχος* της υποδομής. Αυτό σημαίνει ότι έχει τη δυνατότητα να εγκαταστήσει και να ρυθμίσει (*στήσει*) τις υπηρεσίες που χρειάζεται με τρόπο καθαρά προσαρμοσμένο στις ανάγκες του, αξιοποιώντας επίσης το υλικό του βέλτιστα. Φυσικά σε αυτή την περίπτωση επιβαρύνεται ο ίδιος με το κόστος αγοράς και συντήρησης του υλικού για την επεξεργασία και αποθήκευση των δεδομένων. Η εκτέλεση δε των εφαρμογών γίνεται τοπικά χωρίς να επιβαρύνεται ιδιαίτερα απο το overhead του διαδικτύου (συγκριτικά με ένα απομακρυσμένο Cloud). Αυτό μπορεί να αποδειχθεί σημαντικός παράγοντας στο χρόνο εκτέλεσης μιας εφαρμογής για το χρήστη, ειδικά εάν ο καθαρός χρόνος εκτέλεσής της είναι ικανοποιητικά μικρός σε σχέση με το overhead που θα προσέθετε η ανάθεση της σε κάποιο απομακρυσμένο Cloud Infrastructure.

2.2.2 Cloud Computing

Η τεχνολογία του Cloud Computing επιτρέπει τη διαρκή πρόσβαση σε κοινόχρηστους ρυθμιζόμενους υπολογιστικούς πόρους (δίκτυο, εξυπηρετητές, εφαρμογές και υπηρεσίες) με υψηλή ευελιξία, ελάχιστη διαχειριστική προσπάθεια από το χρήστη και υψηλή αυτοματοποίηση. Η αποθήκευση, επεξεργασία και η χρήση των δεδομένων καθώς και του λογισμικού και των υπηρεσιών γίνεται μέσω απομακρυσμένων υπολογιστών σε κεντρικά Datacenter. Όπως αναφέραμε και στην προηγούμενη παράγραφο, η εκθετική αύξηση των δεδομένων που χρήζουν επεξεργασίας σε συνδυασμό με την αναντίστοιχη εξέλιξη της δυνατότητας επεξεργασίας τους, μεταφρασμένη πάντα σε υπολογιστικούς πόρους, καθιστά μάλλον πιο ακριβή την κατοχή και διατήρηση μιας κατάλληλης σχετικής υπολογιστικής υποδομής συγκριτικά με την απευθείας δέσμευση πόρων από κάποιον πάροχο μιας τέτοιας. Αυτή η παρατήρηση δικαιολογεί και την ευρεία επικράτηση του Cloud Computing στις υπολογιστικές ανάγκες των εταιρειών, οργανισμών ή ακομα και των (πιο απαιτητικών) χρηστών.

Βασικά χαρακτηριστικά

Τα 5 βασικά χαρακτηριστικά του Cloud Computing είναι[1]:

- Αυτοεξυπηρέτηση κατά παραγγελία : Ο χρήστης μπορεί μονομερώς να εφοδιαστεί με υπολογιστικούς πόρους όπως χρόνο εξυπηρέτησης ή αποθηκευτικό χώρο στο δίκτυο ανάλογα με τις ανάγκες του και αυτοματοποιημένα χωρίς να απαιτείται ανθρώπινη αλληλεπίδραση με τον πάροχο της υπηρεσίας.

- Ευρεία δικτυακή πρόσβαση: Οι πόροι είναι προσβάσιμοι μέσω του δικτύου χρησιμοποιώντας μηχανισμούς που επιτρέπουν τη χρήση τους από διαφορετικές πλατφόρμες (κινητά, ταμπλέτες, φορητούς υπολογιστές κλπ)

- Συγκεντρωση πόρων: Ο πάροχος της υποδομής συγκεντρώνει όλους τους πόρους εξυπηρετώντας πολλαπλούς πελάτες. Η εξυπηρέτηση αυτή γίνεται με δυναμικό τρόπο, οι ανάθεση των φυσικών ή εικονικών πόρων στους πελάτες χαρακτηρίζεται από μία κινητικότητα ανάλογα με τη συνολική ζήτηση.

- Ταχεία Ελαστικότητα: Οι δυνατότητες του Cloud μπορούν να ανακοινώνονται και να παρέχονται με ελαστικό τρόπο, σε ορισμένες περιπτώσεις αυτοματοποιημένα, και αντιστοίχως να κλιμακώνονται ταχέως προς τα έξω ή προς τα μέσα ανάλογα με τη ζήτηση. Στον ίδιο το χρήστη οι διαθέσιμες δυνατότητες μπορεί να εμφανίζονται απεριόριστες και διαθέσιμες σε οποιαδήποτε ποσότητα και ανά πάσα στιγμή.

- Καταμέτρηση υπηρεσιών: Τα συστήματα Cloud έχουν τη δυνατότητα να ελέγχουν αυτόματα και να βελτιστοποιούν τη χρήση των πόρων σύμφωνα με μία αφαιρετική μετρική κατάλληλη για τον τύπο της υπηρεσίας (π.χ αποθήκευση, επεξεργασία, εύρος ζώνης ή λογαριασμούς ενεργών χρηστών). Η χρήση δε των πόρων παρακολουθείται, ελέγχεται και καταγράφεται διασφαλίζοντας έτσι διαφάνεια τόσο για τον πάροχο όσο και για το χρήστη μιας συγκεκριμένης υπηρεσίας.

Μοντέλα Παροχής Υπηρεσιών

Η παροχή των υπηρεσιών από ένα Cloud μπορεί να κατηγοριοποιηθεί σε διαφορετικά μοντέλα, τα οποία αφαιρετικά συνοψίζονται σε 3 βασικά, σύμφωνα με το NIST (National Institute of Standards and Technology) [1] :

Infrastructure as a service (IaaS): Παρέχεται η δυνατότητα στο χρήστη να υλοποιήσει και να εκτελέσει *τυχαίο* λογισμικό στους πόρους που έχει δεσμεύσει. Το λογισμικό αυτό μπορεί να είναι είτε ολόκληρο Λειτουργικό Σύστημα είτε κάποια εφαρμογή. Ο χρήστης μπορεί να μην διαχειρίζεται τη βαθύτερη υποδομή του νέφους (πχ απευθείας πρόσβαση στο υλικό) αλλά έχει τον έλεγχο του ΛΣ, του αποθηκευτικού χώρου και των υπό εκτέλεση εφαρμογών καθώς και σε ορισμένες περιπτώσεις περιορισμένο έλεγχο συσκευών / υπηρεσιών δικτύου (π.χ τείχη προστασίας κλπ)

Platform as a service (PaaS): Παρέχεται η δυνατότητα στο χρήστη να εκτελέσει έτοιμες εφαρμογές ή εφαρμογές που ο ίδιος δημιούργησε χρησιμοποιώντας τις βιβλιοθήκες, τα εργαλεία και τις υπηρεσίες που ο ίδιος ο πάροχος του PaaS του παρέχει. Όπως και στο IaaS ο χρήστης δεν έχει έλεγχο της βαθύτερης υποδομής του νέφους. Σε αντίθεση όμως με το παραπάνω μοντέλο τα δικαιώματα του χρήστη περιορίζονται μόνο στις υπό εκτέλεση εφαρμογές και πιθανώς στις ρυθμίσεις αυτής χωρίς όμως να του δίνεται η δυνατότητα να διαχειριστεί το ΛΣ, το δίκτυο, τους εξυπηρετητές ή τον αποθηκευτικό χώρο.

Software as a service (SaaS): Η δυνατότητα που παρέχεται στο χρήστη είναι να χρησιμοποιεί τις εφαρμογές που εκτελεί ο πάροχος της υπηρεσίας. Η πρόσβαση στις εφαρμογές

γίνεται από τον πελάτη μέσω μιας διεπαφής όπως ο φυλλομετρητής (π.χ web-based email) ή μέσω ενός προγράμματος (π.χ OpenVPN). Σε αντίθεση με τα παραπάνω μοντέλα τα δικαιώματα του χρήστη δεν περιλαμβάνουν κανέναν έλεγχο σε οποιοδήποτε επίπεδο εκτός ίσως από περιορισμένες ατομικές ρυθμίσεις που αφορούν την εκτέλεση της εφαρμογής.

2.2.3 Internet Of Things

To Internet Of Things (IoT) αναφέρεται στη δικτύωση φυσικών συσκευών, οχημάτων, οικιακών συσκευών και γενικότερα οποιουδήποτε αντικειμένου με κάποιο ενσωματωμένο ηλεκτρονικό κύκλωμα, λογισμικό, αισθητήρα ή γενικότερα κάποια συσκευή με τη δυνατότητα μετουσίωσης κάποιας πληροφορίας από το περιβάλλον σε μία έγκυρη αναπαράστασή της στο ψηφιακό κόσμο. Κοινό στοιχείο αυτών των συσκευών είναι η δυνατότητα συνδεσιμότητάς τους επιτρέποντάς τους έτσι να ανταλλάσουν δεδομένα και να δημιουργούν την ευκαιρία μιας άμεσης απεικόνισης στοιχείων του φυσικού κόσμου σε υπολογιστικά συστήματα, οδηγώντας στη βελτίωση της αποτελεσματικότητάς τους, με ενδεχόμενα οικονομικά οφέλη καθώς και στη μειωμένη ανάγκη του ανθρώπινου παράγοντα/προσπάθειας. Από τα παραπάνω μπορούμε να κρατήσουμε δύο εμφανή χαρακτηριστικά ενός IoT Συστήματος, δηλαδή την ύπαρξη συσκευών/αισθητήρων και της δυνατότητας συνδεσιμότητάς τους. Υπάρχουν ωστόσο ακόμη δύο τουλάχιστον εξίσου σημαντικά χαρακτηριστικά. Η τύχη (αποθήκευση, επεξεργασία κλπ) όλων αυτών των δεδομένων καθώς το User Interface (UI). Υπάρχουν διαφορετικές προσεγγίσεις όσον αφορά το πού αποθηκεύονται τα δεδομένα καθώς και το πού γίνεται η επεξεργασία τους. Σε γενικές γραμμές τα δεδομένα στέλνονται στο Cloud όπου και γίνεται η αποθήκευση και περαιτέρω επεξεργασία και ανάλυσή τους. Ωστόσο σήμερα υπάρχει έντονα η τάση αποκέντρωσης των παραπάνω ενεργειών από το κεντρικό σύστημα επεξεργασίας τους *σπρώχνοντας* στο βαθμό του εφικτού και αποδοτικού διάφορες σχετικές προεργασίες των δεδομένων προς τις *άκρες* του συστήματος (όπου άκρες οι πιο απομακρυσμένοι κόμβοι από το κεντρικό Cloud όπως για παράδειγμα ένας ενδιάμεσος κόμβος στο δίκτυο κατά την πορεία των δεδομένων) ή ακόμα και μία άλλη συσκευή κατάλληλη για μια συγκεκριμένη εργασία επί των δεδομένων (FPGA, PLC κλπ).

Αναφερόμαστε φυσικά στις παρακάτω προσεγγίσεις:

- Fog Computing: Η νοημοσύνη *σπρώχνεται* στο επίπεδο του τοπικού δικτύου και τα δεδομένα περνούν από κάποια επίπεδα επεξεργασίας σε ένα fog node ή σε μια πύλη του IoT.

- Edge Computing: Η νοημοσύνη, η επεξεργαστική δύναμη και οι δυνατότητες επικοινωνίας *σπρώχνονται* από μια ακριανή πύλη ή συσκευή απευθείας σε άλλες συσκευές όπως FPGAs, PACs κλπ)

2.3 Περιβάλλοντα Εκτέλεσης Εφαρμογών

Οι παραπάνω έννοιες αφορούν τις διαφορετικές επιλογές που έχει ένας χρήστης όσον αφορά το *πού* θα επιλέξει να εκτελεί τις εφαρμογές του. Τα κριτήρια επιλογής που μέχρι

στιγμής συναντήσαμε είναι κυρίως το κόστος των απαιτούμενων πόρων ως συνάρτηση της ταχύτητας, της ασφάλειας των δεδομένων και της προσβασιμότητας καθώς επίσης και το απαιτούμενο επίπεδο τεχνογνωσίας. Ωστόσο σε κάθε μία από τις παραπάνω περιπτώσεις υπάρχουν διαφορετικές επιλογές ως προς την πλατφόρμα / αρχιτεκτονική *πάνω* στην οποία θα εκτελεστεί μια εφαρμογή. Έχει λοιπόν νόημα να απαντήσουμε και στην ερώτηση: *πώς* εκτελείται μία εφαρμογή; Ποιες είναι οι διαφορετικές *πλατφόρμες* που παρέχουν ένα περιβάλλον εκτέλεσης για μια εφαρμογή; Ποιες δυνατότητες μας παρέχει η καθεμία και ποιες ιδιαίτερες ανάγκες έρχονται να καλύψουν καλύτερα ή χειρότερα έναντι των υπολοίπων;

2.3.1 Standalone

Η πιο απλή περίπτωση είναι η απλή εκτέλεση μιας εφαρμογής σε μία υπολογιστική μονάδα με κάποιο Λειτουργικό Σύτημα (Linux, Windows, MacOS κλπ) αξιοποιώντας τα components του (είσοδος/έξοδος δεδομένων, χρονοδρομολόγηση, δίκτυο κλπ) ανάλογα με τις απαιτήσεις της ίδιας της εφαρμογής. Πιο επιγραμματικά μπορούμε να αναφερθούμε στην περίπτωση του Linux (στη φιλοσοφία του οποίου κινείται και το θέμα αυτής της εργασίας).

Μία εφαρμογή εκτελείται στο user space με απλά ή προχωρημένα δικαιώματα όσον αφορά τις επιμέρους λειτουργίες της (read, write κλπ) επί συγκεκριμένων components του Λειτουργικού Συστήματος. Ανάλογα με τις ειδικότερες λειτουργίες που μια εφαρμογή μπορεί να απαιτεί (πρόσβαση στο δίκτυο, πρόσβαση στη μνήμη του συστήματος κλπ) επικοινωνεί τελικά με το υλικό που υλοποιεί τις αντίστοιχες λειτουργίες (προσαρμογέας δικτύου, μνήμη RAM / δίσκος αντίστοιχα κλπ) από το χώρο εκτέλεσής της (user space) μέσω system calls που σηματοδοτούν το *πέρασμα* του ελέγχου (context switch) από το user space στο kernel space. Ο πυρήνας εκτελεί την εν λόγω επί μέρους λειτουργία της εφαρμογής (που εμπλέκει δηλαδή με κάποιο τρόπο το φυσικό υλικό) αξιοποιώντας λογισμικό (drivers) που διασφαλίζουν τη σωστή / έγκυρη πρόσβαση στο υλικό. Στη συνέχεια επιστρέφει τον έλεγχο στο χώρο χρήστη (και φυσικά τα αποτελέσματα της λειτουργίας που *καλέστηκε* να εκτελέσει) όπου και συνεχίζεται η εκτέλεση της εφαρμογής μέχρι το επόμενο αναγκαίο context switch.

Στην παραπάνω ροή εκτέλεσης μιας εφαρμογής αξίζει να αναλύσουμε τα ακόλουθα χαρακτηριστικά:

- Ο πυρήνας του Λειτουργικού Συστήματος είναι ουσιαστικά ένα σύνολο ρουτινών που επιτρέπουν σε μία εφαρμογή να εκτελείται σε μια συγκεκριμένη πλατφόρμα.

- Το system call αποτελεί ένα interface μεταξύ της εφαρμογής και του kernel space. Με απλά λόγια είναι ο τρόπος που μία εφαρμογή μπορεί τελικά να καλεί και να χρησιμοποιεί τα χαρακτηριστικά ενός υπολογιστικού συστήματος. Η κλήση τους σηματοδοτεί την εκτέλεση κώδικα που *ανήκει* στον πυρήνα του Λειτουργικού Συστήματος. Αξίζει να σημειωθεί ότι η κλήση τους γίνεται με καλυμμένο τρόπο αξιοποιώντας βιβλιοθήκες συστήματος (wrappers πχ στην libc), διασφαλίζοντας έτσι και ότι μία εφαρμογή μπορεί να εκτελείται σε διαφορετικές εκδόσεις του Λειτουργικού Συστήματος (ή ακόμα και σε διαφορετικά Λειτουργικά Συστήματα) χωρίς αλλαγές στον κώδικά της, αφού η κλήση γίνεται με γενικό τρόπο και τελικά η λειτουργία είναι υπόθεση του πυρήνα. Έχει σημασία

να αντιληφθούμε ότι ακριβώς επειδή τα system calls και οι λειτουργίες που απαιτεί μία εφαρμογή από το Λειτουργικό επιδιώκεται να είναι ανεξάρτητες πλατφόρμας, υλικού και αρχιτεκτονικής, **δημιουργείται τελικά η δυνατότητα αποσύνθεσης του Λειτουργικού Συστήματος από ένα σύνολο λειτουργιών / ρουτινών / εξαρτημάτων και η αντιμετώπισή του ως μία επιλογή ορισμένων μόνο μελών του παραπάνω συνόλου, αναγκαίων για την εκτέλεση της συγκεκριμένης εφαρμογής.** Μπορούμε να πούμε δηλαδή ότι υπάρχει η δυνατότητα να ορίσουμε το λειτουργικό με βάση τις ανάγκες μιας εφαρμογής σε αντίθεση με την επικρατούσα προσέγγιση που μάλλον βασίζεται στην ακριβώς αντίθετη φιλοσοφία.

Οι παράγοντες που επηρεάζουν την εκτέλεση μιας εφαρμογής όσον αφορά την ταχύτητα και την ασφάλεια είναι άμεσο επακόλουθο των παραπάνω. Αν εστιάσουμε λοιπόν μόνο στην εκτέλεση μιας εφαρμογής κάνοντας την παραδοχή ότι ολόκληρο το σύστημα υπάρχει μόνο για την εφαρμογή μας και θα έπρεπε να είναι tailored σε αυτή τότε μπορούμε να ξεχωρίσουμε τους παρακάτω επιβαρυντικούς παράγοντες για την υπόθεσή μας:

- Η συνεχής εναλλαγή από το user space σε kernel space (context switch)
- Η ποσότητα μνήμης που δεσμεύεται από χαρακτηριστικά του λειτουργικού (network, filesystem κλπ) που μπορεί τελικά να μην χρησιμοποιούνται από την εφαρμογή
- Η ύπαρξη των ίδιων *άχρηστων* (για την εφαρμογή πάντα) χαρακτηριστικών αυξάνει τις γραμμές κώδικα που εκτελούνται άρα και τις πιθανές *τρύπες* ασφαλείας του όλου συστήματος.
- Οι υπόλοιπες εφαρμογές που εκτελεί παράλληλα το λειτουργικό για λειτουργίες άσχετες με την εφαρμογή που μας ενδιαφέρει, έχουν επίσης το μερίδιό τους σε CPU time.

2.3.2 Platform Virtualization

Η εικονοποίηση της πλατφόρμας ή του υλικού (Platform / Hardware Virtualization) αναφέρεται στη δημιουργία Virtual Machines (VMs) που προσομοιάζουν σε έναν υπαρκτό υπολογιστή με (συνήθως) πλήρες Λειτουργικό Σύστημα (πυρήνα). Σε σύνδεση με την προηγούμενη παράγραφο, μπορούμε να φανταστούμε ένα εικονικό hardware πάνω στο οποίο τρέχει το Linux και μια εφαρμογή standalone. Φυσικά τόσο το Λειτουργικό Σύστημα του guest machine όσο και η εφαρμογή δεν έχουν πλέον άμεση σχέση με το πραγματικό hardware του host machine.

Η δημιουργία και διαχείριση του εικονικού hardware γίνεται από ένα προγραμμα hypervisor που τρέχει πάνω στο πραγματικό hardware. Ο hypervisor είναι σε θέση να προσομοιώνει πολλαπλά virtual machines πλήρως απομονωμένα μεταξύ τους. Θα αναφερθούμε στα 3 βασικά είδη platform virtualization πάντα υπό το πρίσμα της εκτέλεσης μια εφαρμογής στην κάθε πλατφόρμα[2]:

- Emulation: η εικονική μηχανή προσομοιώνει εξολοκλήρου μία αρχιτεκτονική υλικού, πιθανώς διαφορετική από το πραγματικό υποκείμενο υλικό, επιτρέποντας έτσι να εκ-

τελεστεί επάνω της ένα μη τροποποιημένο, φιλοξενούμενο ΛΣ σχεδιασμένο για τον εξομοιούμενο επεξεργαστή (π.χ. QEMU, έκδοση για PowerPC του VirtualPC κλπ).

- Full virtualization: η εικονική μηχανή προσομοιώνει επαρκές τμήμα του πραγματικού hardware ώστε να επιτρέπει την εκτέλεση επάνω της ενός μη τροποποιημένου, guest OS σχεδιασμένου για τον ίδιο τύπο επεξεργαστή με την πραγματική CPU (π.χ. VirtualPC, VMware, Win4Lin κλπ). Στην πλήρη εικονικοποίηση δεν χρειάζεται εξομοίωση του συνόλου εντολών του επεξεργαστή και μάλιστα ένα τμήμα του κώδικα του guest OS μπορεί να εκτελείται απευθείας από το υλικό, χωρίς μεσολάβηση του επόπτη, αρκεί να μην επηρεάζει υποσυστήματα εκτός του άμεσου ελέγχου του τελευταίου. Τα κρίσιμα σημεία του φιλοξενούμενου κώδικα ωστόσο, όπως αυτά που προσπαθούν να αποκτήσουν πρόσβαση στο υλικό (π.χ. System calls), συλλαμβάνονται από το hypervisor και προσομοιώνονται, αφού τα αποτελέσματα κάθε λειτουργίας που επιτελείται σε ένα VM δεν επιτρέπεται να τροποποιούν την κατάσταση άλλων VM, του hypervisor ή του hardware. Αν το πραγματικό hardware βοηθά και επιταχύνει τη λειτουργία του hypervisor τότε η πλήρης εικονικοποίηση ονομάζεται εγγενής (native) (πχ KVM). Η βοήθεια αυτή αφορά κυρίως εύκολη διάκριση μεταξύ εντολών που μπορούν να εκτελεστούν απευθείας και εντολών που πρέπει να προσομοιωθούν από το λογισμικό. Όπως και στην εξομοίωση το VM παρέχει στο guest OS μία αφαίρεση της μνήμης, των συσκευών Εισόδου / Εξόδου κλπ, ενώ η εγγενής εκτέλεση μεγάλου μέρους του κώδικα παρέχει πολύ καλύτερες επιδόσεις σε σχέση με την εξομοίωση.

- Paravirtualization: Το VM δεν προσομοιώνει επακριβώς το hardware αλλά παρέχεται ένα API, μία προγραμματιστική διασύνδεση, ώστε να επιτρέπει την εκτέλεση επάνω του ενός τροποποιημένου, guest OS σχεδιασμένου για εκτέλεση από τον συγκεκριμένο επόπτη (π.χ. XEN). Το προαναφερθέν API ονομάζεται διασύνδεση υπερκλήσεων (hypercall interface) και ένα Λειτουργικό Σύστημα πρέπει να μεταφερθεί ρητά σε έκδοση κατάλληλη για εκτέλεση από ένα σύστημα παραεικονικοποίησης, ώστε ο guest πυρήνας αντί να προσπαθεί να προσπελαύνει το hardware άμεσα να εκτελεί hypercalls και να αναμένει απαντήσεις ή ασύγχρονες ειδοποιήσεις από το hypervisor. Το όφελος από τη βελτίωση των επιδόσεων και την απλοποίηση της γραφής του hypervisor είναι μεγάλο.

Μία άλλου είδους κατηγοριοποίηση των hypervisor είναι σε αυτούς που εκτελούνται ως εφαρμογές πάνω από το host σύστημα (Type-2 ή hosted hypervisors π.χ. VirtualPC, VMware κλπ) και σε αυτούς που λειτουργούν οι ίδιοι ως λιτά Λειτουργικά Συστήματα και άρα εκτελούνται απευθείας πάνω από το υποκείμενο υλικό (Type-1, native ή bare-metal hypervisors π.χ. Xen). Ιδιαίτερα, στην περίπτωση του Kernel-based Virtual Machine (KVM) αναφερόμαστε σε μία δομή (module) που περιλαμβάνεται στον πύρηνα του Linux και τον μετατρέπει σε type-1 hypervisor· καθώς όμως το Linux παραμένει ένα Λειτουργικό γενικού σκοπού όπου και άλλες εφαρμογές *διεκδικούν* εικονικούς πόρους μπορούμε ταυτόχρονα να το κατηγοριοποιήσουμε και ως hosted hypervisor.

2.3.3 Containerization

Το containerization αποτελεί ουσιαστικά μία εικονικοποίηση σε επίπεδο Λειτουργικού Συστήματος. Για να γίνουμε πιο παραστατικοί από την πλευρά του Λειτουργικού Συστήματος ένας container είναι ένας ιδιόμορφος απομονωμένος user space ενώ από την πλευρά της containerized εφαρμογής είναι ένας πραγματικός υπολογιστής. Τόσο στην περίπτωση που μια εφαρμογή εκτελείται standalone σε κάποιο πραγματικό hardware με κάποιο Λειτουργικό Σύστημα όσο και στην περίπτωση που εκτελείται σε κάποιο εικονικό hardware με κάποιο Λειτουργικό Σύστημα πάνω απο ένα hypervisor, μπορεί να *δει* όλους τους πόρους (πραγματικούς ή εικονικούς αντίστοιχα) αυτού του συστήματος, ή για να είμαστε πιο ακριβείς μπορεί να έχει πρόσβαση σε οτιδήποτε περιλαμβάνεται στο user space του Λειτουργικού Συστήματος. Στην περίπτωση που μια εφαρμογή εκτελείται *μέσα* σε ένα container μπορεί να δει μόνο τα περιεχόμενα αυτού και τις συσκευές που έχουν ανατεθεί σε αυτό. Σε μια αντιστοιχία με πριν δηλαδή η εφαρμογή αντιλαμβάνεται ότι εκτελείται σε ένα user space του οποίου οι διαθέσιμοι πόροι είναι τα περιεχόμενα του container.

2.4 Unikernels

2.4.1 Εισαγωγή - Ιστορία

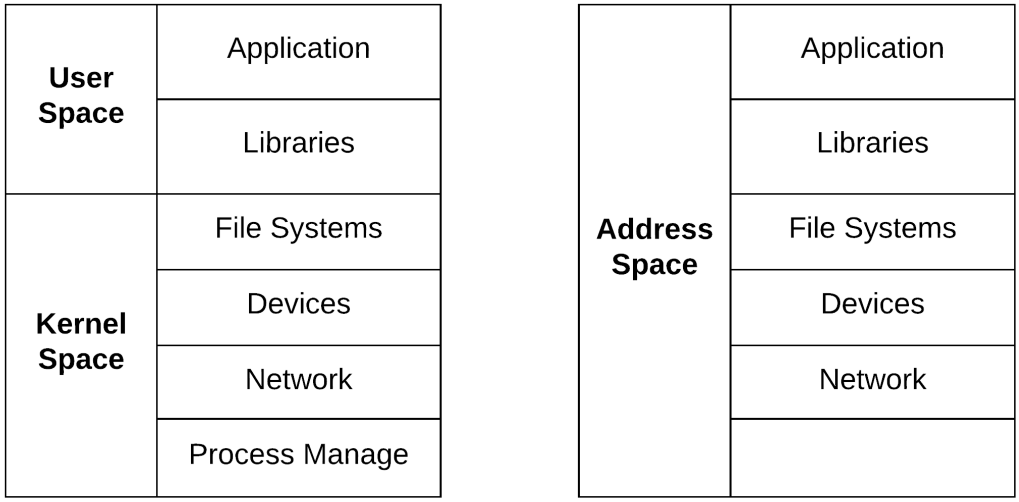
Τα Unikernels είναι εικόνες μηχανών (κατά το virtual machice) οι οποίες διαθέτουν ενιαίο χώρο διευθύνσεων και έχουν κατασκευαστεί χρησιμοποιώντας το ελάχιστο σύνολο, από τις βιβλιοθήκες ενός κλασικού Λειτουργικού Συστήματος, που απαιτεί η εκτέλεση κάποιας συγκεκριμένης εφαρμογής. Οι βιβλιοθήκες του Λειτουργικού Συστήματος γίνονται compile μαζί με τον κώδικα της εφαρμογής και τις ρυθμίσεις της ως μια ενιαία εικόνα ή καλύτερα εκτελέσιμο, που *τρέχει* τελικά πάνω από έναν hypervisor ή / και απευθείας στο hardware χωρίς να μεσολαβεί ένα ολόκληρο Λειτουργικό Σύτημα. Με λίγα λόγια οι Unikernels είναι μικρά, γρήγορα και ασφαλή virtual machines χωρίς Λειτουργικό Σύτημα.

Μπορεί η τεχνολογία των Unikernels να είναι σχετικά καινούργια, ωστόσο η φιλοσοφία τους, η αντίληψη δηλαδή του Λειτουργικού ως η επιλογή των απαραίτητων μόνο components από μία δεξαμενή βιβλιοθηκών εντοπίζεται σε αρχιτεκτονικές που είχαν αναπτυχθεί ήδη από το 1990: Library Operating Systems (libOS)[3]. Σε μία τέτοια αρχιτεκτονική τα όρια προστασίας (protection boundaries) εκτοπίζονται πιο κοντά στο επίπεδο του υλικού. Σε αντίθεση με ένα παραδοσιακό Λειτουργικό Σύστημα όπου η πρόσβαση στο hardware και στο δίκτυο παρέχονται ως services στο user space, στην περίπτωση των libOS οι αντίστοιχες λειτουργίες υλοποιούνται ως:

- ένα σύνολο βιβλιοθηκών που υλοποιούν μηχανισμούς καθοδήγησης του υλικού, πρόσβασης στο δίκτυο κλπ
- ένα σύνολο πολιτικών που επιβάλλουν το access control και isolation στο application layer.

Η βασική λοιπόν διαφορά στην εκτέλεση μιας εφαρμογής είναι ότι έχει απευθείας πρόσβαση στο hardware χωρίς να απαιτούνται επαναλαμβανόμενα privilege transitions και μεταφορά δεδομένων μεταξύ user και kernel space. Ταυτόχρονα η έλλειψη μιας κεντρικές υπηρεσίας δικτύου σημαίνει και ότι δεν υπάρχει διαχωρισμός μεταξύ πακέτων υψηλής και χαμηλής προτεραιότητας: οι εφαρμογές που τρέχουν ως libOS έχουν δηλαδή ανεξάρτητες ουρές προτεραιότητας και τα πακέτα τελικά *ανακατεύονται* μόνο στο επίπεδο της συσκευής δικτύου. Τα παραπάνω δύο χαρακτηριστικά αποτελούν και δύο σημαντικούς παράγοντες στο execution time και αντίστοιχα στο latency μίας εφαρμογής, και όπως είναι λογικό χαρακτηρίζουν και τους Unikernels ως παραγόμενη τεχνολογία.

2.4.2 Μία εξήγηση της θεωρίας



(a) Linux Application Stack (b) Unikernel Application Stack

Figure 2.1: Linux and Unikernel Stack

Αν σκεφτούμε τους Unikernel ως μία πιο απλοποιημένη στοίβα μιας εφαρμογής στο Linux (application stack) γίνεται αρκετά πιο εύκολη η κατανόηση της λειτουργίας τους. Στην περίπτωση του Linux Application Stack (Σχ. 2.1a) υπάρχει σαφής διαχωρισμός μεταξύ του user address space όπου και εκτελείται η εφαρμογή χρησιμοποιώντας κοινόχρηστες βιβλιοθήκες (που υλοποιούν και τη διεπαφή) και του kernel space όπου υλοποιούνται και εκτελούνται και οι λειτουργίες πρόσβασης στα devices και στο δίκτυο. Παράλληλα ο πυρήνας αναλαμβάνει τη διαχείριση των εφαρμογών που εκτελούνται (process management). Αντίθετα στην περίπτωση του Unikernel Application Stack (Σχ. 2.1b) δεν υπάρχει διαχωρισμός μεταξύ user και kernel space. Ο χώρος διευθύνσεων είναι ενιαίος. Υπάρχει μόνο ένα πρόγραμμα που εκτελείται το οποίο περιέχει τα πάντα από το υψηλότερο επίπεδο της ίδιας της εφαρμογής μέχρι τις ρουτίνες του πιο χαμηλού επιπέδου για την πρόσβαση στις συσκευές και το δίκτυο. Η εφαρμογή είναι μια ενιαία εικόνα που δε χρειάζεται τίποτα περισσότερο από τον

εαυτό της για να εκκινήσει και να εκτελέσει όλες τις λειτουργίες της.

Το γεγονός ότι όλες οι ρουτίνες και οι λειτουργίες που απαιτεί η εφαρμογή αποτελούν και *μέρος* της ενώ σε ένα παραδοσιακό σενάριο εκτέλεσής της, θα της παρέχοταν στο runtime από το Λειτουργικό Σύστημα, δημιουργεί ένα εύλογο ερώτημα ως προς το χρόνο ανάπτυξής της με την έννοια των πρόσθετων λειτουργιών που πρέπει να προγραμματιστούν. Ωστόσο όπως θα δούμε και παρακάτω (Κεφάλαιο 3) εξετάζοντας τα υπάρχοντα Unikernel Frameworks παρέχονται στο χρήστη τα απαραίτητα εργαλεία και περιβάλλοντα ώστε οι εφαρμογές που θα επιλέξει να αναπτύξει ως Unikernels να κάνουν κατά το compile time ότι τα *παραδοσιακά* προγράμματα κάνουν στο runtime.

Τα περισσότερα Unikernel Frameworks λοιπόν διαθέτουν ένα ειδικό compiling system που προσφέρει τις χαμηλότερου επιπέδου ρουτίνες που ο προγραμματιστής θα επιλέξει για να τρέξει η εφαρμογή - Unikernel. Ο κώδικας για αυτές τις ρουτίνες γίνεται compile απευθείας μαζί με το εκτελέσιμο της εφαρμογής μέσω ενός Library Operating System, μιας συλλογής δηλαδή από βιβλιοθήκες που προσφέρουν τις απαραίτητες ρουτίνες ενός Λειτουργικού Συστήματος σε μορφή ικανή για compile. Το αποτέλεσμα τελικά είναι η εκτελέσιμη εικόνα ενός προγράμματος που περιέχει οτιδήποτε χρειάζεται για να εκτελεστεί. Δε χρειάζεται καμία κοινόχρηστη βιβλιοθήκη ή κάποιο Λειτουργικό Σύστημα, είναι ένα πλήρως αυτόνομο πρόγραμμα που μπορεί να τρέξει ως ένα virtual machine και να εκκινήσει επιτυχώς.[4]

2.4.3 Development stages

Θα μπορούσαμε να χωρίσουμε τη διαδικασία ανάπτυξης των Unikernel σε τρεις φάσεις οι οποίες γενικά ισχύουν αφαιρετικά στα περισσότερα frameworks (τουλάχιστον σε οσα μελετήσαμε στα πλαίσια αυτής της εργασίας):

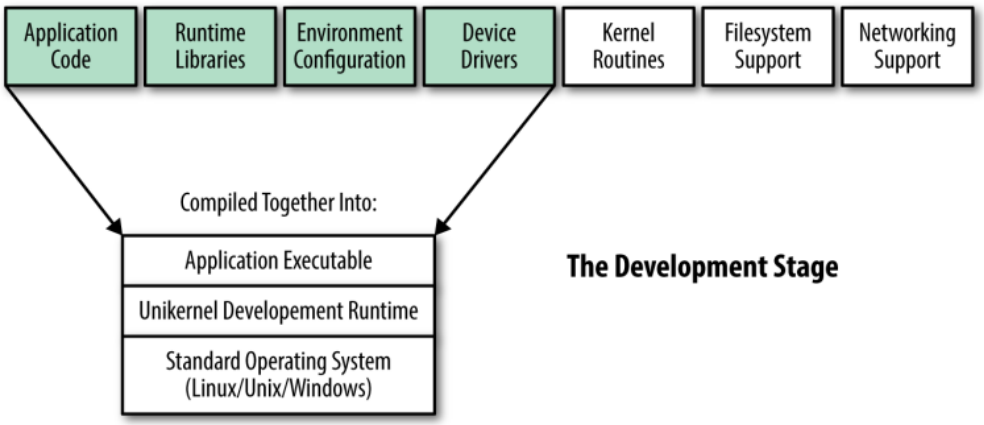


Figure 2.2: Unikernel Development Stage [4]

Κατά τη φάση του **development** (Σχ. 2.2) η εφαρμογή γίνεται compiled όπως θα γινόταν εάν την παράγαμε (deploy) στοχεύοντας μία κλασική στοίβα λογισμικού. Όλες τις λειτουργίες που κανονικά θα σχετιζόταν με λειτουργίες του πυρήνα, τις χειρίζεται όντως ο πυρήνας του μηχανήματος στο οποίο γίνεται η ανάπτυξη της εφαρμογής, όπως θα περίμενε

κανείς σε μια παραδοσιακή στοίβα λογισμικού. Αυτό σημαίνει ότι σε αυτή τη φάση είναι διαθέσιμα όλα τα παραδοσιακά εργαλεία ανάπτυξης λογισμικού (debuggers κλπ). Η ανάπτυξη σε αυτό το στάδιο λοιπόν δεν είναι πιο περίπλοκη από ότι θα ήταν υπο *φυσιολογικές* συνθήκες.

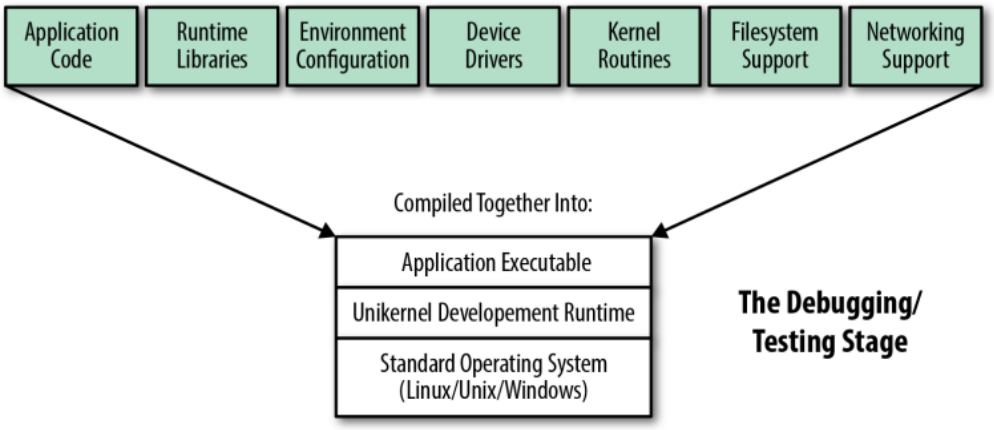


Figure 2.3: Unikernel Testing Stage [4]

Κατά τη φάση του **testing** (Σχ. 2.3) ο compiler προσθέτει στην παραγόμενη εικόνα τις λειτουργίες που σχετίζονται με τις δραστηριότητες του πυρήνα. Τα συνήθη εργαλεία είναι διαθέσιμα όπως και πριν. Η κύρια διαφορά είναι ότι ο compiler *βάζει* στην παραγόμενη εικόνα βιβλιοθήκες από το user space ώστε το testing να γίνεται χωρίς να βασίζεται στις βιβλιοθήκες του Λειτουργικού Συστήματος που το φιλοξενεί.

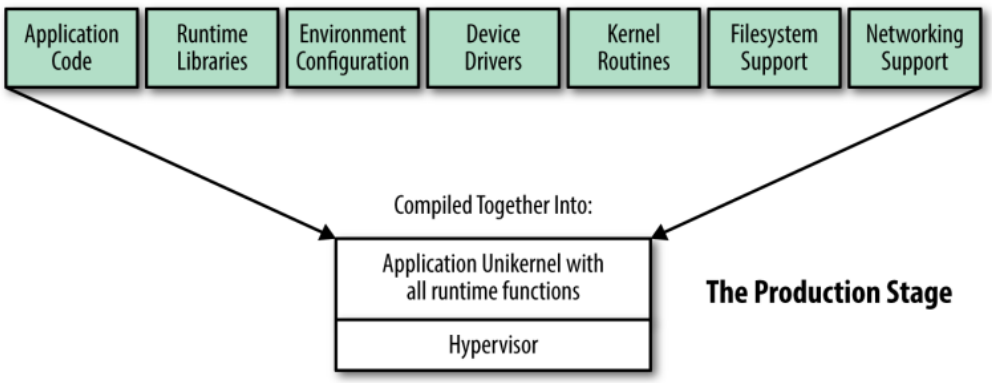


Figure 2.4: Unikernel Production Stage [4]

Στο στάδιο του **production** (Σχ. 2.4) η εικόνα είναι έτοιμη ως ένας λειτουργικός Unikernel περιέχοντας πλέον όλες τις απαραίτητες βιβλιοθήκες για να λειτουργήσει σαν αυτόνομη εικονική μηχανή.

2.4.4 Πλεονεκτήματα

Μικρό μέγεθος: Οι Unikernels είναι αισθητά πολύ μικροί σε μέγεθος και ανάλογα τις λειτουργίες που προσφέρουν μπορεί να κυμαίνονται από μερικές εκατοντάδες KB έως λίγα MB. Αποτελώντας τη σύνθεση των απολύτως απαραίτητων εξαρτήσεων μιας εφαρμογής για να εκτελεστεί αυτόνομα ως VM, εξαλείφοντας κάθε τι περιττό το μέγεθος των Unikernels κάνει ακόμα και τα πιο minimal VM που προσφέρουν διάφοροι Cloud πάροχοι να μοιάζουν τεράστια.

Ταχύτητα: Η εξάλειψη περιττών υπηρεσιών και δομών που ένα παραδοσιακό VM θα προσέφερε, έχει ως αποτέλεσμα και την πολύ γρήγορη εκκίνηση (boot) των Unikernels. Αναφορές στη σχετική βιβλιογραφία δείχνουν ότι το boot time κυμαίνεται στην κλίμακα των milliseconds. Το γεγονός αυτό αποτελεί μεγάλο πλεονέκτημα στο χώρο του Cloud καθώς σημαίνει ότι μια υπηρεσία μπορεί να δημιουργείται άμεσα κατά τη ζήτησή της και αντίστοιχα να τερματίζει εξίσου άμεσα όταν πλέον δεν χρησιμοποιείται ελευθερώνοντας και τους σχετικούς πόρους που θα είχε δεσμεύσει.

Ασφάλεια / Απομόνωση: Μειώνοντας τις γραμμές κώδικα, τις βιβλιοθήκες, τις υπηρεσίες, τα component που θα είχε ένα πλήρες Λειτουργικό Σύτημα είναι φυσικό να μειώνονται και οι πιθανοί στόχοι επίθεσης ενός κακόβουλου χρήστη. Από τη φύση τους οι Unikernels είναι πιο ασφαλείς καθώς συνδυάζουν την απομόνωση που προσφέρει η εκτέλεση *πάνω* από τον hypervisor και την απαλοιφή όλων των *άχρηστων* components πιθανών στόχων (για παράδειγμα το VENOM (security vulnerability)[5]).

2.5 ARM Devices

Η τεχνολογία επεξεργαστών ARM (παραδοσιακά Acorn RISC Machines και πλέον Advanced RISC Machines) έχουν εξελιχθεί τα τελευταία χρόνια ως η επικρατέστερη επιλογή στη σχεδίαση μικροεπεξεργαστών και ενσωματωμένων συσκευών. Όπως υποδηλώνεται και από τα αρχικά τους ανήκουν σε μία οικογένεια αρχιτεκτονικής που ονομάζεται RISC (Reduced Instruction Set Computer - Υπολογιστές Περιορισμένου Συνόλου Εντολών).

Ο όρος reduced δεν χαρακτηρίζει τις εντολές ως αριθμό αλλά προορίζεται για να περιγράψει το γεγονός ότι οι εντολές αυτές είναι απλές και απαιτούν λιγότερους κύκλους ρολογιού (Clock Cycles) για να εκτελεστούν σε αντίθεση με την έτερη τεχνολογία CISC (Complex Instruction Set Computer - Υπολογιστές Πολύπλοκου Συνόλου Εντολών) όπου οι εντολές είναι πιο σύνθετες.

Οι ARM επεξεργαστές αξιοποιώντας ακριβώς αυτή την αρχιτεκτονική έχουν μια σειρά από ιδιαίτερα χαρακτηριστικά:

- Χαμηλή κατανάλωση Ενέργειας (low power) : Οι απλοποιημένες εντολές αφού εκτελούν λιγότερες επιμέρους λειτουργίες απαιτούν μικρότερο αριθμό transistors, επομένως καταναλώνουν τελικά και λιγότερη ενέργεια.

- Μικρότερο μέγεθος: Καθώς η υλοποίηση των εντολών τους απαιτεί μικρότερο αριθμό

transistors τελικά οδηγούμαστε σε περισσότερο *χώρο* ψηφίδας (chip space) και σε μικρότερα chips.

- Load/Store Αρχιτεκτονική: Οι εντολές είναι χωρισμένες σε δύο κατηγορίες: α) πρόσβασης στη μνήμη (memory access) και β) λειτουργιών αριθμητικής / λογικής μονάδας (Arithmetic Logic Unit (ALU) operations). Στην προσέγγιση αυτή η δεύτερη κατηγορία εντολών απαιτεί οι τελεστές να είναι registers. Το αποτέλεσμα είναι ότι κάθε απλή εντολή απαιτεί συνήθως το πολύ ένα κύκλο πρόσβασης στη μνήμη (single data memory cycle).

Φυσικά οι απλοποιημένες εντολές έχουν και ένα trade off: απαιτούνται περισσότερες εντολές για μία εργασία (task) αυξάνοντας τις ανάγκες σε μνήμη (memory consumption) και ενδεχομένως το συνολικό χρόνο εκτέλεσης της εργασίας (execution time). Ωστόσο τα χαρακτηριστικά που περιγράψαμε παραπάνω αναπληρώνουν το κόστος αυτό καθώς το μικρότερο μέγεθος και κατανάλωση (και κόστος παραγωγής τελικά) επιτρέπουν στους ARM να υλοποιούν περισσότερους πυρήνες (cores), περισσότερες περιφερειακές μονάδες επεξεργασίας (peripherals) και μονάδες επεξεργασίας γραφικών (graphic processor units).

Κάθε αρχιτεκτονική ωστόσο πρέπει να κρίνεται με βάση τις ανάγκες που στοχεύει να καλύψει. Ή για να το θέσουμε πιο ορθά κάθε task ή application πρέπει να συνδυάζεται με τη σωστή αρχιτεκτονική. Εφαρμογές που χρειάζονται βαριές και ποικίλες επεξεργαστικές ανάγκες δεν είναι γενικά οι καταλληλότεροι υποψήφιοι για την εν λόγω αρχιτεκτονική.

Ωστόσο εφαρμογές που περιστρέφονται κυρίως γύρω από πιο απλές και συγκεριμενοποιημένες εργασίες με σχετικά μικρό αποτύπωμα στη μνήμη (memory footprint) και πιο *ελαφρύ* workload συνήθως μπορούν να υλοποιηθούν και από πιο απλά σύνολα εντολών καθιστώντας τες ιδανικούς υποψηφίους για επεξεργαστές ARM. (Παράδειγμα: ένας διαδικτυακός εξυπηρετητής (Web Server) που ο κύριος ρόλος του είναι να μεταφέρει δεδομένα σερβίροντας σελίδες από κάποιο storage στο δίκτυο.)

2.5.1 Virtualization on ARM

Το virtualization φιλοξενούμενο σε πλατφόρμες ARM είναι μία σχετικά πρόσφατη εξέλιξη. Τα πρώτα σύνολα εντολών στην τεχνολογία αυτή δεν έδιναν τη δυνατότητα σε κάποιο hypervisor να διαχειρίζεται και να δεσμεύει υπολογιστικούς πόρους με αποδοτικό και ασφαλή τρόπο. Οι πρώτες υλοποιήσεις γινόταν σε επίπεδο λογισμικού έχοντας ως αποτέλεσμα σοβαρά θέματα απόδοσης. Η πρώτη σοβαρή προσπάθεια και υποστήριξη του virtualization ήρθε στη 7η οικογένεια των επεξεργαστών της (ARMv7) όπου πλέον εισήγαγε και αυτή τα σενάρια της επιτάχυνσης σε υλικό για virtualization (hardware acceleration for virtualization) ακολουθώντας τους ανταγωνιστές της.

Οι επεξεργαστές αυτής της οικογένειας ωστόσο παρέμεναν 32-bits. Ως γνωστόν με 32-bit είναι δυνατόν να διευθυνσιοδοτήσουμε μέχρι 4GB μνήμης RAM περίπου. Σε περιβάλλοντα εικονικοποίησης όμως το μέγεθος της μνήμης RAM είναι σημαντικός παράγοντας καθώς επηρεάζει το συνολικό αριθμό εικονικών μηχανών που ένα host σύστημα μπορεί να υποστηρίξει και φυσικά τους πόρους που θα διαθέσει σε κάθε μία από αυτές. Η ARM ξεπέρασε

σε κάποιο βαθμό αυτόν τον περιορισμό αρχικά αντιστοιχίζοντας τις 32-bit διευθύνσεις σε ένα εύρος 40-bit διευθύνσεων.

Η πιο πρόσφατη οικογένεια επεξεργαστών της ωστόσο (ARMv8) ξεπέρασε όλους τους παραπάνω περιορισμούς καθώς είναι 64-bit, διαθέτει μονάδες επιτάχυνσης στο υλικό για εικονικοποίηση και επιπλέον υποστηρίζεται από μία σειρά hypervisors (KVM, Xen etc) που παρέχουν μεθόδους αξιοποίησης των μηχανισμών αυτών που υπάρχουν στο υλικό. Η εξέλιξη αυτή σηματοδοτεί και την δυναμική εισαγωγή της ARM στην αγορά των servers.

2.5.2 Περιφερειακή κλιμάκωση

Όπως εξηγήσαμε και παραπάνω ένας ARM επεξεργαστής δεν μπορεί μάλλον να παρέχει την ίδια υπολογιστική ισχύ ανά πυρήνα χωριστά σε σχέση ενδεχομένως με έναν Intel (που είναι CISC). Επομένως ενώ το σενάριο του virtualization βασίζεται στην ίδια διαδικασία της δέσμευσης υπολογιστικών πόρων, ο τρόπος με τον οποίο αυτοί γίνονται scale είναι διαφορετικός. Ένας παραδοσιακός x86 server με πολλαπλούς επεξεργαστές κλιμακώνει τα απαιτούμενα resources κεντρικά (scale up) αυξάνοντας ενδεχομένως την ταχύτητα του επεξεργαστή ώστε να διαχειρίζεται τους απαιτούμενους φόρτους εργασίας. Αντιθέτως ένας ARM server θα χρησιμοποιούσε/συνδύαζε πολλαπλάσιους μικρότερους και λιγότερο δυνατούς (low-power) επεξεργαστές που θα μοιραζόταν ανάμεσα σε περισσότερες παραλληλες εργασίες αντί για λιγότερους δυνατότερους επεξεργαστές.Η παραπάνω παρατήρηση καθιστά τους ARM ιδανικούς υποψηφίους για πλατφόρμες που προορίζονται να τρέχουν πολλαπλές απλές εφαρμογές που απαιτούν λίγα resources, που μπορούν δηλαδή να τρέξουν εξίσου αποδοτικά δεσμεύοντας έναν ή λίγους χαμηλής κατανάλωσης και δύναμης επεξεργαστές. Ειδικότερα αν αυτές τρέχουν on demand, δεσμεύοντας και σύντομα απο-δεσμεύοντας resources ώστε αυτά να χρησιμοποιηθούν από άλλες.

Τέτοιες εφαρμογές είναι και οι Unikernels. Single address space images, με αρκετά μικρό footprint που υπολοποιούν ιδανικά απλές και κυρίως συγκεκριμένου σκοπού εργασίες. Παράλληλα επειδή έχουν πολύ μικρό boot time ταιριάζουν απόλυτα σε ένα σενάριο εκτέλεσης κατα παραγγελία και εικονικά. Τέλος, όπως αναφέρθηκε και στις προηγούμενες παραγράφους, η σύγχρονη τάση στο Cloud Computing ως κεντρική υποδομή των IoT συστημάτων, περιστρέφεται γύρω από την αποκέντρωση της επεξεργασίας των παραγόμενων δεδομένων εκτοπίζοντας την προς τις άκρες (edge) του συστήματος αλλά και σε ενδιάμεσους κόμβους του δικτύου (fog). Το ρόλο αυτό μπορούν να αναλαμβάνουν low-power devices που ζουν στη διαδρομή των δεδομένων προς το κεντρικό Cloud. Τέτοιες συσκευές μπορεί να είναι επιταχυντές υλικού υλοποιημένοι σε FPGA, Single Board Computers (SBC) που συνήθως διαθέτουν μικροεπεξεργαστές ARM (πχ Xilinx Zynq, Raspberry Pi κλπ).

Κεφάλαιο 3

Τεχνολογίες Unikernel

3.1 Εισαγωγή

Στα πλαίσια αυτής της εργασίας μελετήσαμε μερικά από τα πιο *γνωστά* Unikernel frameworks (MirageOS, Rumprun, Solo5, OSv, IncludeOS). Στο κεφάλαιο αυτό θα επιχειρήσουμε να περιγράψουμε τη λογική και τη φιλοσοφία του καθενός, ενώ στο επόμενο κεφάλαιο θα μελετήσουμε πιο αναλυτικά τη δομή ορισμένων εξ'αυτών (Rumprun, MirageOS, Solo5) στην προσπάθεια μας να εκτελέσουμε μια σειρά από πειράματα/μετρήσεις σε πλατφόρμες αρχιτεκτονικής ARM και x86_64.

3.2 Rumprun

3.2.1 Γενικά

Το Rumprun αποτελεί μία υλοποίηση των Rump Kernels που επιτρέπει την μετατροπή σχεδόν κάθε εφαρμογής που ακολουθεί POSIX (Portable Operating System Interface) [1] πρότυπο σε Unikernel.

Αρχικά το Rump Kernel project έρχεται από τον κόσμο του NetBSD. Το NetBSD αποτελεί ένα Λειτουργικό Σύστημα το οποίο έχει σχεδιαστεί με την προοπτική να είναι κατά το δυνατό modular, με σκοπό οι drivers που παρέχει να είναι εύκολα τροποποιήσιμοι στοχεύοντας πολλαπλές πλατφόρμες. Το Rump Kernel παρέχει λοιπόν (αμετάβλητους) τους drivers από το NetBSD σε μία μορφή που μπορούν να χρησιμοποιηθούν για να φτιάξουμε ελαφριές (lightweight) συγκεκριμένου σκοπού εικονικές μηχανές. Αποτελεί λοιπόν τη βάση των Rumprun Unikernels επιτρέποντας τη μετατροπή POSIX εφαρμογών σε Unikernels.

3.2.2 Rationale

Ως γνωστόν οι εφαρμογές χρειάζονται υπο-ρουτίνες για να λειτουργήσουν και αυτές οι υπο-ρουτίνες είναι διαθέσιμες μέσω του λειτουργικού συστήματος. Εφεξής θα αποκαλούμε

[1] Το POSIX αναφέρεται σε ένα σύνολο από standards (*πρότυπα*) καθορισμένα από την IEEE με σκοπό τη διατήρηση της συμβατότητας μεταξύ των διαφόρων Λειτουργικών Συστημάτων

αυτές τις υπορουτίνες drivers επομένως μπορούμε να δηλώσουμε ότι μία τυπική εφαρμογή χρειάζεται όχι μόνο ένα μεγάλο σύνολο drivers αλλά και ότι ο προγραμματισμός και η συντήρηση του κώδικά αυτών δεν είναι σε καμία περίπτωση τετριμμένες διαδικασίες. Μπορεί τα επικρατέστερα λειτουργικά συστήματα να είναι πλούσια σε drivers, καθώς ιστορικά έχουν και μια μεγάλη διάρκεια ύπαρξης, ωστόσο δεν είναι κατάλληλα για τις σύγχρονες ανάγκες. Το Rump Kernel project κάνει την εξής παραδοχή[6]:
Πρέπει να αρχίσουμε να αντιμετωπίζουμε τους drivers σαν να ήταν συστατικά βιβλιοθηκών αντί να απαιτείται μια ξεχωριστή υλοποίησή τους για κάθε λειτουργικό σύστημα. Αυτή η προσέγγιση μας επιτρέπει να χτίσουμε τη στοίβα του λογισμικού με σκοπό να ταιριάζει στο σενάριο μας, αντί να χτίζουμε το σενάριο μας με σκοπό να ταιριάζει στο λειτουργικό σύστημα.

Για να κάνουμε πιο σαφή το στόχο που έρχονται να καλύψουν οι rump kernels παραθέτουμε ένα απόσπασμα από τη σχετική βιβλιογραφία[7]:

Στη σχετική βιβλιογραφία, υπάρχουν διάφορα σύγχρονα projects που εστιάζουν στην αποφυγή της επιβάρυνσης (overhead) και της έμμεσης εμπλοκής του στρώματος του λειτουργικού συστήματος στο cloud: για παράδειγμα, MirageOS, OSv, και Erlang-on-Xen. Ο δικός μας στόχος ωστόσο με τους rump kernels είναι διαφορετικός. Να προσφέρουμε μία εργαλειοθήκη από drivers για οποιαδήποτε πλατφόρμα αντί για ένα λειτουργικό περιβάλλον για πλατφόρμες cloud [...]. Προσφέρουμε επίσης πλήρη στήριξη των rump kernels σε ένα σύνολο από πλατφόρμες, συμπεριλαμβανομένου του POSIXy user space και του Xen. Αλληλεπιδρούμε επίσης με μία σειρά από άλλες δομές. Για παράδειγμα, υπάρχουν διαθέσιμοι drivers για να συνδυάσει κανείς τη στοίβα TCP/IP που προσφέρουν οι rump kernels με L2 δομές πακέτου του user space όπως το netmap, Snabb Switch και DPDK.

Με δεδομένα τα παραπάνω μπορούμε να συνοψίσουμε το concept των Rump Kernels ως εξής. **Στοχεύουν στην κατά το δυνατόν πληρέστερη υποστήριξη κλασικών εφαρμογών στα πρότυπα του POSIX, παρέχοντας ένα τεράστιο σύνολο από αμετάβλητους drivers διατηρώντας έτσι τα προσδοκώμενα πρότυπα *ποιότητας* που θα είχαμε από ένα παραδοσιακό Λειτουργικό Σύστημα και παράλληλα ελαχιστοποιώντας την προσπάθεια που θα απαιτούσε η κατασκευή τους από την αρχή.**

3.2.3 Από τους Rump Kernel στους Rumprun Unikernels

Πριν προχωρήσουμε στη σχέση μεταξύ του NetBSD, των rump kernels και των rumprun unikernels, θα περιγράψουμε κάποιες βασικές έννοιες που αφορούν διαφορετικές προσεγγίσεις στην αρχιτεκτονική του πυρήνα ενός λειτουργικού συστήματος.

Ένας μονολιθικός πυρήνας (monolithic kernel) είναι μια αρχιτεκτονική λειτουργικού συστήματος όπου ολόκληρο το λειτουργικό σύστημα λειτουργεί στον χώρο του πυρήνα. Το μονολιθικό μοντέλο διαφέρει από άλλες αρχιτεκτονικές δεδομένου ότι ορίζει μόνο του μια εικονική διασύνδεση υψηλού επιπέδου πάνω από το υλικό του υπολογιστή. Ένα σύνολο κλήσεων συστήματος υλοποιεί όλες τις υπηρεσίες του λειτουργικού συστήματος, όπως είναι η διαχείριση των διαδικασιών (process management), ο συγχρονισμός (concurrency) και η

διαχείριση μνήμης. Οι drivers μπορούν να προστεθούν στον πυρήνα ως ενότητες (modules).

Ο όρος anykernel από την άλλη προσπαθεί να διατηρήσει τα πλεονεκτήματα ενός μονολιθικού πυρήνα ενώ ταυτόχρονα επιτρέπει την ταχύτερη ανάπτυξη drivers και την πρόσθετη ασφάλεια στο user space. Η βασική ιδέα αναφέρεται σε μια αρχιτεκτονικά αγνωστική προσέγγιση των drivers, όπου οι τελευταίοι μπορεί είτε να μεταφραστούν μαζί με ένα μονολιθικό πυρήνα ή να εκτελούνται ως διαδικασία του user space χωρίς αλλαγές στον κώδικά τους. Όπου drivers όπως αναφέραμε και νωρίτερα νοούνται όχι μόνο τα προγράμματα οδήγησης συσκευών, αλλά και τα συστήματα αρχείων (file systems) και η στοίβα δικτύου (network stack).

Ο rump kernel, όπως υποδηλώνει και το όνομά του (rump: ένα μικρότερο ή ασήμαντο υπόλειμμα από κάτι αρχικά μεγαλύτερο), είναι ουσιαστικά ένας πυρήνας (ενός παραδοσιακού λειτουργικού με καταμερισμό χρόνου κλπ (timesharing style kernel)) από τον οποίο έχουν αφαιρεθεί κομμάτια. Το υπόλειμμα (rump) είναι οι drivers και οι βασικές ρουτίνες απαραίτητες για να λειτουργήσουν - συγχρονιστούν,οι κατανεμητές μνήμης (memory allocators) και ούτω καθεξής. Αυτό που αφαιρέθηκε είναι πολιτικές για χρονοδρομολόγηση νημάτων (thread scheduling), η εικονική μνήμη, οι πολλαπλές εφαρμογές - διεργασίες και ούτω καθεξής.

Οι Rumprun Unikernels είναι unikernels που χτίζονται ως παράγωγα των drivers και των συστατικών που παρέχουν οι rump kernels. Αποτελούν ουσιαστικά μία από τις πολλές εφαρμογές (use case) που μπορούν να έχουν οι rump kernels. Μία σχέση μεταξύ αυτών των εννοιών μπορεί να γίνει πιο ξεκάθαρη στο Σχ. 3.1

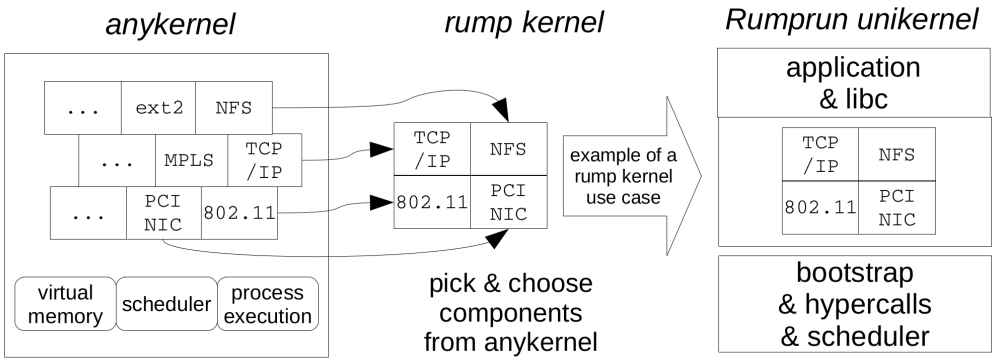


Figure 3.1: Σχέση μεταξύ Anykernel, Rump kernel, Rumprun Unikernel: Ο anykernel επιτρέπει την εξαγωγή drivers και components από το NetBSD source tree και τη δημιουργία rump kernels από τα εξαγώμενα. Οι τελευταίοι χρησιμοποιούνται για να χτιστούν εφαρμογές και πλατφόρμες. Οι Rumprun Unikernel είναι ένα use case των Rump Kernels.

3.2.4 Υποστηριζόμενες υπηρεσίες και πλατφόρμες

Εφαρμογές

Οι Rump kernels και κατ'επέκταση οι Rumprun Unikernels υποστηρίζουν μία ευρεία γκάμα κλασικών εφαρμογών επιτρέποντας την εκτέλεση αυτών σε ένα περιβάλλον απαλλαγμένο από το πρόσθετο overhead των *περιττών* συστατικών του Λειτουργικού Συστήματος. Παράλληλα η πλούσια υποστήριξη τους σε κλασικές βιβλιοθήκες και βασικά συστατικά όπως

η TCP/IP στοίβα, τα συστήματα αρχείων κλπ καθιστά το porting των εφαρμογών αρκετά εύκολη υπόθεση. Πιο συγκεκριμένα τα rumprun-packages[8]παρέχουν πάνω από 30 έτοιμες κλασικές εφαρμογές, βιβλιοθήκες ή δομές (βάσεις κλπ) δεδομένων που μπορούν άμεσα να παραχθούν και να εκτελεστούν ως Rumprun Unikernels, όπως: apache2, redis, nginx, mpg123, mysql, php κλπ.

Πλατφόρμες

Οι rumprun kernels κατηγοριοποιούν τις πλατφόρμες πάνω από τις οποίες μπορούν να εκτελεστούν ως εξής[9]:

- **hw** (hardware): στοχεύει ενσωματωμένα συστήματα και το cloud. Μπορεί να εκτελεστεί baremetal αλλά υποστηρίζει επίσης virtio drivers άρα και QEMU/KVM. Οι υποστηριζόμενες αρχιτεκτονικές επεξεργαστή είναι x86_32 και x86_64. Θεωρητικά υποστηρίζεται επίσης σε αρχιακό στάδιο η αρχιτεκτονική ARM (μόνο 32-bit). Ωστόσο όπως θα δούμε πιο αναλυτικά στην παράγραφο 4.1, από πρακτικής άποψης, δεν καταφέραμε να επιβεβαιώσουμε τον ισχυρισμό αυτό.

- **Xen**: στην εν λόγω πλατφόρμα παρέχονται βελτιστοποιήσεις για εκτέλεση ως paravirtualized guest στον Xen Hypervisor, καθώς και virtualization functions μη διαθέσιμες στην hw platform. Η xen platform υποστηρίζει όχι μόνο τα xl tools αλλά και το Amazon EC2 cloud. Οι υποστηριζόμενοι επεξεργαστές είναι x86_32 και x86_64.

The beef of rump kernels, pun perhaps intended, is allowing third-party projects access to a pool of kernel-quality drivers, and Genode OS has already made use of this possibility.[7]
Όταν γραφόταν αυτό στη σχετική βιβλιογραφία υπήρχαν πολύ λιγότερα παραδείγματα αξιοποίησης του περιβάλλοντος των rump kernels και των rumprun unikernels *πάνω* από άλλα frameworks. Σήμερα υπάρχουν πολλά παραδείγματα όπου το Rump χρησιμοποιείται ως πάροχος drivers και components συνδυαζόμενο με άλλα Unikernel Frameworks που λειτουργούν ως backend επαφή με το *από κάτω* στρώμα (hypervisor, hardware). Μία αξιόλογη και πολλά υποσχόμενη πρόσφατη προσπάθεια από τους Nabla Containers[10] είναι η εκτέλεση των rumprun unikernels αξιοποιώντας ως middleware το Solo5 framework (βλ. παράγραφο 3.3).

3.3 MirageOS

3.3.1 Γενικά

Το MirageOS είναι ένα Library Operating System που χρησιμοποιείται για την κατασκευή Unikernels, που εκτελούνται σε Xen ή KVM hypervisors. Χρησιμοποιεί την προγραμματιστική γλώσσα OCaml, με βιβλιοθήκες που παρέχουν υποστήριξη για δίκτυο, αποθήκευση και συγχρονισμό[11].

3.3.2 Rationale

[3] Παρόλο που η εικονικοποίηση του λειτουργικού συστήματος είναι χρήσιμη, προσθέτει ένα ακόμη στρώμα λογισμικού στα ήδη υπάρχοντα περιλαμβάνοντας υποστήριξη για *παλιά* πρωτόκολλα (π.χ IDE), άσχετες βελτιστοποιήσεις, εφαρμογές χρήστη και νήματα, περιβάλλοντα εκτέλεσης για διάφορες τεχνολογίες (π.χ OCaml, Java κλπ) και πολλά ακόμη φαινομενικά *άχρηστα* για τον κύριο ρόλο που επιτελεί ένα single purpose VM. Όλα αυτά τα στρώματα βρίσκονται κάτω από το στρώμα της κύριας εφαρμογής.

Το παραπάνω πρόβλημα αποτέλεσε σημαντικό πεδίο προβληματισμού στο Εργαστήριο Υπολογιστών του Πανεπιστημίου του Cambridge (όπου και πρωτο-αναπτύχθηκε ο Xen hypervisor το 2003) καθώς και στους κόλπους του Xen Project. Η λύση στους προβληματισμούς αυτούς ήταν η ανάπτυξη του MirageOS.

Ο σκοπός του MirageOS είναι να ανακατασκευάσει τα VMs - τόσο τον πυρήνα όσο και τον κώδικα στο χώρο χρήστη - ώστε να αποτελούνται απο πιο δομοστοιχειωτά συστατικά με έμφαση στην ελαστική και ασφαλή επαναχρησιμοποίησή τους στη βάση ενός Library Operating System.

Σήμερα, είναι πολύ σύνηθες, ένα κλασικό VM σε περιβάλλον cloud που αποτελείται ουσιαστικά από την εικόνα ενός ολόκληρου λειτουργικού συστήματος να λειτουργεί με έναν κυριώς συγκεκριμένο σκοπό. Αυτό σημαίνει ότι αποτελείται από τον πυρήνα του λειτουργικού συστήματος που φιλοξενεί μία κύρια υπηρεσία/εφαρμογή στο χώρο χρήστη (π.χ MySQL, nginx κλπ) μαζί με ορισμένες δευτερεύοντες υπηρεσίες που εκτελούνται παράλληλα (π.χ syslog). Κάθε φορά που το VM εκκινεί, το λογισμικό που το αποτελεί αρχικοποιείται εκ νέου διαβάζοντας τις σχετικες ρυθμίσεις από τις αποθηκευτικές μονάδες. Τα περισσότερα VM λοιπόν χρησιμοποιούνται ως ειδικού σκοπού (single purpose) μηχανές ενώ το λογισμικό τους αποτελείται από πολλά στρώματα παρέχοντας πολύ περισσότερες δυνατότητες σε σχέση με τις ανάγκες του σκοπού, περιορίζοντας έτσι τους ελεύθερες πόρους που θα ήταν δυνατό να ικανοποιήσουν περισσότερες ανάγκες[12]. Παράλληλα οι σύγχρονοι hypervisor προσφέρουν μια αφαιρετική διαχείριση πόρων, επιτρέποντας την εξάπλωσή τους δυναμικά, τόσο με κάθετο τρόπο προσθέτοντας επεξεργαστές και μνήμη όσο και οριζόντια *γεννώντας* περισσότερα VM. Ωστόσο τα υπάρχοντα λειτουργικά συστήματα δεν είναι σε θέση να εκμεταλλευτούν πλήρως αυτή τη δυνατότητα καθώς σχεδιάστηκαν πριν τους hypervisors. Είναι συχνό φαινόμενο η παρακολούθηση και διαχείριση του φόρτου των εφαρμογών ενός VM μέσω μίας εξωτερικής εφαρμογής για αυτόν ακριβώς το σκοπό: όταν αυξάνεται ο φόρτος να *γεννιούνται* νέα VM για να μπορεί η βασική υπηρεσία να αποκρίνεται με ελαστικό τρόπο. Όμως τα παραδοσιακά λειτουργικά συστήματα δεν είναι βέλτιστα ως προς το μέγεθός τους ούτε ως προς το χρόνο εκκίνησής τους με αποτέλεσμα τελικά οι εν λόγω διαχειριστές του φόρτου να κρατάνε *άδεια* αχρησιμοποίητα VM *σε επιφυλακή* ώστε να αντιμετωπίσουν άμεσα τυχόν εξάρσεις στο φόρτο εργασίας, σπαταλώντας έτσι πόρους.

3.3.3 Applicability

Το MirageOS περιλαμβάνει μια σειρά από καθαρές λειτουργικές υλοποιήσεις διάφορων χρήσιμων πρωτοκόλλων όπως TCP/IP, DNS, SSH, Openflow (switch/controller), HTTP, XMPP και Xen inter-VM transports. Τα παραπάνω το καθιστούν ιδιαίτερα χρήσιμο για ανάπτυξη δικτυακών εφαρμογών και υπηρεσιών web. Υπάρχουν πολλά παραδείγματα self-hosted διαδικτυακών ιστοσελίδων που εκτελούνται ως appliances του MirageOS απευθείας πάνω σε Xen hypervisor σε γνωστά δημόσια clouds (π.χ Amazon EC2) ή υπηρεσιών (π.χ DNS) που τρέχουν ως unikernels χρησιμοποιώντας τις OCaml βιβλιοθήκες που παρέχει το MiragOS. Παράλληλα το MirageOS με τη βοήθεια του Solo5 framework (βλ. παράγραφο 3.3) είναι πλέον εφικτό να εκτελεστεί και πάνω από το KVM σε αρχιτεκτονικές ARM προσφέροντας την ευκαιρία για αποδοτική εξοικονόμηση πόρων και ασφάλεια σε χαμηλής κατανάλωσης ενσωματωμένα συστήματα.

Στο Σχ. 3.2 φαίνεται ένα παράδειγμα ανάπτυξης μίας http εφαρμογής ως MirageOS unikernel.

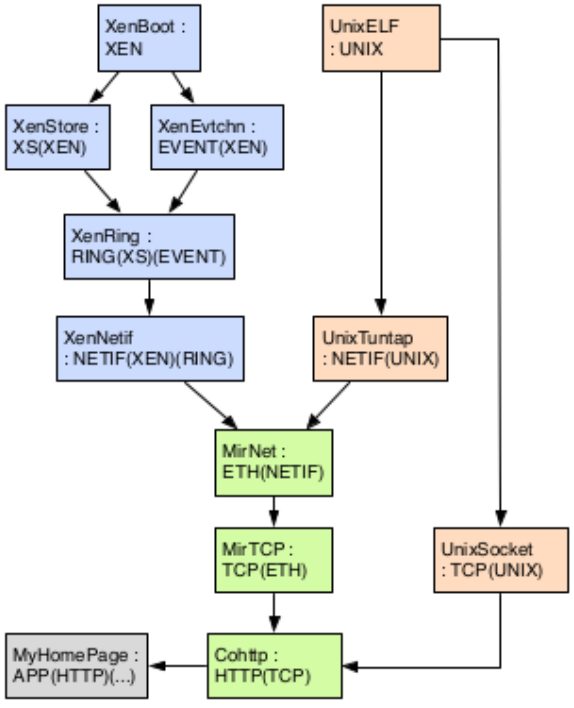


Figure 3.2: Παράδειγμα Ανάπτυξης MirageOS εφαρμογής[13]

Η εφαρμογή MyHomePage εξαρτάται από μία HTTP υπογραφή που της παρέχεται μέσω της βιβλιοθήκης Cohhttp. Η βιβλιοθήκη αυτή χρειάζεται μια υλοποίηση του TCP πρωτοκόλλου. Κατά τη διάρκεια της ανάπτυξης της εφαρμογής (θεωρούμε σε περιβάλλον Unix) από τον προγραμματιστή οι ανάγκες της Cohhtp καλύπτονται από κάποια UnixSocket Library που παρέχει το περιβάλλον ανάπτυξης. Στο τελευταίο στάδιο και αφού η εφαρμογή έχει πλέον αναπτυχθεί, το Unix κομμάτι "αποβάλλεται" από το σχεδιασμό, και η εφαρμογή γίνεται ξανά compile χρησιμοποιώντας το MirNet στοιχείο του MirageOS, που αποτελεί ουσιαστικά τη διεπαφή με τον οδηγητή δικτύου του hypervisor (Xen εν προκειμένω) καταλήγοντας τελικά, ανεβαίνοντας το *δέντρο* εξαρτήσεων προς τα πάνω, να συμπεριλάβει (μόνο) όλα τα απαραίτητα στοιχεία που χρειάζονται, μέχρι τελικά τον απαραίτητο κώδικα (XenBoot) για την εκκίνηση του Unikernel Image.

3.3.4 Υποστηριζόμενες πλατφόρμες

- Xen Hypervisor
- KVM σε aarch64 πλατφόρμες χρησιμοποιώντας το Solo5 framework

3.4 Solo5

3.4.1 Γενικά

Το Solo5 αρχικά ξεκίνησε ως έργο του Dan Williams από την IBM Research, στοχεύοντας την εκτέλεση του MirageOS στο KVM. Από τότε, έχει εξελιχθεί σε ένα γενικότερο περιβάλλον εκτέλεσης που προσφέρει απομόνωση (sandbox), κατάλληλο για την εκτέλεση εφαρμογών που έχουν δημιουργηθεί με τη χρήση διαφορετικών unikernel frameworks (γενικότερα library OS), με στόχο διάφορες τεχνολογίες sandboxing σε διαφορετικά host λειτουργικά συστήματα και hypervisors[14].

Ουσιαστικά το Solo5 είναι ένα middleware interface μεταξύ μιας εφαρμογής-unikernel και του host συστήματος, και δεν αποτελεί από μόνο του ένα τελικό προϊόν (Unikernel).

3.4.2 Rationale

Αν παρομοιάσουμε τους unikernel με μία διεργασία χρήστη τότε μπορούμε να σκεφτούμε το Solo5 ως έναν επανασχεδιασμό του interface μεταξύ της διεργασίας και του host λειτουργικού συστήματος ή του hypervisor με στόχο αυτό να είναι:

- όσο το δυνατόν πιο *legacy-free* και *λεπτό* σε σχέση με τις υπάρχοντες υλοποιήσεις. Να *εκθέτει* δηλαδή *λιγότερα* σε μια διεργασία σε σχέση με αυτά που εκθέτει ο πυρήνας σε μία διεργασία Linux ή σε μία εικονική μηχανή μέσω του QEMU. Στοχεύει έτσι στην αύξηση της ασφάλειας αφού μειώνεται η πιθανή επιφάνεια - στόχος και στην απομόνωση των Solo5 εφαρμογών.
- όσο το δυνατόν πιο εύκολο να υλοποιηθούν νέοι στόχοι όπως νέες τεχνολογίες sandboxing (π.χ hardware virtualization), host λειτουργικά συστήματα και hypervisors. Στοχεύει έτσι στο portability των Solo5-εφαρμογών σε διαφορετικές αρχιτεκτονικές χωρίς να χρειάζονται αλλαγές στον κώδικά τους.

Ακόμη το Solo5 εισάγει την έννοια ενός προγράμματος φύλακα (tender), του hvt (πρώην uKVM), που αποτελεί ουσιαστικά ένα εξειδικευμένο πρόγραμμα monitor για unikernels. Σε αναλογία μπορούμε να παρομοιάσουμε το ρόλο του με το ρόλο του QEMU σε ένα σενάριο QEMU/KVM. Το hvt ωστόσο αποτελεί monitor εξειδικευμένου σκοπού σε αντίθεση με το QEMU που είναι γενικού σκοπού. Αυτό σημαίνει ότι το interface που παρέχει μεταξύ του Solo5 kernel και του Linux/KVM είναι στοχευμένο ως προς τις ανάγκες της Unikernel εφαρμογής. Στη μέχρι τώρα μελέτη μας είδαμε οτι οι Unikernels αφαιρούν ουσιαστικά τα

μη χρησιμοποιούμενα από την εφαρμογή components του λειτουργικού. Το Solo5 με το hvt προσθέτει ένα ακόμα επίπεδο αφαίρεσης. Το εξειδικευμένο monitor εκθέτει, με το modular interface του, προς τον Unikernel μόνο τα απαραίτητα. Στο Σχ. 3.3 παραθέτουμε μια σύγκριση μεταξύ ενός παραδοσιακού VMM και του hvt για έναν Unikernel που χρησιμοποιεί τη συσκευή δικτύου, και παρατηρούμε ότι στην πρώτη περίπτωση περιλαμβάνονται άσκοπα interfaces (πχ block device backend).

Τα παράγωγα του Solo5 είναι η εικόνα της εφαρμογής (Solo5 kernel + Unikernel) και το εκτελέσιμο του hvt προσαρμοσμένο για την εικόνα αυτή.

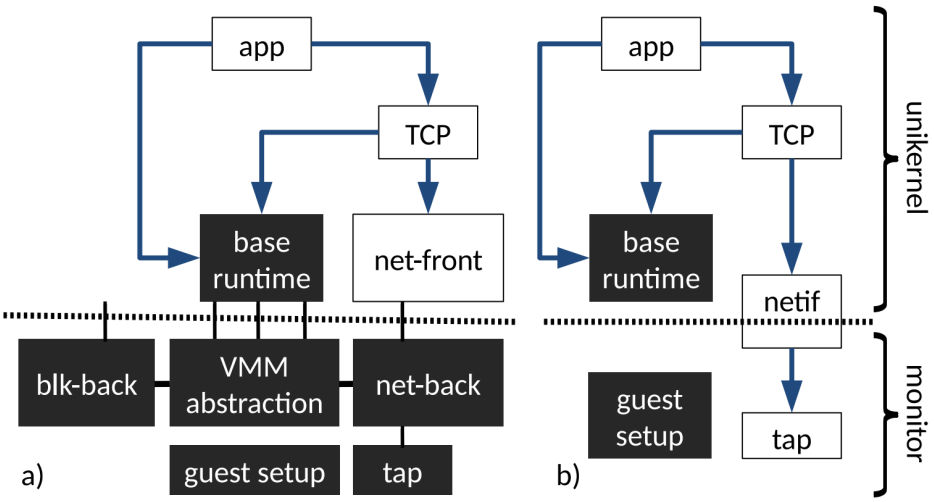


Figure 3.3: Monitor γενικού σκοπού και monitor ειδικού σκοπού [15]

Εξειδικεύοντας το monitor, το Solo5 υλοποιεί μικρότερα interfaces (με την έννοια ότι εκθέτει λιγότερα virtual devices) από το Linux/KVM προς τον unikernel και επομένως πετυχαίνει υψηλότερη ασφάλεια και γρηγορότερα boot times, γλυτώνοντας χρόνο από το initialization *άχρηστων* συσκευών. Επίσης η σχεδίαση των interfaces γίνεται πιο απλή αφού όντας εξειδικευμένο προς την εφαρμογή δε χρειάζεται να ακολουθεί τα legacy standards ενός monitor γενικού σκοπού όπως το QEMU. Τέλος, όπως είναι αναμενόμενο τα μέγεθος του monitor είναι αισθητά μικρότερο από το QEMU μειώνοντας έτσι το λόγο (μέγεθος unikernel)/(μέγεθος monitor) σε ποσοστά της τάξης του 0.02%[15].

Συνοψίζοντας, ο στόχος του Solo5 εύστοχα περιγράφεται από τη σχετική βιβλιογραφία ως εξής[16]:
Η υλοποίηση του Solo5 προσπαθεί να είναι μινιμαλιστική και σωστή, δίνοντας έμφαση στην ανάπτυξη καλών εξωτερικών, εσωτερικών και user interfaces που θα διευκολύνουν την εισαγωγή νέων χαρακτηριστικών και θα διευρύνουν το πεδίο εφαρμογής του.

3.4.3 Applicability

Υπάρχουν ήδη μια σειρά από Unikernel Frameworks τα οποία υποστηρίζουν ως backend πλατφόρμα το Solo5 αξιοποιώντας το στόχο που το τελευταίο επιδιώκει να καλύψει. Επίσημα το Solo5 μπορεί να συνδυαστεί με το MirageOS (άλλωστε αυτός ήταν και ο αρχικός του στόχος) καθώς και με το IncludeOS (βλ. παράγραφο 3.4). Παράλληλα όπως αναφέραμε στην παράγραφο 3.1.4 σε πιο πειραματικό στάδιο βρίσκεται και μία προσπάθεια υποστήριξης των Rumprun Unikernels.

Το Solo5 λειτουργεί ουσιαστικά ως η βάση του Unikernel και με αυτό τον τρόπο καθορίζει σε ποιες πλατφόρμες μπορεί να τρέξει ο unikernel (**πού**) , πόσο γρήγορα θα γίνει το **boot**, και τι **δυνατότητες** έχουν τα υψηλότερα στρώματα στη στοίβα. Σχηματικά τα παραπάνω συνοψίζονται στο Σχ. 3.4

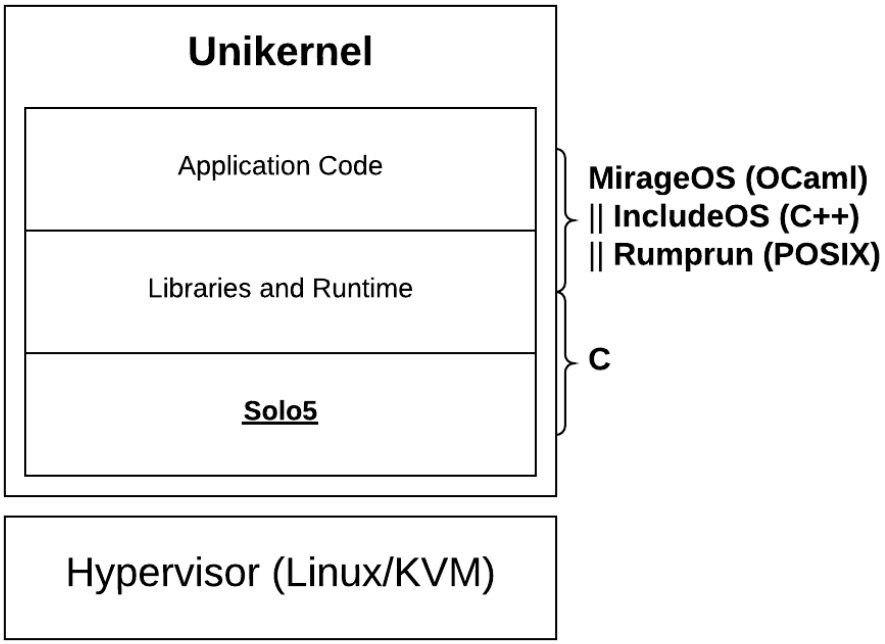


Figure 3.4: Το Solo5 ως βάση του Unikernel

3.4.4 Υποστηριζόμενες πλατφόρμες

Το Solo5 υποστηρίζει τους παρακάτω συνδυασμούς tender/Virtual Machine Monitors (VMM), λειτουργικών συστημάτων και αρχιτεκτονικών:

- hvt ως tender με Linux/KVM ή FreeBSD VMM σε αρχιτεκτονική x86_64 (επίσημη παραγωγή)
- hvt ως tender με OpenBSD σε αρχιτεκτονική x86_64 (πειραματικά)
- hvt ως tender με Linux/KVM σε αρχιτεκτονική aarch64 (ARM 64-bit) (πειραματικά)

- Οποιοδήποτε hypervisor με virtio συσκευές σε αρχιτεκτονική x86_64 όπως QEMU / KVM ή Google Compute Engine (περιορισμένη υποστήριξη - επίσημα deprecated)

- Muen Separation Kernel[17] σε αρχιτεκτονική x86_64 (πειραματικά)

3.5 IncludeOS

3.5.1 Γενικά

Το IncludeOS είναι ένα μινιμαλιστικό unikernel λειτουργικό σύστημα που μπορεί να συμπεριληφθεί σε C++ υπηρεσίες που εκτελούνται στο cloud. Αν ξεκινήσουμε ένα πρόγραμμα με *include ¡os¿* τότε κυριολεκτικά θα συμπεριληφθεί κατά το link-time ένα μικροσκοπικό λειτουργικό σύστημα μέσα στην υπηρεσία που χτίζουμε[18]. Το IncludeOS βρίσκεται αυτή τη στιγμή (Σεπτέμβριος 2018) υπό ανάπτυξη στην έκδοση v0.12.0.

3.5.2 Rationale

Το cloud computing ως η επικρατέστερη πλατφόρμα για ελαστικές και κλιμακωτές (ως προς απαιτήσεις) υπηρεσίες έχει γεννήσει την ανάγκη και την ευκαιρία για νέους τύπους λειτουργικών συστημάτων. Το IncludeOS με βάση την παραπάνω παραδοχή εστιάζει:

- Σε υψηλού επιπέδου εξειδικευμένα components.

- Σε ελάχιστη σπατάλη πόρων κατά την εκτέλεση.

Σε αντίθεση με άλλα unikernel frameworks - library operating systems το IncludeOS δε βασίζεται σε υπάρχοντες βιβλιοθήκες και drivers αλλά αντιθέτως έχει αναπτυχθεί αυτόνομα και τα περισσότερα συστατικά του έχουν σχεδιαστεί και "γραφτεί" από την αρχή σε C++.

Τα κύρια χαρακτηριστικά που επιδιώκει να προσφέρει είναι:

- Αποδοτική διαχείριση πόρων και ίχνους στην μνήμη

- Αποδοτική διαδικασία deployment: Χρησιμοποιείται μία custom clang-based εργαλειοθήκη.

- Ανεξαρτησία από την πλατφόρμα εικονικοποίησης: Το IncludeOS μπορεί να τρέξει σε οποιαδήποτε πλατφόρμα εικονικοποίησης αρχιτεκτονικής x86 (openStack, KVM, VirtualBox κλπ).

3.5.3 Applicability

Οι παραπάνω στόχοι υλοποιούνται στο επίπεδο σχεδιασμού και αρχιτεκτονικής του IncludeOS με βασικά σημεία τα παρακάτω[19]:

Zero overhead principle

Μόνο τα χαρακτηριστικά που απαιτεί από το λειτουργικό μία συγκεκριμένη υπηρεσία (που τρέχει ως IncludeOS εφαρμογή) συμπεριλαμβάνονται στην τελική εικόνα, μειώνοντας έτσι τη σπατάλη πόρων και οδηγώντας σε καλύτερη απόδοση του δικτύου και της μνήμης. Παράλληλα σε κατάσταση idle δεν εκτελούνται περαιτέρω υπο-διεργασίες ή διακοπές λόγω του χρονοδρομολογητή επομένως δε χρησιμοποιείται καθόλου η CPU. Τα παραπάνω έρχονται σε συμφωνία με το zero overhead principle της C++: *για ότι δε χρησιμοποιείς, δεν πληρώνεις*[20].

Στατικές βιβλιοθήκες και εργαλειοθήκη clang

Κατά τη διαδικασία ανάπτυξης της εφαρμογής η δήλωση *include ¡os¿* περιλαμβάνει το *λειτουργικό σύστημα*. Κατά το link time το build system αφαιρεί από την (precompiled) βιβλιοθήκη, που έγινε include, ότι χρειάζεται η εφαρμογή και δημιουργεί ένα εκτελέσιμο. Η επιλογή και αφαίρεση από το σύνολο των βιβλιοθηκών μόνο των απαραίτητων στοιχείων που απαιτεί η εφαρμογή γίνεται με τις μεθόδους που προσφέρουν οι σύγχρονοι linkers. Κάθε αυτόνομη μονάδα-στοιχείο του λειτουργικού μετατρέπεται σε ένα object file (π.χ ip4.o, udp.o κλπ), και στη συνέχεια αυτά μέσω του ar προγράμματος *ενώνονται* σε μία στατική βιβλιοθήκη os.a. Με αυτό τον τρόπο όταν ένα πρόγραμμα γίνει link με το os.a, ο linker θα επιλέξει και θα συμπεριλάβει στο τελικό εκτελέσιμο μόνο τα απαραίτητα αντικείμενα.

Κλασικές βιβλιοθήκες

Ως κλασική βιβλιοθήκη της C το IncludeOS επέλεξε την newlib της RedHat. Πρόκειται για μία μικρού μεγέθους βιβλιοθήκη που υλοποιεί μικρότερο αριθμό system calls και η οποία είναι σχεδιασμένη με τέτοιο τρόπο ώστε να μπορεί να κατασκευαστεί στατικά; με αυτόν τον τρόπο ικανοποιείται και το κριτήριο που περιγράψαμε ακριβώς απο πάνω.

Ως κλασική βιβλιοθήκη της C++ για τον πυρήνα του IncludeOS επιλέχθηκε αρχικά η EASTL (Electronic Arts) η οποία δεν περιλαμβάνει exceptions, καθιστώντας την "ελαφρύτερη" και λιγότερο πολύπλοκη από την C++ standard library που σε μεγάλο βαθμό βασίζεται στην STL (Standard Template Library). Η EASTL περιλαμβάνει τα πιο σημαντικά στοιχεία της STL όπως strings, streams, vectors, maps κλπ, ωστόσο η έλλειψη ορισμένων στοιχείων συμπληρώνεται με custom υλοποιήσεις του που παρέχει το IncludeOS. Πλέον στη σημερινή του έκδοση (v0.12.0) έχει αντικατασταθεί από την libc++ του LLVM[21] χωρίς ωστόσο την υποστήριξη νημάτων (threads) και filestreams[22].

Virtio network driver

Ο hypervisor δε χρειάζεται να κάνει emulate μία φυσική συσκευή δικτύου αλλά αντιθέτως μπορεί να μοιράζεται δεδομένα απευθείας με τον guest μέσω μίας ουράς σε κοινής χρήσης μνήμη.

Modular αντικειμενοστραφής στοίβα δικτύου

Για να ανταποκριθεί στο no overhead principle, το IncludeOS υλοποιεί μια αντικειμενοστραφή στοίβα δικτύου που επιτρέπει την δόμησή της με τρόπο που να κατά την κατασκευή της να περιλαμβάνονται μόνο τα στοιχεία που θα χρησιμοποιήσει η εφαρμογή. Για παράδειγμα στην περίπτωση του UDP δε χρειάζεται να πληρώσουμε το κόστους του routing ή να συμπεριληφθεί και η υλοποίηση του TCP.

Ασύγχρονο I/O και αναβαλλόμενες διακοπές (IRQ)

Όλοι οι χειριστές διακοπών (IRQ Handlers) στο IncludeOS απλά αυξάνουν έναν μετρητή και παραδίδουν το χειρισμό στο βρόχο κύριων συμβάντων, όποτε υπάρχει χρόνος. Με αυτόν τον τρόπο δεν υπάρχει ανάγκη για context switch και επίσης μειώνονται τα προβλήματα συγχρονισμού όπως τα race conditions. Με το ασύγχρονο I/O ο επεξεργαστής μένει πάντα απασχολημένος χωρίς να εμποδίζεται η λειτουργία του μέχρι να ολοκληρωθούν τα I/O requests.

Τα παραπάνω χαρακτηριστικά φαίνεται πώς έχουν ευεργετικό ρόλο στο memory footprint και στο χρόνο εκκίνησης, παράγοντες αρκετά σημαντικοί στο cloud computing στο οποίο στοχεύει το IncludeOS. Τα παραδείγματα από τη σχετική βιβλιογραφία[19] που συγκρίνουν με άλλες ευρέως χρησιμοποιούμενες πλατφόρμες στον τομέα του cloud, είναι χαρακτηριστικά: Ένα IncludeOS πρόγραμμα *Hello World* μαζί με το QEMU έχει πολύ μικρότερο αποτύπωμα στη μνήμη, σε σχέση με ένα αντίστοιχο πρόγραμμα Java (8.45 MB vs 28.29 MB) ενώ η διαφορά σε σχέση με το ελάχιστο αποτύπωμα μνήμης που θα απαιτούσε ένα Ubuntu VM (επιλέχθηκε ως reference λειτουργικό στο OpenStack) είναι ακόμη μεγαλύτερη (300 MB). Τέλος ο χρόνος εκκίνησης για μία (minimal) IncludeOS εικονική μηχανή κυμαίνεται στα 300ms.

3.5.4 Υποστηριζόμενες πλατφόρμες

Το IncludeOS στην τρέχουσα έκδοσή του (v0.12.0) υποστηρίζει την αρχιτεκτονική x86 ενώ μπορεί να τρέξει πάνω από τους περισσότερους γνωστούς hypervisor καθώς και στο openStack. Οι παραγωγοί του υποστηρίζουν ότι μπορεί να τρέξει και bare-metal ενώ στα πλάνα τους για τις επόμενες εκδόσεις περιλαμβάνεται και η υποστήριξη αρχιτεκτονικής ARM. Τέλος όπως αναφέραμε και παραπάνω το IncludeOS μπορεί να χρησιμοποιήσει ως βάση το Solo5-hvt *πάνω* από το KVM.

3.6 OSv

3.6.1 Γενικά

Το OSv δημιουργήθηκε από την Cloudius Systems ως ένα λειτουργικό σύστημα που στοχεύει το cloud. Ο αρχικός σκοπός του ήταν να προσφέρει μία πλατφόρμα εκτέλεσης Java εικονικών μηχανών, απλώς με την τοποθέτηση ενός αρχείου WAR (Web Application

Resource ή Web application ARchive)[2] σε αυτές. Ωστόσο η αρχιτεκτονική του υποστηρίζει σχεδόν κάθε single-process υπηρεσία και πολλές διαφορετικές γλώσσες προγραμματισμού (C, C++, Ruby, Perl κλπ).

3.6.2 Rationale

Το OSv διαφέρει από τα σχετικά frameworks καθώς εξαρχής ο βασικός του στόχος ήταν η μετατροπή σχεδόν κάθε εφαρμογής σε έναν λειτουργικό Unikernel. Είναι δηλαδή μία λύση γενικότερου σκοπού η οποία στοχεύει σε εφαρμογές που μπορούν να τρέξουν ως single processes (πολλαπλά νήματα υποστηρίζονται, αλλά όχι πολλαπλές διεργασίες). Αυτό φυσικά σημαίνει ότι οι παραγώμενοι Unikernels είναι πιο μεγάλοι σε μέγεθος (τάξη μεγέθους σε MBs) σε αντίθεση με άλλες σχετικές τεχνολογίες που μετράνε το μέγεθος τους σε τάξεις κάποιων KBs. Σε αυτό συνηγορεί επίσης το γεγονός ότι στοχεύεται μια ευρεία γκάμα hypervisors (π.χ KVM, Xen, Virtualbox, VMware).

3.6.3 Applicability

Σύμφωνα με τη σχετική βιβλιογραφία[24] το OSv καλύπτει ένα μεγάλο εύρος εφαρμογών και διαφορετικών τεχνολογιών.

Εικονικοποιημένες εφαρμογές

Οι ανεξάρτητοι προμηθευτές λογισμικού (ISVs - Independent software vendors) που προσφέρουν εφαρμογές σε εικονικές μηχανές μπορούν να εκμεταλλευτούν το OSv για τις εφαρμογές τους. Το μέγεθος των OSv-based VMs είναι συνήθως μόνο 12-20 MB μεγαλύτερο από την εφαρμογή αυτή καθε αυτή. Εκτός από το μικρότερο μέγεθος τα οφέλη αφορούν και τη συντήρηση ενός σημαντικά μικρότερου συνόλου λογισμικού και ρυθμίσεων σε σχέση ακόμα και με τις πιο μινιμαλίστικες υλοποιήσεις παραδοσιακών guests από άλλες πλατφόρμες (qemu linux guest κλπ).

Εικονικοποίηση δικτυακών λειτουργιών

Η εικονικοποίηση των συσκευών δικτύου απαιτεί χαμηλή καθυστέρηση και υψηλή διέλευση. Το OSv ακολουθεί μία τακτική που ονομάζεται "κανάλια δικτύου" (Network Channels). Ένα μεγάλο μέρος του φόρτου που απαιτεί η στοίβα του TCP/IP μεταφέρεται σε μία χωριστή εφαρμογή-νήμα (application thread) και ένας μικρός ταξινομητής πακέτων (packet classifier) εκτελείται σε ένα νήμα διαχείρισης των διακοπών δικτύου. Με αυτόν τον τρόπο τόσο η επεξεργασία γίνεται στο ίδιο context στον ίδιο επεξεργαστή, κρατώντας τα δεδομένα στην cache αποφεύγοντας το πρόσθετο κόστος καθυστέρησης που θα προκαλούσε το context switch από user σε kernel space.

[2] ένα αρχείο που χρησιμοποιείται για τη διανομή μιας συλλογής απο αρχεία JAR, JavaServer Pages, Java Servlets, Java κλάσεις, αρχεία XML, tag βιβλιοθήκες, στατικές ιστοσελίδες (HTML) και γενικά resources που όλα μαζί συνιστούν μια διαδικτυακή εφαρμογή [23]

Εξυπηρετητής εφαρμογών Java

Οι χρήστες μπορούν να "ανεβάσουν" ένα αρχείο WAR, μέσω ενός REST API[3], και η εφαρμογή εκτελείται χωρίς να χρειάζεται κάποια επιπρόσθετη ρύθμιση.

Εφαρμογές C/C++

Η μεταφορά C/C++ εφαρμογών στην πλατφόρμα του OSv απαιτεί συνήθως ελάχιστες αλλαγές στα αρχεία Makefile καθώς δίνεται η δυνατότητα στο χρήστη να χρησιμοποιήσει αμετάβλητες κοινόχρηστες βιβλιοθήκες του Linux.

Οριζόντια κλιμάκωση

Ο αρκετά χαμηλός χρόνος εκκίνησης καθιστά το OSv κατάλληλο για εφαρμογές NoSQL και άλλες που απαιτούν οριζόντια κλιμάκωση και μικρή περίοδο αποτυχίας (failover). Είναι συχνά πιο γρήγορο να εκκινηθεί ένας καινούριος OSv guest παρά να ξεπεραστεί η αποτυχία ενός που εκτελείται ήδη.

Τέλος αξίζει να σημειωθεί ότι το OSv χρησιμοποιείται στο έργο MIKELANGELO[25] που στοχεύει να καλύψει το κενό μεταξύ του HPC (υπολογισμός υψηλών επιδόσεων) και των τεχνολογιών Νέφους, προσφέροντας την ευκαμψία του cloud στο HPC, και την αποδοτικότητα του HPC στο cloud.

3.6.4 Υποστηριζόμενες πλατφόρμες

Επισήμως υποστηρίζεται η αρχιτεκτονική x86_64 και τα KVM, Xen ως hypervisor καθώς και το Amazon EC2 ως εξωτερικός πάροχος υπηρεσιών νέφους. Πειραματικά στην τρέχουσα έκδοση (Σεπτέμβριος 2018) υποστηρίζονται τα VMWare και VirtualBox ως hypervisors και το Google Compute Engine (GCE) ως εξωτερικός πάροχος υπηρεσιών νέφους. Τέλος σε καθαρά πειραματικό στάδιο βρίσκεται και η προσπάθεια υποστήριξης της αρχιτεκτονικής aarch64 σε QEMU/KVM για την πλατφόρμα Mach-virt[26].

[3] μία προγραμματιστική διεπαφή (API) που χρησιμοποιεί αιτήματα HTTP (GET, PUT, POST, DELETE) για την επικοινωνία μεταξύ διαφορετικών διαδικτυακών εφαρμογών ή/και μεταξύ μίας εφαρμογής και του λειτουργικού συστήματος.

Κεφάλαιο 4

Υποστήριξη αρχιτεκτονικής ARM

4.1 Rumprun

Στα πλαίσια της εργασίας αυτής επιχειρήσαμε να αξιολογήσουμε τους Rumprun Unikernels σε πλατφόρμες αρχιτεκτονικής ARM. Ωστόσο τελικά στη μελέτη μας αυτή εντοπίσαμε μια σειρά από προβλήματα τόσο στο build system του συγκεκριμένου framework όσο και στην ουσιαστική έλλειψη του απαραίτητου κώδικα χαμηλού επιπέδου για την εικονικοποίηση της συγκεκριμένης αρχιτεκτονικης. Πριν προχωρήσουμε σε μια πιο αναλυτική εξήγηση αυτών, θα επιχειρήσουμε σύντομα να περιγράψουμε ορισμένα τεχνικά χαρακτηριστικά του Rumprun απαραίτητα για να κατανοήσει ο αναγνώστης τα παραπάνω συμπεράσματα.

4.1.1 Στοίβα Rumprun

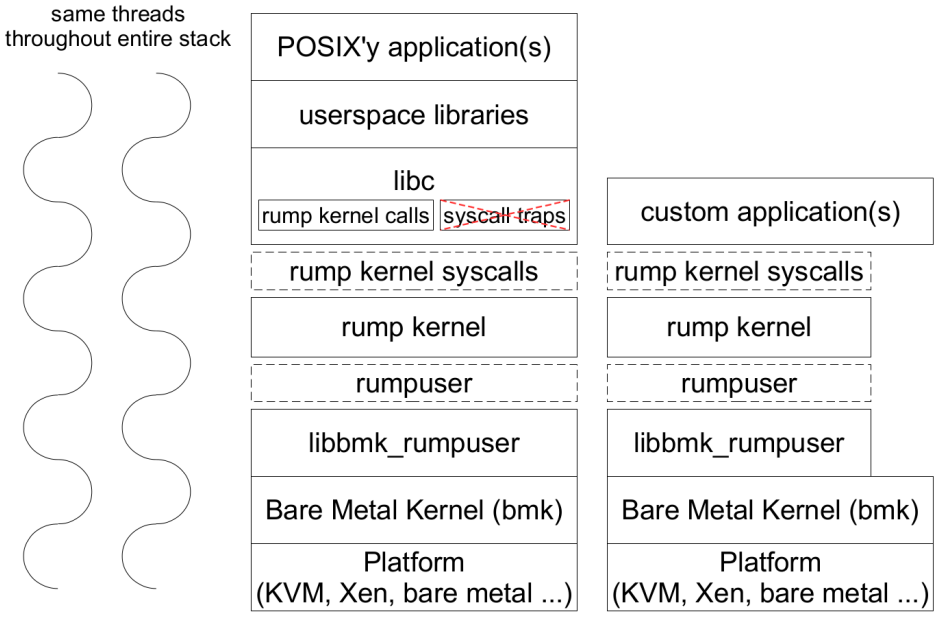


Figure 4.1: Στοίβα Rumprun [6]

Όπως αναφέραμε και στο προηγούμενο κεφάλαιο ο rump kernel δεν είναι ένας *κανονικός* kernel με την έννοια ότι στο σενάριο του Rumprun δεν έχει τη δυνατότητα της άμεσης επικοινωνίας με το χαμηλότερο επίπεδο (τον hypervisor) αλλά ούτε παρέχει τις απαραίτητες μεθόδους για την εκκίνηση (bootstrap), τη χρονοδρομολόγηση, τις διακοπές και τη διαχείριση των σελίδων μνήμης. Στη στοίβα λοιπόν του rumprun unikernel (Σχ. 4.1) υπάρχει ένα ακόμη layer που ονομάζεται Bare Metal Kernel (bmk), και αποτελεί ουσιαστικά τον πυρήνα που εκτελεί τις απούσες παραπάνω απαραίτητες λειτουργίες ώστε να μπορεί να εκτελεστεί ο rump kernel σε κάποια από τις πλατφόρμες που υποστηρίζει. Τα κομμάτια της υλοποίησης του bmk που είναι κοινά για όλες τις πλατφόρμες βρίσκονται στη βιβλιοθήκη libbmk_core. Τα υπόλοιπα που αφορούν την κάθε πλατφόρμα και αρχιτεκτονική βρίσκονται κάτω από το φάκελο platform και αφορούν είτε τον Xen hypervisor (platform/xen) είτε το KVM ή bare metal (platform/hw). Ο συγκεκριμένης αρχιτεκτονικής κώδικας είναι απαραίτητος για όλες τις διαδικασίες που εμπλέκουν χαμηλού επιπέδου (κοντά στο πραγματικό υλικό) λειτουργίες όπως το bootstrap, οι διακοπές, η αρχικοποίηση διαφόρων καταχωρητών κλπ. Όσον αφορά την αρχιτεκτονική ARM που μας απασχόλησε στα πλαίσια αυτής της εργασίας, το rumprun παρέχει τον κώδικα αυτό στοχεύοντας συγκεκριμένα baremetal ή KVM hypervisor για την πλατφόρμα ARM Integrator, αρχιτεκτονικής ARM 32-bit. Παράλληλα η στοίβα περιλαμβάνει ένα hypercall interface (libbmk_rumpuser), απαραίτητο για την επικοινωνία του rump kernel με τον bare metal kernel.

4.1.2 Bootstrap

Τα πρώτα στάδια της εκκίνησης του rumprun στην περίπτωση του KVM παρέχονται από το QEMU. Το σημείο στο οποίο εισέρχεται το bmk βρίσκεται στο platform/hw/arch/arm/integrator/locore.S όπου, αφού γίνονται οι απαραίτητες ενέργειες, στη συνέχεια μέσω της arm_boot() (matchdep.c) εισέρχεται ο προγραμματισμός (σε C) των διάφορων απαραίτητων αρχικοποιήσεων για τη συγκεκριμένη αρχιτεκτονική (π.χ ελεγκτής διακοπών κλπ). Τελικά μετά την αρχικοποίηση όλων των στοιχείων του bmk, καλείται η bmk_sched_startmain() και δημιουργείται το κύριο νήμα που εκτελεί την εφαρμογή του χρήστη.

4.1.3 Build Process for ARM

Το build process του Rumprun βασίζεται σε cross-compilation. Για να ολοκληρωθεί επιτυχώς, για ARM αρχιτεκτονική, χρειάζεται να κάνουμε αρκετές παρεμβάσεις στο default build system του. Αυτές περιλαμβάνουν αλλαγές στα Makefile και τα script που το απαρτίζουν για την ορθή αναγνώριση των χαρκτηριστικών του cross compiler που χρησιμοποιούμε (πιο συγκεκριμένα το floating point unit) καθώς και την ανάπτυξη μίας βιβλιοθήκης υποστήριξης του Integrator board στο πρώιμο στάδιο παραγωγής του Rump Kernel, με τη χρήση των σχετικών components από το source tree του NetBSD.

4.1.4 Αξιολόγηση

Παρότι στη σχετική βιβλιογραφία υπάρχουν παραδείγματα εκκίνησης των rumprun unikernel σε virtualized arm περιβάλλον (σε QEMU emulation mode και χωρίς KVM), δε καθέστει δυνατό να το επαληθεύσουμε (δεν υπήρξε output στην κονσόλα του QEMU σε καμία από τις απόπειρές μας). Τα σχετικά παραδείγματα αφορούσαν ωστόσο παλαιότερες εκδόσεις του rumprun και των rump kernels, σε σχέση με την τρέχουσα έκδοσή του, επομένως συνυπολογίζοντας την *ανενεργή* κατάσταση του επίσημου repository μπορούμε να υποθέσουμε ότι οι τελευταίες αλλαγές που υπήρξαν δεν έχουν ακόμα διορθωθεί πλήρως. Η έλλειψη ακόμη και του σχετικού μηνύματος που τυπώνει η arm_boot() στο αρχικό στάδιο της εκκίνησης του bare metal core μας οδηγεί στην υποψία ότι το πρόβλημα βρίσκεται στον architecture specific κώδικα για τη συγκεκριμένη πλατφόρμα.

Επιπρόσθετα η έλλειψη documentation και το γεγονός ότι η *επίσημη* έκδοση του rumprun παρόλο που θεωρητικά υποστηρίζει το ARM integrator board, χρειάζεται μια σειρά από εξωτερικές παρεμβάσεις (αναφερόμαστε στα προβλήματα του build συστήματος που εντοπίσαμε και αντιμετωπίσαμε στοχεύοντας την ARM αρχιτεκτονική) ώστε να παράξει το τελικό unikernel image, είναι ενδεικτικά της προβληματικής υποστήριξης για την αρχιτεκτονική ARM.

Τελικά τόσο η έλλειψη του σχετικού hardware που θα επέτρεπε την αξιολόγηση του rumprun σε KVM, όσο και το γεγονός ότι ήδη η συγκεκριμένη αρχιτεκτονική έχει αντικατασταθεί από την 64-bit εκδοχή της (aarch64) καθώς και η έλλειψη υποστήριξης νεότερων εκδόσεων / πλατφορμών μας αποθάρρυναν από την περαιτέρω μελέτη του συγκεκριμένου προβλήματος. **Άποψή μας είναι ότι οι μελλοντικές προσπάθειες που θα αφορούν το συγκεκριμένο framework πρέπει να εστιάσουν στην αναβάθμιση του κώδικα πηγής (source code) των rump kernel στις τελευταίες εκδόσεις του NetBSD και κυρίως στην υλοποίηση των χαμηλών επιπέδου ρουτινών που θα προσθέσουν υποστήριξη για την τρέχουσα επικρατούσα ARM αρχιτεκτονική (aarch64).**

4.2 Solo5

Παρόλο που ο αρχικός στόχος του Solo5 ήταν η υποστήριξη virtio για το MirageOS, ώστε αυτό να μπορεί να εκτελεστεί σε KVM με QEMU monitor, πλέον το hvt (ο specialized unikernel monitor) αποτελεί τον επίσημο και βασικό στόχο του, παρέχοντας ένα απομονωμένο περιβάλλον εκτέλεσης με άμεση πρόσβαση στο /dev/kvm. Πιο συγκεκριμένα και ειδικά για την αρχιτεκτονική aarch64 που μας απασχόλησε, το hvt αποτελεί τη μοναδική επιλογή διεπαφής του Solo5 με το KVM

4.2.1 Δομή

Η υλοποίηση του Solo5 unikernel βρίσκεται στο φάκελο bindings. Περιέχει την υλοποίηση της διεπαφής μεταξύ των υποστηριζόμενων πλατφορμών (εν προκειμένω του hvt, με τον hvt συγκεκριμένο κώδικα στο bindings/hvt) και της βάσης του unikernel όπως περιγράφεται στο

include/solo5/solo5.h. Στο τελευταίο περιλαμβάνονται οι "δημόσιες" μέθοδοι που μπορεί να χρησιμοποιήσει το πιο πάνω στρώμα (δηλαδή ο χρήστης στις εφαρμογές του, ή κάποια διεπαφή με κάποιο άλλο library OS π.χ MirageOS, IncludeOS) για να επικοινωνήσει με το υλικό και να εκτελέσει βασικές λειτουργίες (ρολόι, δίκτυο, block device κλπ). Οι μεθοδοι αυτές αντιστοιχίζονται σε hypercalls που υλοποιούνται από το hvt.

Στο φάκελο tenders/hvt βρίσκεται η υλοποίηση του hvt. Οι εσωτερικές διεπαφές του hvt προς το KVM ορίζονται στο tenders/hvt/hvt.h και ο aarch64 συγκεκριμένος κώδικας όσον αφορά την υλοποίηση του KVM backend support (αρχικοποίηση του εικονικού επεξεργαστή κλπ) βρίσκεται στο hvt_kvm_aarch64.c. Τέλος περιλαμβάνεται η υλοποίηση των hypercalls του Solo5 τα οποία ορίζονται στο include/solo5/hvt_abi.h.

```
1 /*
2  * Canonical list of hypercalls supported by all tender modules. Actual calls
3  * supported at run time depend on module configuration at build time.
4  */
5 enum hvt_hypercall {
6     /* HVT_HYPERCALL_RESERVED=0 */
7     HVT_HYPERCALL_WALLTIME=1,
8     HVT_HYPERCALL_PUTS,
9     HVT_HYPERCALL_POLL,
10    HVT_HYPERCALL_BLKINFO,
11    HVT_HYPERCALL_BLKWRITE,
12    HVT_HYPERCALL_BLKREAD,
13    HVT_HYPERCALL_NETINFO,
14    HVT_HYPERCALL_NETWRITE,
15    HVT_HYPERCALL_NETREAD,
16    HVT_HYPERCALL_HALT,
17    HVT_HYPERCALL_MAX
18 };
```

Πιο συγκεκριμένα τα παραπάνω hypercalls καλούνται με ενιαίο τρόπο με την hvt_do_hypercall() η υλοποίηση της οποίας για αρχιτεκτονική aarch64 ακολουθεί:

```
1 static inline void hvt_do_hypercall(int n, volatile void *arg)
2 {
3 #   ifdef assert
4     assert(((uint64_t)arg <= UINT32_MAX));
5 #   endif
6         __asm__ __volatile__ ("str %w0, [%1]"
7                 :
8                 : "rZ" ((uint32_t)((uint64_t)arg)),
9                   "r" ((uint64_t)HVT_HYPERCALL_ADDRESS(n))
```

```
10                  : "memory");
11 }
```

4.2.2 Boot

Στη διαδικασία εκκίνησης του Solo5 unikernel το ρόλο της αρχικοποίησης του virtualized environment που παρέχει το KVM αναλαμβάνει το hvt. Όπως φαίνεται και στο αρχείο hvt_main.c αρχικά καλείται η hvt_init() η οποία και δημιουργεί το VM, τον εικονικοποιημένο επεξεργαστή (vcpu) και την απεικόνιση του στη μνήμη καθώς και την απεικόνιση της μνήμης του guest στο χώρο χρήστη. Η διαδικασία μέχρι στιγμής είναι ανεξάρτητη αρχιτεκτονικής (x86_64 ή aarch64). Στη συνέχεια μέσω της hvt_elf_load() φορτώνεται στη μνήμη που αντιστοιχεί στον guest, ο unikernel. Έπειτα καλείται η hvt_vcpu_init() που αρχικοποιεί πλέον τη vcpu (μνήμη, registers κλπ) η υλοποίηση της οποίας είναι φυσικά συγκεκριμένη ως προς την αρχιτεκτονική (αρχείο hvt_kvm_aarch64.c). Τέλος καλείται η κύρια επανάληψη (hvt_vcpu_loop) η οποία και διαχειρίζεται τα διάφορα γεγονότα που οδηγούν σε έξοδο από το απομονωμένο εικονικοποιημένο περιβάλλον του KVM. Αυτή είναι και η πρώτη φορά που αρχίζει να εκτελείται ο unikernel με την εξής ioctl εντολή του KVM API:

```
1 ret = ioctl(hvb->vcpufd, KVM_RUN, NULL);
```

Το παραπάνω ioctl χρησιμοποιείται για να τρέξει ένα εικονικό επεξεργαστή του επισκέπτη. Από το σημείο αυτό και μετά αρχίζει η εκτέλεση της _start() (bindings/hvt/start.c) που αρχικοποιεί πλέον τον πυρήνα του solo5 (κονσόλα, πίνακας εξαιρέσεων του επιπέδου εξαίρεσης EL1) και τυπώνει στην κονσόλα τα πρώτα μηνύματα του Solo5 πριν προχωρήσει στην αρχικοποίηση της μνήμης του guest, του ρολογιού, του δικτύου και τελικά καλέσει την κύρια εφαρμογή του unikernel (solo5_app_main).

Τα παραπάνω φαίνονται και στο σχετικό απόσπασμα (**παραθέτουμε από την hvt_main μέχρι το πρώτο console output του solo5 unikernel**) από την έξοδο του προγράμματος strace για την εντολή εκτέλεσης ενός Solo5 unikernel με το hvt:

```
1 openat(AT_FDCWD, "/dev/kvm", O_RDWR|O_CLOEXEC) = 3
2 ioctl (3, KVM_GET_API_VERSION, 0)          = 12
3 ioctl (3, KVM_CREATE_VM or LOGGER_GET_LOG_BUF_SIZE, 0) = 4
4 ioctl (4, KVM_CREATE_VCPU, 0)              = 5
5 ioctl (3, KVM_GET_VCPU_MMAP_SIZE or LOGGER_FLUSH_LOG, 0) = 8192
6 mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_SHARED, 5, 0) =
      0xffff99497000
7 mmap(NULL, 536870912, PROT_READ|PROT_WRITE,
      MAP_SHARED|MAP_ANONYMOUS, -1, 0) = 0xffff79314000
8 ioctl (4, KVM_SET_USER_MEMORY_REGION, 0xffffc36f6a78) = 0
```

```
 9 openat(AT_FDCWD, "test_hello.ukvm", O_RDONLY) = 6 #hvt_elf_load
10 pread64(6,
       "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\2\0\267\0\1\0\0\0\0\0\20\0\0\0\0\0"...,
       64, 0) = 64
11 pread64(6,
       "\1\0\0\0\5\0\0\0\0\20\0\0\0\0\0\0\0\0\20\0\0\0\0\0\0\0\20\0\0\0\0\0"...,
       168, 64) = 168
12 pread64(6, "\375{\276\251\375\3\0\221\363S\1\251\364\3\0\252\
       3\1\0\224\24\2\0\224\340\3\24\252I\0\0\224"..., 18296, 4096) = 18296
13 mprotect(0xffff79414000, 20480, PROT_READ|PROT_EXEC) = 0
14 pread64(6, "\1\0\0\0\2\0\0\0", 8, 24576) = 8
15 mprotect(0xffff79419000, 8192, PROT_READ|PROT_WRITE) = 0
16 close(6)                                  = 0
17 ioctl(4, KVM_ARM_PREFERRED_TARGET, 0xffffc36f6a78) = 0
18 ioctl(5, KVM_ARM_VCPU_INIT, 0xffffc36f6a78) = 0
19 ioctl(5, KVM_ARM_SET_DEVICE_ADDR or KVM_GET_ONE_REG,
       0xffffc36f6a68) = 0
20 #Get Architectural Feature Access Control Register
21 ioctl(5, KVM_SET_ONE_REG, 0xffffc36f6a68) = 0
22 #Enable the floating-point and Advanced SIMD #for Guest
23 ioctl(5, KVM_SET_ONE_REG, 0xffffc36f6a68) = 0
24 #Setup Memory Attribute Indirection Register EL1, this register must be set
       before setup page tables.
25 ioctl(5, KVM_SET_ONE_REG, 0xffffc36f6a68) = 0
26 #Setup Translation Control Register EL1
27 ioctl(5, KVM_SET_ONE_REG, 0xffffc36f6a68) = 0
28 #Setup Translation Table Base Register 0 EL1. The translation range doesn't
       exceed the 0 ~ 1^64. So the TTBR0_EL1 is enough
29 ioctl(5, KVM_SET_ONE_REG, 0xffffc36f6a68) = 0
30 #Enable MMU and I/D Cache for EL1
31 ioctl(5, KVM_SET_ONE_REG, 0xffffc36f6a68) = 0
32 #Set default PSTATE flags to SPSR_EL1
33 ioctl(5, KVM_SET_ONE_REG, 0xffffc36f6a68) = 0
34 #Set Stack Pointer for Guest. ARM64 require stack be 16-bytes alignment by
       default.
35 ioctl(5, KVM_SET_ONE_REG, 0xffffc36f6a68) = 0
36 #Passing hvt_boot_info through x0
37 ioctl(5, KVM_SET_ONE_REG, 0xffffc36f6a68) = 0
38 #Set guest reset PC entry here
39 ioctl(5, KVM_RUN, 0)                       = 0
40 #First time called by hvt_vcpu_loop
```

```
41 write(1, "                |        ___|\n", 24) = 24
42 #First output in console
43 ioctl(5, KVM_RUN, 0)                    = 0
44 #vmentry
```

Παρατηρούμε ότι από την στιγμή που συμβαίνει το KVM_RUN ioctl μέχρι το πρώτο write στην κονσόλα δεν βλέπουμε κατι ενδιάμεσα, αφού αυτό που συμβαίνει είναι ότι αρχίζει να εκτελείται ο Unikernel στο απομονωμένο περιβάλλον που του παρέχει η εικονικοποίηση του KVM. Η πρώτη έξοδος (vmexit) γίνεται για να τυπώσει στην κονσόλα. Στη συνέχεια συνεχίζει η εκτέλεση του unikernel στον εικονικό επεξεργαστή με το KVM_RUN ioctl (vmentry).

4.2.3 Interaction with higher level stack

Το Solo5 παρέχει μία διεπαφή μεταξύ του hvt και της εφαρμογής. Το hypercall API λειτουργεί ως ο ενδιάμεσος αφαιρετικός τρόπος να επικοινωνήσει το ανώτερο στρώμα με τις συσκευές δικτύου, αποθήκευσης κλπ. Ακριβώς αυτό το επίπεδο αφαίρεσης το καθιστά περισσότερο ως Unikernel base παρά ως συνολικό Unikernel Framework με την έννοια ενός Library Operating System. Με αυτή την έννοια ήδη υπάρχοντα Unikernel Frameworks (βλ. κεφάλαιο 3: MirageOS, IncludeOS, Rumprun) μπορούν σχετικά εύκολα να εκτελεστούν στις υποστηριζόμενες πλατφόρμες του Solo5-hvt χρησιμοποιώντας τον πυρήνα του Solo5 για την αρχικοποίηση του εικονικού υλικού και την επικοινωνία με τον hypervisor.

4.3 MirageOS on top of Solo5

Για να γίνει αντιληπτό το συμπέρασμα της παραγράφου 4.2.3, παραθέτουμε το σχήμα που ακολουθήθηκε για την εκτέλεση του MirageOS πάνω από KVM, με τη χρήση του Solo5 (Σχ. 4.3), σε αντιπαραβολή με τη κλασική πλατφόρμα του MirageOS που είναι το Xen (Σχ. 4.2)

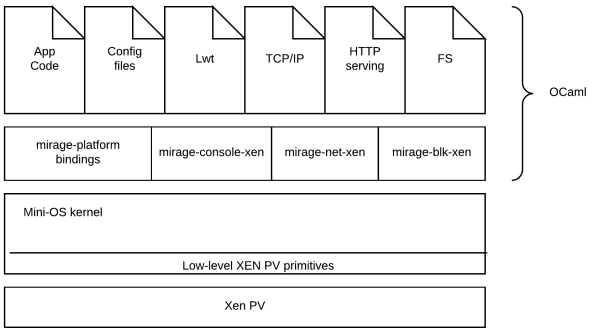


Figure 4.2: MirageOS με Xen[27]

Συνδέσεις πλατφόρμας (Platform bindings)
OCaml runtime : Χρειάζεται ένα υποσύνολο της libc και μερικές Xen-specific functions.
Οδηγοί (Drivers)
Κώδικας σε OCaml με βάση το Xen PV split model, ορισμένες κλήσεις προς την πλατφόρμα.
Βάση Unikernel (Unikernel base)
miniOS: Παρέχει όση libc χρειάζεται η πιο πάνω πλατφόρμα και ορισμένες χαμηλού επιπέδου πληροφορίες από το Xen.
Όλα τα παραπάνω χτίζονται ως μία ενιαία εικόνα VM με τη χρήση του mirage tool.

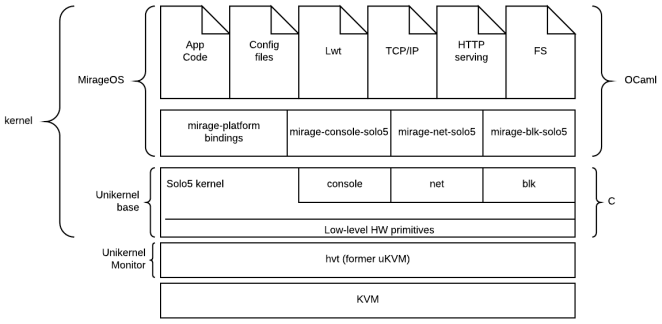


Figure 4.3: MirageOS με KVM σε Solo5[27]

Καμία αλλαγή στον κώδικα της εφαρμογής και των βιβλιοθηκών που έχουν αναπτυχθεί σε OCaml.
Συνδέσεις πλατφόρμας (Platform bindings)
Οι εξαρτήσεις του OCaml runtime και των βιβλιοθηκών της OCaml από στοιχεία της C αντιμετωπιζόταν (στην αρχική περίπτωση του MirageOS - Xen) με την παροχή ενός ενδιάμεσου POSIX-like στρώ-ματος. Κατά τη μεταφορά του MirageOS στο Solo5 ακολουθήθηκε μία πιο δομοστοιχειωτή λογική. Αντί να παρέχεται ένα ψευδο-POSIX στρώμα για την κάλυψη των εξαρτήσεων (π.χ libc), κάθε ανεξάρτητο στοιχείο (OCaml runtime, τυχαία OCaml library) θεωρήθηκε αυτόνομο (π.χ freestanding OCaml runtime) και χτίστηκε συμπεριλαμβάνοντας τα πραγματικά του dependencies (όσα στοιχεία από τη C χρειάζεται).
Οδηγοί (Drivers)
Χρησιμοποιήθηκαν οι drivers του Solo5 (κώδικας σε C) με wrappers σε OCaml. Τα απλά interfaces του solo5 παραμένουν ίδια. Για παράδειγμα το mirage-net-solo5 περιλαμβάνει OCaml

wrappers για τις net διεπαφές του Solo5.

Βάση Unikernel (Unikernel base)

Solo5: Αρχικοποίηση του υλικού, μνήνης, διακοπών. Υλοποίηση χωρίς νήματα και με ενιαίο χώρο διευθύνσεων.

Monitor

hvt (προηγουμένως uKVM): Δημιουργία εικονικής μηχανής, εικονικού επεξεργαστή κλπ στο χαμηλότερο επίπεδο του KVM.

Τέλος με τις κατάλληλες αλλαγές στο εργαλείο - mirage όλα τα παραπάνω χτίζονται ως μια ενιαία εικόνα VM (*.hvt) (εκτός από το εξειδικευμένο monitor (hvt) που χτίζεται ως το χωριστό εκτελέσιμο που τελικά λαμβάνει ως παράμετρο εκτέλεσης την παραγώμενη εικόνα).

Κεφάλαιο 5

Αξιολόγηση Unikernels σε πλατφόρμες SoC

5.1 Εισαγωγή

Στο κεφάλαιο αυτό θα παρουσιάσουμε αποτελέσματα σχετικά με βασικές λειτουργίες (δίκτυο, αποθήκευση, http) του Solo5 καθώς και του MirageOS (με χρήση του Solo5) στις ακόλουθες πλατφόρμες:

1. Xilinx ZynqMP zcu102 (ARM A53 - 4 cores)
    - Αρχιτεκτονική: aarch64
    - Έκδοση πυρήνα: Linux 4.9.0-xilinx-v2017.3
2. AppliedMicro X-Gene1 (8x ARMv8 cores)[28]
    - Αρχιτεκτονική: aarch64
    - Έκδοση πυρήνα: Linux 4.17.0-00001-g010275f

    Εκτός από τις πλατφόρμες SoC, και για λόγους πληρότητας, παραθέτουμε τα αποτελέσματα εκτέλεσης και σε ένα παραδοσιακό προσωπικό υπολογιστή αρχιτεκτονικής Intel x86.
3. Lenovo Carbon X1 5th gen (Intel Core i7-7500U - 4 cores)
    - Αρχιτεκτονική: x86_64
    - Λειτουργικό σύστημα: Debian 4.18.6-1

Στόχος μας είναι να αποτιμήσουμε το overhead της εκτέλεσης μίας εφαρμογής, που χρησιμοποιεί τις παραπάνω βασικές λειτουργίες, στο isolated περιβάλλον που παρέχουν οι Unikernels και να το συγκρίνουμε με το αντίστοιχο ενός Linux Guest σε QEMU/KVM.

Οι Linux Guest που χρησιμοποιήθηκαν στις μετρήσεις αυτού του κεφαλαίου έχουν τις ακόλουθες εκδόσεις πυρήνα:

- Linux 4.17.0-00001-g010275f aarch64 GNU/Linux

- Linux 4.17.0 x86_64 GNU/Linux

5.2 Μετροπρογράμματα

5.2.1 Μέτρηση Επιδόσεων Δικτύου (net test)

Πρόγραμμα στον host

Για την μέτρηση των επιδόσεων της εικονικής συσκευής δικτύου στις παραπάνω πλατφόρμες αναπτύξαμε σε C ένα πρόγραμμα μέτρησης επιδόσεων, το οποίο εκτελείται στο μηχάνημα οικοδεσπότη (host machine). Ο σχετικός κώδικας σε C παρατίθεται στο Παράρτημα Α.1. Ακολουθεί συνοπτικά η περιγραφή της λειτουργίας του:

1. Αρχικά μέσω της συνάρτησης socket() δημιουργείται ένα νέο, αδέσμευτο socket και επιστρέφεται ένας περιγραφητής αρχείου (file descriptor) που μπορεί να χρησιμοποιηθεί απο επόμενες κλήσεις που διαχειρίζονται τα sockets.

2. Χρησιμοποιώντας αυτό το socket ως παράμετρο της ioctl() μαζί με το όνομα της διεπαφής δικτύου του host που έχει "προσδεθεί" στο guest, λαμβάνουμε τη διεύθυνση MAC της συγκεκριμένης διεπαφής η οποία είναι απαραίτητη για να χτίσουμε το ethernet header των πακέτων που σκοπεύουμε στη συνέχεια να στείλουμε στον guest (για το πεδίο του ethernet header που αντιστοιχεί στην διεύθυνση MAC του αποστολέα).

3. Για να μάθουμε τη MAC διεύθυνση του guest, φτιάχνουμε ένα ARP αίτημα ενθυλακωμένο σε ένα ethernet πακέτο συμπληρώνοντας τα αντίστοιχα πεδία (με μηδενικές τιμές στη διεύθυνση MAC του παραλήπτη και τη διεύθυνση IP 10.0.0.2 που συμβατικά έχουμε παραχωρήσει στον guest/Solo5 για τους σκοπούς των μετρήσεων)

4. Στη συνέχεια ανοίγουμε ξανά ένα socket (όπως πριν) και με τη συνάρτηση sendto() στέλνουμε, μέσω του socket, το ARP αίτημα ρωτώντας ποια είναι η MAC του guest.

5. Στη συνέχεια μέσω της recv() λαμβάνουμε όλα τα πακέτα που στέλνονται στο socket που έχουμε ανοίξει. Μόλις λάβουμε την απάντηση στο ARP πακέτο που στείλαμε, έχουμε μάθει την MAC του guest και κλείνουμε το socket.

6. Φτιάχνουμε ένα πακέτο ICMP echo request. Ανάλογα με το μέγεθος των data που επιλέγει ο χρήστης να στείλει, "γεμίζουμε" το πεδίο data του ICMP πακέτου με ίδιο αριθμό από χαρακτήρες του ενός byte. Αντίστοιχα οριζουμε το ICMP header, το IP header και το ethernet header και με τη χρήση της memcpy() αντιστοιχίζουμε όλα τα παραπάνω στη διεύθυνση μνήμης που αντιστοιχεί στο πακέτο ethernet που θα στείλουμε.

   Πλέον μπορούμε να προχωρήσουμε στην αποστολή και λήψη πακέτων και στην μέτρηση του χρόνου:

7. Ανοίγουμε ξανά ένα socket μέσω του οποίου θα γίνεται η αποστολή αιτημάτων (ICMP echo request) και η λήψη των αντίστοιχων απαντήσεων (ICMP echo reply).

8. Μέσω της bind() αντιστοιχίζουμε στο socket τη διεύθυνση MAC του αποστολέα (της διεπαφής δικτύου του host δηλαδή) ώστε ο πυρήνας να μην "προωθεί" στο socket πακέτα που δεν έχουν ως παραλήπτη τον host. Αυτό είναι απαραίτητο ώστε να μην επηρεάζονται οι μετρήσεις μας από "άσχετα" πακέτα που θα λάμβανε / "άκουγε" το socket.

9. Μέσω της clock_gettime() λαμβάνουμε μία τιμή για το χρόνο αποστολής (t1). Αμέσως μετά στέλνουμε με την sendto() το πακέτο icmp που περιγράψαμε πριν και καλούμε την recv() αναμένοντας την απάντηση σε αυτό το πακέτο.

10. Μόλις λάβουμε την απάντηση που αναμένουμε, ξανακαλούμε την clock_gettime() για το χρόνο λήψης (t2).

Ο χρόνος t2-t1 είναι ο χρόνος που αντιστοιχεί στην αποστολή ενός πακέτου στον guest και στη λήψη της αντίστοιχης απάντησης από τον host. Η διαδικασία αυτή (τα βήματα 9-10) γίνεται επαναληπτικά (αυξάνοντας αντιστοίχως το seq number στο icmp header κάθε φορά) όσες φορές έχει ορίσει ο χρήστης.

Πρόγραμμα στον guest

Linux Guest: Το πρωτόκολλο ICMP είναι υλοποιημένο απευθείας πάνω από το IP stack του Linux, και (εκτός αν έχει γίνει κάποια διαφορετική ρύθμιση στον πυρήνα του guest) απαντάει στα ICMP echo requests σύμφωνα με το πρωτόκολλο.

Solo5: Δεν υπάρχει ολόκληρο IP stack υλοποιημένο. Βασιστήκαμε στο πρόγραμμα test_ping_serve που περιλαμβάνεται στο Solo5. Η εφαρμογή αυτή χρησιμοποιεί το net module του Solo5 και τα solo5_net_read και solo5_net_write hypercalls για να διαβάζει / στέλνει πακέτα από / προς τη διεπαφή δικτύου. Μόλις λάβει ένα πακέτο ελέγχει τη διεύθυνση παραλήπτη και τον τύπο του πακέτου (icmp request) και στην περίπτωση που απευθύνεται στον guest, απλά αλλάζει τα πεδία αποστολέα - παραλήπτη και το πεδίο που αντιστοιχεί στον τύπο του πακέτου από Request σε Reply. Στη συνέχεια μέσω της solo5_net_write "στέλνει" το πακέτο - απάντηση στη διεπαφή δικτύου.

Solo5 - MirageOS: Το MirageOS περιλαμβάνει βιβλιοθήκες που υλοποιούν τα βασικά πρωτόκολλα δικτύου και χρησιμοποιεί στο χαμηλότερο επίπεδο τα hypercalls του Solo5 για την επικοινωνία με τη διεπαφή δικτύου (βλ. παράγραφο 4.3)

5.2.2 Μέτρηση επιδόσεων κλήσεων προς την εικονική συσκευή αποθήκευσης (virtual block device test)

Σκοπός του προγράμματος αυτού είναι να μετρήσει τις επιδόσεις ενός read call από τον guest σε ένα virtual block device που αντιστοιχεί σε ένα raw αρχείο - εικόνα του host. Το συγκεκριμένο test χρησιμοποιεί την συνάρτηση pread της libc στον Linux Guest και τη

solo5_blk_read στο Solo5 Guest επαναληπτικά διαβάζοντας 512 bytes τη φορά για 100.000 επαναλήψεις. Με την χρηση της clock_gettime μετράμε το συνολικό χρόνο για τις 100.000 επαναλήψεις. Ο κώδικας βρίσκεται στα Παραρτήματα Α.2 και Α.3 αντίστοιχα.

Περαιτέρω ανάλυση του χρόνου

Ωστόσο το παραπάνω δεν αρκεί για να αξιολογήσουμε πλήρως τις επιδόσεις καθώς αφορά το συνολικό χρόνο από τη στιγμή που ο guest θα κάνει το read request μέχρι τη στιγμή που θα επιστρέψει η pread. Αυτός ο χρόνος περιλαμβάνει την έξοδο από το απομονωμένο περιβάλλον του KVM (vmexit) , τη διαχείριση του αιτήματος από το πρόγραμμα φύλακα (monitor) που περιλαμβάνει το system call από το χώρο χρήστη του host (όπου εκτελείται το monitor) στο χώρο του host πυρήνα και ανάποδα (vmenter) μέχρι τα δεδομένα να φτάσουν στον guest. Για το λόγο αυτό, επειδή ακριβώς θέλουμε να μελετήσουμε το overhead που προσθέτει η κάθε υλοποίηση (QEMU/KVM, Solo5/hvt) προχωρήσαμε σε ορισμένες αλλαγές στα προγράμματα monitor (QEMU, hvt) ώστε να μετράμε το χρόνο του syscall στον host. Αφαιρώντας αυτό το χρόνο από το συνολικό μπορούμε να έχουμε μια πιο συγκεκριμένη εικόνα για το overhead της κάθε υλοποίησης. Τις αλλαγές αυτές παραθέτουμε στο Παράρτημα Β.1 με τη μορφή diff σε σχέση με την αντίστοιχη επίσημη έκδοση όπως αυτή υπάρχει στα αντίστοιχα επίσημα git repositories.

5.3 Πείραμα 1: Μέτρηση Επιδόσεων Δικτύου

5.3.1 Προετοιμασία

Στο host σύστημα δημιουργούμε μια εικονική διεπαφή δικτύου και της αναθέτουμε ένα σύνολο διευθύνσεων IP ως εξής:

```
$ ip tuntap add tap100 mode tap
$ ip addr add 10.0.0.1/24 dev tap100
$ ip link set dev tap100 up
```

Στη συνέχεια εκκινούμε το guest VM αναθέτοντας του την παραπάνω διεπαφή και μία IP που ανήκει στο σύνολο διευθύνσεων που εξυπηρετούνται από αυτή τη διεπαφή (10.0.0.2).

Στο Παράρτημα Β.2 παραθέτουμε τις σχετικές εντολές για την εκκίνηση του QEMU Linux Guest και του Solo5 Unikernel.

5.3.2 Αποτελέσματα

Θα παραθέσουμε τα αποτελέσματα εκτέλεσης του μετρο-προγράμματος 1 σε κάθε πλατφόρμα αξιολόγησης. Οι μετρήσεις που ακολουθούν αφορούν το roundtrip 100.000 πακέτων icmp από και προς τον guest για τα εξής μεγέθη icmp data: 1 byte, 512 bytes, 1024 bytes, 1472 bytes (MTU). Σε κάθε μία από τις ARM πλατφόρμες συγκρίνουμε τις εξής τεχνολογίες εικονικοποίησης:

- Solo5
- Linux Guest (QEMU/KVM)
- MirageOS on Solo5

Επίσης για λογους αναφοράς θα παραθέσουμε και μία σύγκριση των 2 πρώτων στην πλατφόρμα x86_64.

Ακολουθούν τα αποτελέσματα που αφορούν το Μέσο Όρο (των πακέτων ανά δευτερόλεπτο) από 5 εκτελέσεις του προγράμματος μέτρησης επιδόσεων του δικτύο, που περιγράψαμε παραπάνω.

Πλατφόρμα -Xilinx ZynqMP zcu102 (ARM Cortex A53 - αρχιτεκτονική aarch64)

Πλατφόρμα / μέγεθος	1 byte	512 bytes	1024 bytes	1472 bytes (MTU)
Solo5	38295.62298	29992.52519	28116.2606	26509.80456
MirageOS on Solo5	19607.56478	15774.63349	15576.23435	15528.79682
Linux Guest (QEMU)	14417.60182	13621.23929	13185.32434	12827.88068

Table 5.1: Μετρήσεις επίδοσης δικτύου - Xilinx ZynqMP zcu102 (ARM Cortex A53)

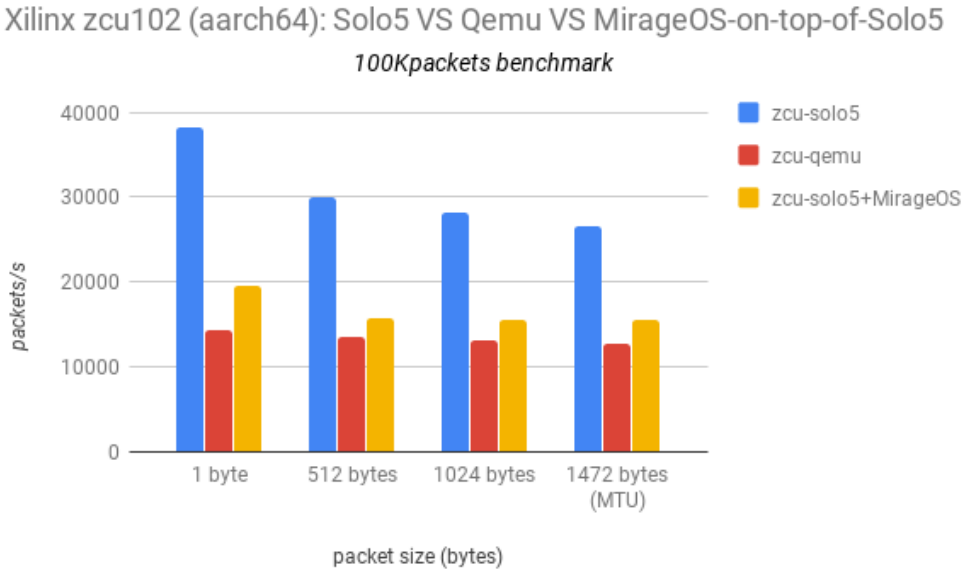


Figure 5.1: Xilinx ZynqMP zcu102 (ARM Cortex A53) - network benchmark

Πλατφόρμα AppliedMicro® X-Gene1 (ARMv8 - αρχιτεκτονική aarch64)

Πλατφόρμα / μέγεθος	1 byte	512 bytes	1024 bytes	1472 bytes (MTU)
Solo5	46139.63723	46779.95462	47548.55658	36401.99586
MirageOS on Solo5	29072.75915	26577.98929	17009.67067	16556.67353
Linux Guest (QEMU)	28897.25052	27532.70096	27128.95936	26556.2585

Table 5.2: Μετρήσεις επίδοσης δικτύου - AppliedMicro® X-Gene1 (ARMv8)

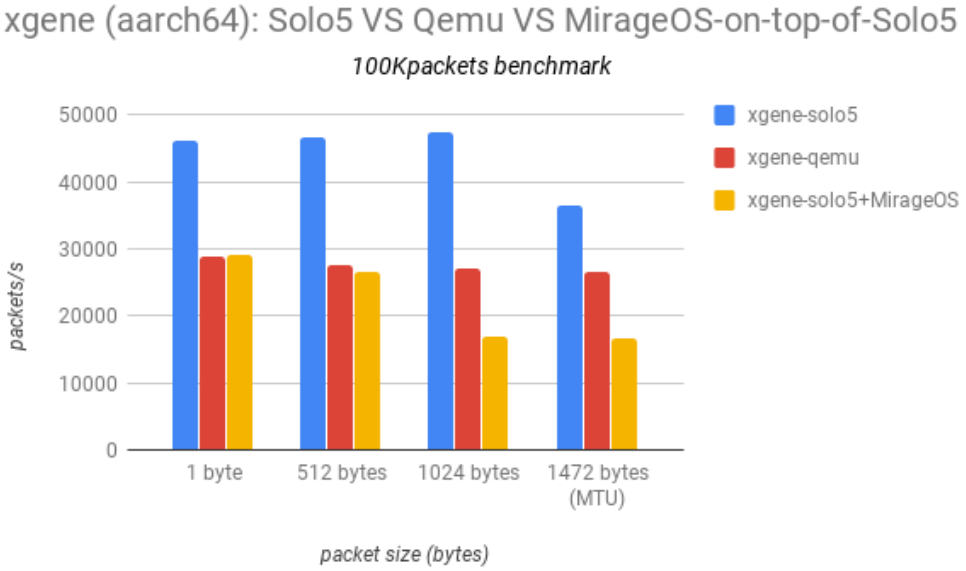


Figure 5.2: AppliedMicro® X-Gene1 (ARMv8) - network benchmark

Πλατφόρμα Lenovo Carbon X1 5th Gen (Intel Core i7 7500U - αρχιτεκτονική x86_64)

Πλατφόρμα / μέγεθος	1 byte	512 bytes	1024 bytes	1472 bytes (MTU)
Solo5	93604.17084	92656.75815	93662.51174	94914.62217
Linux Guest (QEMU)	70809.59258	71175.57828	66738.84606	68531.20974

Table 5.3: Μετρήσεις επίδοσης δικτύου - Lenovo Carbon X1 (Intel i7 7500U)

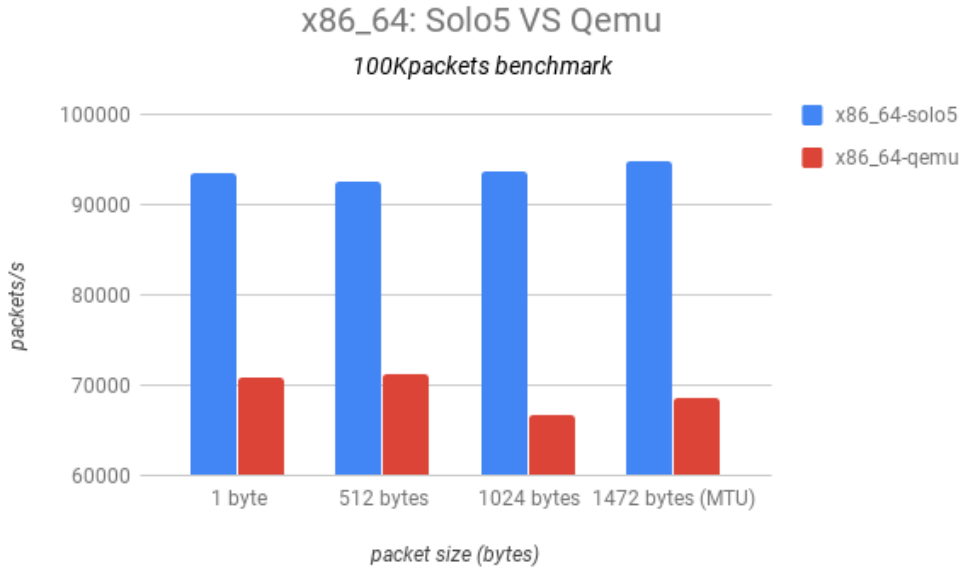


Figure 5.3: Lenovo Carbon X1 (Intel i7 7500U) - network benchmark

5.3.3 Συμπεράσματα

Παρατηρούμε ότι η μινιμαλίστικη και απλουστευμένη υλοποίηση των διεπαφών δικτύου στο Solo5 παρουσιάζει αρκετά καλύτερες επιδόσεις σε σχέση με το QEMU Linux Guest. Είναι αξιοσημείωτο ότι παρόλο που το icmp echo request προς το Linux Guest το διαχειρίζεται ο πυρήνας του, και επομένως η αποστολή του κάθε icmp echo reply δεν απαιτεί μεταβάσεις προς το χώρο χρήστη και ανάποδα, οι επιδόσεις του QEMU παραμένουν αισθητά χειρότερες σε όλες τις πλατφόρμες.

Αξίζει ωστόσο να σημειωθεί ότι στην περίπτωση του Solo5 εκτελείται μία πολύ απλή εφαρμογή (test_ping_serve) η οποία με το που διαβάσει από τη διεπαφή δικτύου ένα εισερχόμενο πακέτο κάνει πολύ λίγα πράματα: ελέγχει ότι ο unikernel είναι όντως ο παραλήπτης (MAC), ελέγχει εάν το πακέτο είναι τύπου icmp request (ή/και ARP) και αφού ανταλλάξει τα αντίστοιχα πεδία των διευθύνσεων παραλήπτη - αποστολέα καθώς και το icmp type από request σε reply γράφει το πακέτο απευθείας στον buffer που το hvt έχει αντιστοιχίσει στη διεπαφή δικτύου (tap device). Η διαχείριση αυτή είναι σίγουρα πιο "ελαφριά" από την υλοποίηση ολόκληρου του TCP/IP stack (και του ICMP πάνω από το IP) στην περίπτωση του Linux Guest.

Το παραπάνω επιβεβαιώνεται και από τις μετρήσεις του MirageOS με το Solo5 ως backend. Παρόλο που και σε αυτή την περίπτωση χρησιμοποιούνται τα hypercalls του Solo5 (και επομένως θα αναμέναμε αντίστοιχες επιδόσεις με το Solo5 ή έστω λίγο χειρότερες), παρατηρούμε ότι το MirageOS είναι οριακά καλύτερο από το QEMU Linux Guest (σε πακέτα μικρού μεγέθους) και αρκετά χειρότερο από το Solo5. Αυτό δεν μπορεί να οφείλεται στο backend, αφού είναι το ίδιο με το Solo5, επομένως αφορά τη διαχείριση των icmp echo request στο υψηλότερο επίπεδο της υλοποίησης των διαφόρων πρωτοκόλλων δικτύου από τις OCaml βιβλιοθήκες του MirageOS. Όπως αναφέραμε και πριν είναι αναμενόμενο αυτό να προσθέτει κάποιο overhead, ωστόσο η διαφορά Solo5 με MirageOS + Solo5 *χρήζει περαιτέρω διερεύνησης*.

Σε κάθε περίπτωση από τα παραπάνω φαίνεται ότι μπορούμε να πετύχουμε αύξηση των επιδόσεων του δικτύου στα εικονικά περιβάλλοντα μέσω της χρήσης λεπτών στρωμάτων λογισμικού, όπως το Solo5 και της αποδοτικής αξιοποίησης των απλών hypercall που παρέχει στο API του, για τους σκοπούς της εφαρμογής που μας ενδιαφέρει.

5.4 Πείραμα 2 : Μέτρηση επιδόσεων κλήσεων προς την εικονική συσκευή αποθήκευσης

5.4.1 Προετοιμασία

Με την παρακάτω εντολή δημιουργούμε ένα αρχείο-εικόνα (image) στον αποθηκευτικό χώρο του host συστήματος:

```
dd if=/dev/zero of=/path/to/outfile count=200000
```

και στη συνέχεια μέσω της μεθόδου losetup του Linux αντιστοιχίζουμε το παραπάνω αρχείο με μία loop συσκευή του Linux host. Τη συσκευή /dev/loop0 *δίνουμε* στο guest (QEMU, Solo5) ως παράμετρο για να συσχετιστεί με το virtual block device του.

Στο Παράρτημα B.2 παραθέτουμε τις σχετικές εντολές για την εκκίνηση του QEMU Linux Guest και του Solo5 Unikernel.

5.4.2 Αποτελέσματα

Θα παραθέσουμε τα αποτελέσματα εκτέλεσης του μετρο-προγράμματος 2 σε κάθε πλατφόρμα αξιολόγησης. Οι μετρήσεις που ακολουθούν αφορούν το Μέσο Όρο 10 επαναλήψεων για 100.000 κλήσεις της read από τον guest (pread στον Linux Guest, solo5_block_read στο Solo5) για 512 bytes δεδομένων. Σε κάθε μία από τις πλατφόρμες συγκρίνουμε τις εξής τεχνολογίες εικονικοποίησης:

- Solo5
- Linux Guest (QEMU/KVM)

Όπως σημειώθηκε και στην περιγραφή του μετρο-προγράμματος 2 τα αποτελέσματα περιλαμβάνουν 3 πεδία τιμών.

- Total: Συνολικός χρόνος εκτέλεσης των κλήσεων pread/solo5_block_read από το Linux guest/Solo5 αντίστοιχα
- Pread syscall: Χρόνος εκτέλεσης των κλήσεων pread (syscall) από το χώρο χρήστη στο χώρο πυρήνα του host συστήματος.
- Guest + monitor: Η τιμή αυτή είναι η διαφορά των παραπάνω δύο τιμών και αντιπροσωπεύει το χρόνο που αντιστοιχεί στον guest και στη διαχείριση των εξόδων από το απομονωμένο περιβάλλον εικονικοποίησης του KVM (guest + vmexits + vmentries)

Αξίζει να σημειωθεί ότι το QEMU εξυπηρετεί σε ξεχωριστό thread τα αιτήματα I/O σε αντίθεση με το Solo5 που εκτελείται σε ένα thread. Αναθέτοντας με την εντολή taskset ολόκληρη τη διεργασία του QEMU σε έναν πυρήνα του επεξεργαστή τα αποτελέσματα επηρεάζονται αρνητικά. Για το λόγο αυτό παραθέτουμε 2 αποτελέσματα για το QEMU Linux Guest:

- **α) qemu-notaskset:** η διεργασία δεν ανατίθεται σε κανένα πυρήνα επομένως τα threads εκτελούνται παράλληλα σε διαφορετικούς πυρήνες.
- **β) qemu-1 core:** η διεργασία ανατίθεται σε έναν πυρήνα επομένως όλα τα threads εκτελούνται σε ένα πυρήνα μόνο.

Τέλος για λόγους αναφοράς παραθέτουμε επίσης τα αποτελέσματα εκτέλεσης του μετρο-προγράμματος απευθείας στο host σύστημα κάθε πλατφόρμας αναμένοντας η μέτρηση του χρόνου σε αυτή την περίπτωση να έχει παρόμοια τάξη μεγέθους με τη μέτρηση pread syscall για όλες τις τεχνολογίες εικονικοποίησης.

Πλατφόρμα -Xilinx ZynqMP zcu102 (ARM Cortex A53 - αρχιτεκτονική aarch64)

Τεχνολογία	Total (ms)	pread syscall (ms)	Guest + monitor (ms)
Host	1469.22034	1469.22034	-
Solo5 / hvt	2213.496094	1788.517095	424.978999
QEMU-notaskset (4 cores)	10862.00394	1883.089059	8978.914877
QEMU-1 core	15586.46351	2103.822409	13482.6411

Table 5.4: Μετρήσεις επίδοσης read block device - Xilinx ZynqMP zcu102 (ARM Cortex A53)

Το ακόλουθο διάγραμμα απεικονίζει την κατανομή του χρόνου εκτέλεσης:

Figure 5.4: Xilinx ZynqMP zcu102 (ARM Cortex A53) - read execution time breakdown

Πλατφόρμα AppliedMicro® X-Gene1 (ARMv8 - αρχιτεκτονική aarch64)

Τεχνολογία	Total (ms)	pread syscall (ms)	Guest + monitor (ms)
Host	1593.84754	1593.84754	-
Solo5 / hvt	2093.493618	1676.823068	416.67055
QEMU-notaskset (8 cores)	7161.45227	1624.726738	5536.725532
QEMU-1 core	16333.07108	2857.693816	13475.37727

Table 5.5: Μετρήσεις επίδοσης read block device - AppliedMicro® X-Gene1 (ARMv8)

Το ακόλουθο διάγραμμα απεικονίζει την κατανομή του χρόνου εκτέλεσης:

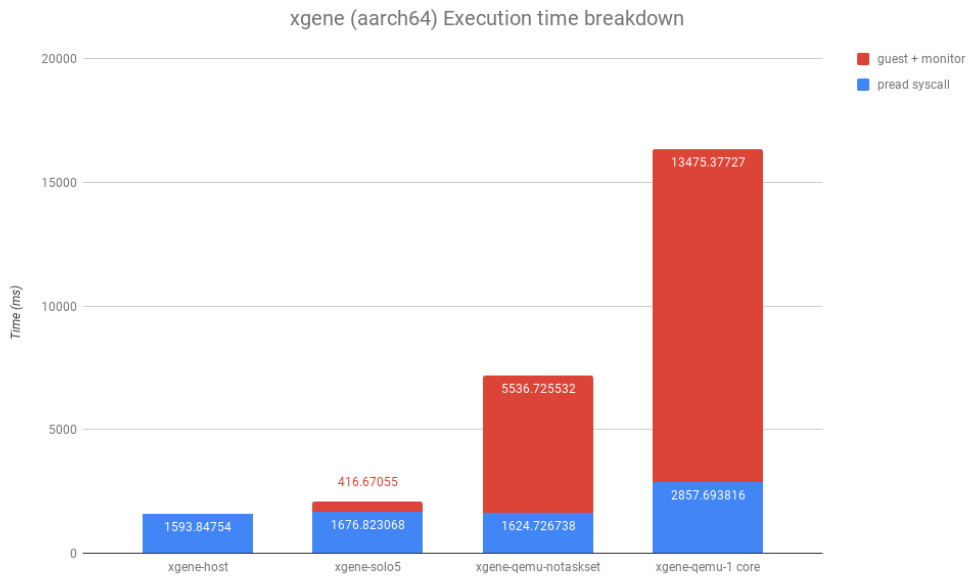


Figure 5.5: AppliedMicro® X-Gene1 (ARMv8) - read execution time breakdown

Πλατφόρμα Lenovo Carbon X1 5th Gen (Intel Core i7 7500U - αρχιτεκτονική x86_64)

Τεχνολογία	Total (ms)	pread syscall (ms)	Guest + monitor (ms)
Host	522.845858	522.845858	-
Solo5 / hvt	856.6942621	610.3461124	246.3481497
QEMU-notaskset (4 cores)	3616.324769	779.9452482	2836.379521
QEMU-1 core	5026.145941	797.6488743	4228.497066

Table 5.6: Μετρήσεις επίδοσης read block device - Lenovo Carbon X1 (Intel i7 7500U)

Το ακόλουθο διάγραμμα απεικονίζει την κατανομή του χρόνου εκτέλεσης:

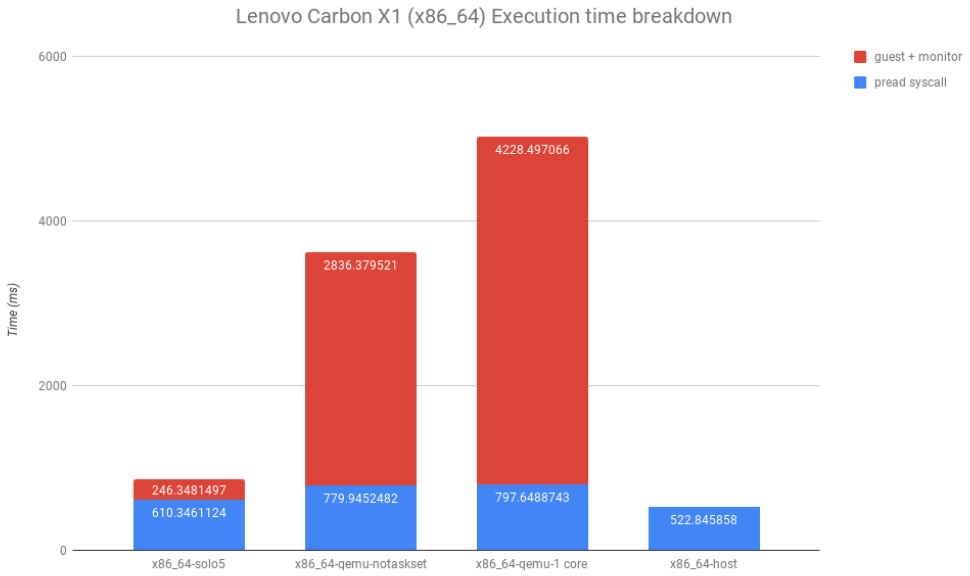


Figure 5.6: Lenovo Carbon X1 (Intel i7 7500U) - read execution time breakdown

5.4.3 Συμπεράσματα

Παρατηρούμε ότι σε όλες τις πλατφόρμες το Solo5 προσθέτει πολύ μικρότερο overhead σε ότι αφορά οτιδήποτε δεν εκτελείται στον host. Ενώ ο χρόνος που παίρνουν οι κλήσεις της pread από το host user space είναι περίπου ίδιος για κάθε monitor (hvt ή QEMU) - και ελάχιστα μεγαλύτερος από την αντίστοιχη εκτέλεση του προγράμματος στον host (αναμενόμενο) - στην περίπτωση του QEMU Linux Guest ο χρόνος που αντιστοιχεί στον Guest (Syscall από Guest user space σε Guest kernel space) και στην διαχείριση από το QEMU των εξόδων από το εικονικοποιημένο περιβάλλον εκτέλεσης (KVM) είναι πολλαπλάσιος σε σχέση όχι μονο με το Solo5 αλλά και με το χρόνο που αντιστοιχεί στον host.

Αντιθέτως στο Solo5 δεν υπάρχει χρόνος που αντιστοιχεί σε syscall (του Guest) αφού το image είναι single address space επομένως δεν υπάρχει το overhead των συνεχόμενων

mode switches. Το γεγονός αυτό, σε συνδυασμό με την ελαφριά υλοποίηση του hvt που δεν προσθέτει πολλαπλά στρώματα από τη στιγμή του vmexit μέχρι να κάνει το syscall στον host, είναι κατά την άποψη μας και ο λόγος που το overhead του Solo5 είναι τόσο αισθητά μικρότερο από το QEMU Linux Guest, και οριακά συναγωνίζεται την εκτέλεση απευθείας στον host.

Τέλος αξίζει να σημειωθεί ότι η μη ύπαρξη νημάτων και η εκτέλεση του Solo5 σε ένα thread είναι ένας ακόμη λόγος για τον οποίο παρουσιάζει καλύτερες επιδόσεις.

Κεφάλαιο 6

Επίλογος

Στο κεφάλαιο αυτό συνοψίζουμε τη μελέτη που εκπονήθηκε στα πλαίσια αυτής της εργασίας, παρουσιάζοντας τα συνολικά συμπεράσματα όσον αφορά τα πλεονεκτήματα και μειονεκτήματα της εκτέλεσης εφαρμογών ως Unikernels. Τέλος παρουσιάζουμε τις πιθανές πτυχές βελτίωσης που εντοπίσαμε, οι οποίες κατά την άποψή μας αποτελούν και αντικείμενο μελλοντικών μελετών.

6.1 Σύνοψη και συμπεράσματα

Η ραγδαία επικράτηση του Cloud Computing θέτει αναπόφευκτα την ανάγκη για εξοικονόμηση επεξεργαστικής ισχύς τόσο για την κάλυψη περισσότερων αναγκών με λιγότερους υπολογιστικούς πόρους όσο και για την κατανάλωση λίγότερης ενέργειας. Καθώς αυξάνεται ο όγκος των δεδομένων προς επεξεργασία όμως, τόσο αυξάνεται και η ανάγκη για επεξεργαστική ισχύ οδηγώντας σε μεγάλα και ενεργοβόρα συστήματα. Στα πλαίσια αυτής της παρατήρησης και της παραδοχής ότι ο όγκος των δεδομένων θα αυξάνεται συνεχώς, η εργασία αυτή μελετά τρόπους που εστιάζουν στην αποδοτικότερη επεξεργασία των δεδομένων και ιδιαίτερα σε συστήματα χαμηλής ενεργειακής κατανάλωσης όπως είναι οι επεξεργαστές ARM. Ένας σημαντικός παράγοντας που επηρεάζει την ποσότητα των απαιτούμενων πόρων από μια εφαρμογή είναι το ίδιο το περιβάλλον εκτέλεσής της. Η εκτέλεση των λειτουργιών στα πλαίσια ενός γενικού σκοπού Λειτουργικού Συστήματος μπορεί μεν να προσφέρει τα πλεονεκτήματα της ευχρηστίας ή της ευκολότερης ανάπτυξης της, ωστόσο έρχεται με το κόστος των πρόσθετων λειτουργιών που προσφέρει ακριβώς το γεγονός ότι αυτό είναι γενικού σκοπού.

Ακολουθώντας μία διαφορετική προσέγγιση, που θέλει το *Λειτουργικό Σύστημα* να είναι tailored στις ανάγκες μίας εφαρμογής και όχι το ανάποδο, μπορούμε να επανασχεδιάσουμε τα περιβάλλοντα εκτέλεσης των εφαρμογών ώστε να περιλαμβάνουν μόνο ό,τι πραγματικά χρειάζεται η εκτέλεσή της πάνω από το υλικό. Η προσέγγιση αυτή καλύπτεται από τους Unikernel, οι οποίοι αποτελούν ανεξάρτητες εικόνες ενιαίου χώρου διευθύνσεων ικανές να εκτελεστούν αυτόνομα, περιλαμβάνοντας όλα και μόνο τα στοιχεία που χρειάζονται για να το πετύχουν αυτό. Ακόμη και σε αυτή τη προσέγγιση υπάρχουν διαφορετικές οπτικές. Η επαναδιαμόρφωση του νεφοϋπολογιστικού περιβάλλοντος εκτέλεσης της εφαρμογής μπορεί

είτε να σημαίνει την εξαρχής επανασχεδίαση και ανάπτυξη των στοιχείων που παραδοσιακά παρείχε το Λειτουργικό Σύστημα είτε την επαναχρησιμοποίηση ήδη υπαρκτών.

Η πρώτη περίπτωση φέρει το κόστος της προσπάθειας που απαιτεί αλλά και το πλεονέκτημα της δυνατότητας αποδοτικού επανασχεδιασμού. Την προσέγγιση αυτή ακολουθούν μια σειρά από Unikernel projects όπως είναι: (i)το MirageOS (OCaml), (ii) το IncludeOS (C++), (iii) το OSv (Java/C++) που περιλαμβάνουν ένα μεγάλο εύρος βιβλιοθηκών, βασικά και χρήσιμα στοιχεία (TCP/IP, DNS κλπ) επιτρέποντας την εκτέλεση παραδοσιακών εφαρμογών στο Cloud.

Η δεύτερη περίπτωση φέρει το πλεονέκτημα του ευκολότερου porting των υπαρκτών εφαρμογών που ακολουθούν τα παραδοσιακά πρότυπα (POSIX) καθώς και την προφανή μη ανάγκη επαναπρογραμματισμού των βιβλιοθηκών και των drivers. Ωστόσο ο κατακερματισμός ενός Λειτουργικού Συστήματος σε αυτόνομες επαναχρησιμοποιούμενες μονάδες δεν είναι εύκολη υπόθεση, ιδιαίτερα αν αναλογιστούμε ότι κατά τον αρχικό σχεδιασμό τους δεν υπήρχε μάλλον η ανάγκη και το κριτήριο της αποφυγής (κατά το δυνατόν) των πολλαπλών αλληλοεξαρτήσεων. Την προσέγγιση αυτή ακολουθούν οι Rump Kernels που ουσιαστικά παρέχουν ένα τρόπο κατακερματισμού του NetBSD σε αυτόνομες μονάδες που μπορούν να συνδυαστούν με διαφορετικούς τρόπους ανάλογα με τις ανάγκες μίας συγκεκριμένης εφαρμογής.

Μία καλή εναλλακτική φαίνεται να είναι το Solo5 το οποίο αποτελεί περισσότερο μία βάση Unikernel, παρέχοντας ένα hypercall API που επιτρέπει την εκτέλεση μίας εφαρμογής αξιοποιώντας το KVM, και θέτει ως στόχο την εύκολη επαναχρησιμοποίησή του ως middleware από άλλα projects όπως το MirageOS, το IncludeOS και το Rumprun (ο παραγόμενος Unikernel των Rump Kernels).

Παρά τις διαφοροποιήσεις που εντοπίζουμε στους διάφορους Unikernels, οι οποίες φυσικά αντικατοπτρίζονται σε διαφορές σε παράγοντες όπως ο χρόνος εκκίνησης, το μέγεθός τους και το αποτύπωμά τους στη μνήμη, το κοινό χαρακτηριστικό τους είναι ο ενιαίος χώρος διευθύνσεων και η έλλειψη των context switches που ένας Linux Guest θα είχε. Οι κλήσεις που εξυπηρετούν αιτήματα Ε/Ε δεν ξεκινάνε από το χώρο χρήστη προς τον χώρο πυρήνα του guest, αφού δεν υπάρχουν διαφορετικοί χώροι. Αντιθέτως προκαλούν άμεσα έξοδο (trap) απο το εικονικό περιβάλλον εκτέλεσης (π.χ KVM) και η εξυπηρέτησή τους ξεκινά από το monitor πρόγραμμα στο χώρο χρήστη του host. Ιδιαίτερα στην περίπτωση του Solo5 τα hypercalls είναι αρκετά απλά (π.χ ένα read request προς το virtual block device προκαλεί ένα vmexit και το read από έναν buffer) και εξυπηρετούνται στο host από ένα αρκετά αποδοτικό και ελαφρύ πρόγραμμα monitor (hvt).

Αφορμώμενοι από την παραπάνω παρατήρηση μετράμε τις επιδόσεις των βασικότερων Ε/Ε λειτουργιών που εκτελούνται σε ένα σύστημα Cloud, δηλαδή των αιτημάτων προς τη συσκευή δικτύου και το block device και αναλύουμε το χρόνο εκτέλεσής τους. Η αξιολόγηση έγινε σε πλατφόρμες ARM χαμηλής ενεργειακής κατανάλωσης. Παράλληλα για λόγους αναφοράς αλλά και επιβεβαίωσης των ισχυρισμών μας εκτελέσαμε τα αντίστοιχα πειράματα και σε πλατφόρμες Intel x86_64. Επιλέξαμε το Solo5 ως βασική τεχνολογία τόσο για λόγους συμβατότητας με το KVM σε ARM αρχιτεκτονική, αλλά και γιατί διευρύνει το πεδίο υποστήριξης και άλλων

Unikernel Frameworks στην ίδια αρχιτεκτονική, όπως είναι το MirageOS. Ως τεχνολογία σύγκρισης επιλέξαμε το QEMU/KVM Linux Guest.

Αυτό που συμπεραίνουμε από τις μετρήσεις των επιδόσεων του δικτύου - μέσω της εκτέλεσης ενός προγράμματος που στέλνει icmp πακέτα από το host προς την guest πλατφόρμα και αναμένει απάντηση - είναι ότι όντως το Solo5 με το hvt παρουσιάζει αισθητά καλύτερες επιδόσεις από έναν κλασικό Linux Guest. Αξιοσημείωτο είναι ότι αυτό συμβαίνει παρόλο που στην περίπτωση του Linux Guest η εξυπηρέτηση του icmp request γίνεται απευθείας από τον πυρήνα του χωρίς να εμπλέκεται το user space. Ένα ακόμη σημαντικό στοιχείο είναι ότι το QEMU χρησιμοποιεί το vhost interface επομένως τα αιτήματα του guest εξυπηρετούνται απευθείας στον πυρήνα του host. Ουσιαστικά δηλαδή η όποια μεταφορά δεδομένων γίνεται μεταξύ των πυρήνων των host-guest και δεν εκτελείται κάποιο read/write syscall στη συσκευή δικτύου. Αντιθέτως στο Solo5, το hvt ως user space εφαρμογή εξυπηρετεί τις εξόδους που προκαλεί το αίτημα του Guest για write στη συσκευή δικτύου, επομένως χρεώνεται με το κόστος του αντίστοιχου syscall προς τον host kernel. Ομοίως και για το read από τη συσκευή δικτύου: το poll στον file descriptor της συσκευής δικτύου γίνεται επίσης από το χώρο χρήστη του host.

Όσον αφορά τις επιδόσεις των κλήσεων πρόσβασης στο Block Device, το Solo5 παρουσιάζει εντυπωσιακά χαμηλότερο χρόνο σε σχέση με έναν Linux Guest. Η ανάλυση του συνολικού χρόνου στις 2 περιπτώσεις επιβεβαιώνει την αρχική μας υπόθεση ότι είναι εφικτό να αποφύγουμε το overhead του Guest με την εκτέλεση της εφαρμογής ως Unikernel. Πιο συγκεκριμένα, όπως φαίνεται στις μετρήσεις που παραθέτουμε, τόσο στο Solo5-hvt όσο και στο QEMU περιλαμβάνεται περίπου ο ίδιος χρόνος για το syscall (αναμενόμενο) που κάνουν ως εφαρμογές χρήστη για την εξυπηρέτηση της εξόδου που προκλήθηκε στον guest. Ο υπόλοιπος πρόσθετος χρόνος αφορά στην περίπτωση του Solo5 την έξοδο/είσοδο στο KVM ενώ στο QEMU Linux Guest προστίθεται το overhead των mode switches. Το συνολικό overhead λοιπόν στη δεύτερη περίπτωση είναι σημαντικά μεγαλύτερο.

Τέλος αξίζει να σημειώσουμε ότι ο χρόνος εκκίνησης των παραπάνω εφαρμογών που εκτελέσαμε ως Unikernels είναι αρκετά μικρός όπως μπορούμε να παρατηρήσουμε με μία σύγκριση του συνολικού χρόνου εκτέλεσης της πραγματικής *δουλειάς* που κάνει η εφαρμογή (δηλαδή της μέτρησης του χρόνου εξυπηρέτησης Ν αιτημάτων) και του συνολικού χρόνου εκτέλεσης του Unikernel που την περιλαμβάνει (από την εκκίνησή του μέχρι την απενεργοποίησή του).

Οι Unikernels φαίνεται να αποτελούν μία καλή εναλλακτική για ελαστικές υπηρεσίες (on demand) που θέλουμε να ξεκινούν και να απενεργοποιούνται γρήγορα όταν πλέον δε χρησιμοποιούνται και άρα όχι μόνο εξοικονομούν πόρους λόγω μικρότερου αποτυπώματος στη μνήμη αλλά εν δυνάμει απαλείφουν το idle time αποδεσμεύοντας υπολογιστικούς πόρους.

6.2 Μελλοντικές επεκτάσεις

Τα αποτελέσματα μας ενθαρρύνουν να μελετήσουμε σε μεγαλύτερο βάθος πιθανές αναπροσαρμογές στην αρχιτεκτονική της εκτέλεσης των εφαρμογών στο cloud. Η περαιτέρω μελέτη

της επιβάρυνσης σε διάσημες εφαρμογές που εκτελούνται σε νεφϋπολογιστικά περιβάλλοντα και η πιο λεπτομερής ανάλυση του χρόνου εκτέλεσης σε user / kernel / hypervisor mode θεωρούμε ότι θα αναδείξει την αναγκαιότητα περαιτέρω βελτιστοποίησης των υπαρχόντων αρχιτεκτονικών. Η μέχρι τώρα μελέτη των τεχνολογιών εικονικοποίησης αποδεικνύει ότι υπάρχει η δυνατότητα απαλειφής των περιττών εξαρτήσεων ενός συμβατικού Λειτουργικού Συστήματος. Η άμεση εξυπηρέτηση των αιτημάτων των guest εφαρμογών, προς τους φυσικούς πόρους του συστήματος, από τον πυρήνα του host, αποφεύγοντας άσκοπα mode switches έχει ήδη αποδειχθεί (με την τεχνολογία vhost) ότι μπορεί να βελτιώσει δραματικά τις επιδόσεις. Η επέκταση της προσέγγισης αυτής σε όλα τα διαφορετικά στάδια εκτέλεσης μίας εφαρμογής, ακόμη και στην διαχείριση των εξόδων του hypervisor απευθείας από τον πυρήνα του host, μηδενίζοντας την χρησιμότητα του *monitor* ή μάλλον εξωθώντας τον στον πυρήνα, θα μπορούσε να σημάνει και την πλήρη απαλοιφή του overhead που προσθέτει το Λειτουργικό Σύστημα.

Οραματιζόμαστε ένα περιβάλλον όπου ο hypervisor είναι ένα λεπτό στρώμα χρονοδρομολόγησης εργασιών στους φυσικούς πόρους του υλικού, λαμβάνοντας υπόψη την ετερογένεια των επεξεργαστικών μονάδων. Μια εφαρμογή θα απαρτίζεται από επιμέρους υπο-εφαρμογές που θα εκτελούνται κατανεμημένα, αποδοτικά, με ασφάλεια, χωρίς περαιτέρω επιβάρυνση από περιττά στρώματα λογισμικού. Ο έλεγχος της πρόσβασης σε λειτουργίες αυξημένων δυνατοτήτων θα γίνεται από τον hypervisor και οι υπο-εφαρμογές θα επικοινωνούν με συμβατικές μεθόδους επικοινωνίας είτε βρίσκονται στον ίδιο φυσικό κόμβο, είτε σε διαφορετικούς.

Βιβλιογραφία

[1] Mell, Peter, and Tim Grance, The NIST definition of cloud computing, *Computer Security Division, Information Technology Laboratory, National Institute of Standards and Technology Gaithersburg*, 2011.

[2] Wikipedia - Εικονικοποίηση, `https://el.wikipedia.org/wiki/`Εικονικοποίηση , Last accessed on 15/10/2018.

[3] Madhavapeddy, Anil & J. Scott, David, Unikernels: The Rise of the Virtual Library Operating System, *Communications of the ACM*, 57. 61-69. 10.1145/2541883.2541895, 2014.

[4] Russel C. Pavlicek, Unikernels: Beyond Containers to the Next Generation of Cloud, *O'Reilly Media Inc.*, 2017.

[5] The VENOM vulnerability, `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-3456`, 2015.

[6] Kantee, A., The Design and Implementation of the Anykernel and Rump Kernels, *2nd edition*, 2016.

[7] Kantee, A. & Cormack J., Rump Kernels: No OS? No Problem!, *USENIX ;login:*, October 2014.

[8] rumprun-packages repository, `https://github.com/rumpkernel/rumprun-packages`, Last accessed on 15/10/2018.

[9] rumprun repository, `https://github.com/rumpkernel/rumprun`, Last accessed on 15/10/2018.

[10] Nabla Containers repository, Rumprun Unikernels with Solo5 backend, `https://github.com/nabla-containers/rumprun`, Last accessed on 15/10/2018.

[11] MirageOS website, `https://mirage.io`, Last accessed on 15/10/2018.

[12] Karl Marx, From each according to his ability, to each according to his needs. *Critique of the Gotha Program*, 1875

[13] MirageOS wiki Technical Background, `https://mirage.io/wiki/technical-background`, Last accessed on 15/10/2018.

[14] Solo5 repository, `https://github.com/Solo5/solo5`, Last accessed on 15/10/2018.

[15] Williams, D., & Koller, R., Unikernel Monitors: Extending Minimalism Outside of the Box, *In HotCloud*, June 2016.

[16] Solo5 Architecture, `https://github.com/Solo5/solo5/blob/master/docs/architecture.md`, Last accessed on 15/10/2018.

[17] Muen Separation Kernel, `https://muen.codelabs.ch`, Last accessed on 15/10/2018.

[18] IncludeOS repository, `https://github.com/hioa-cs/IncludeOS`, Last accessed on 15/10/2018.

[19] Bratterud, Alfred, et al., IncludeOS: A minimal, resource efficient unikernel for cloud services, *Cloud Computing Technology and Science (CloudCom), 2015 IEEE 7th International Conference on. IEEE*, 2015.

[20] Stroustrup, B., The Design and Evolution of C++, *Addison Wesley*, ISBN 0-201-54330-3, March 1994.

[21] LLVM, `http://llvm.org` .

[22] IncludeOS documentation - Features, `https://includeos.readthedocs.io/en/latest/Features.html`, Last accessed on 15/10/2018.

[23] Wikipedia - Web application ARchive, `https://en.wikipedia.org/wiki/WAR_(file_format)`, Last accessed on 15/10/2018.

[24] OSv User Cases, `http://osv.io/user-cases`, Last accessed on 15/10/2018.

[25] Mikelangelo Project, `https://www.mikelangelo-project.eu`, Last accessed on 15/10/2018.

[26] OSv Wiki AArch64, `https://github.com/cloudius-systems/osv/wiki/AArch64`, Last accessed on 15/10/2018.

[27] Williams, D., Solo5: Building a Unikernel Base From Scratch, *Cloud Innovators Forum (CIF16)*, January 2016.

[28] MP30-AR0: AppliedMicro® X-Gene1 SoC, `https://b2b.gigabyte.com/Server-Motherboard/MP30-AR0-rev-11\#ov`, Last accessed on 15/10/2018.

Παράρτημα Α΄

Πηγαίος κώδικας

Α.1 Μετρο-πρόγραμμα επιδόσεων δικτύου

```
1 /*
2     This program is free software: you can redistribute it and/or modify
3     it under the terms of the GNU General Public License as published by
4     the Free Software Foundation, either version 3 of the License, or
5     (at your option) any later version.
6
7     This program is distributed in the hope that it will be useful,
8     but WITHOUT ANY WARRANTY; without even the implied warranty of
9     MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See
      the
10    GNU General Public License for more details.
11
12    You should have received a copy of the GNU General Public License
13    along with this program. If not, see <http://www.gnu.org/licenses/>.
14 */
15
16 #include <stdio.h>
17 #include <stdlib.h>
18 #include <unistd.h>          // close()
19 #include <string.h>          // strcpy, memset(), and memcpy()
20 #include <time.h>
21
22 #include <netdb.h>           // struct addrinfo
23 #include <sys/types.h>       // needed for socket(), uint8_t, uint16_t, uint32_t
24 #include <sys/socket.h>      // needed for socket()
25 #include <netinet/in.h>      // IPPROTO_ICMP, INET_ADDRSTRLEN
26 #include <netinet/ip.h>      // struct ip and IP_MAXPACKET (which is 65535)
```

```
27 #include <netinet/ip_icmp.h>  // struct icmp, ICMP_ECHO
28 #include <arpa/inet.h>       // inet_pton() and inet_ntop()
29 #include <sys/ioctl.h>       // macro ioctl is defined
30 #include <bits/ioctls.h>     // defines values for argument "request" of ioctl.
31 #include <net/if.h>          // struct ifreq
32 #include <linux/if_ether.h>  // ETH_P_IP = 0x0800, ETH_P_IPV6 = 0x86DD
33 #include <linux/if_packet.h> // struct sockaddr_ll (see man 7 packet)
34 #include <net/ethernet.h>
35
36 #include <errno.h>           // errno, perror()
37
38 // Define some constants.
39 #define ETH_HDRLEN 14  // Ethernet header length
40 #define IP4_HDRLEN 20  // IPv4 header length
41 #define ICMP_HDRLEN 8  // ICMP header length for echo request, excludes data
42 #define ARP_HDRLEN 28      // ARP header length
43 #define ARPOP_REQUEST 1    // Taken from <linux/if_arp.h>
44 #define ARPOP_REPLY 2      // Taken from <linux/if_arp.h>
45
46
47 // Function prototypes
48 uint16_t checksum (uint16_t *, int);
49 uint16_t icmp4_checksum (struct icmp, uint8_t *, int);
50 char *allocate_strmem (int);
51 uint8_t *allocate_ustrmem (int);
52 int *allocate_intmem (int);
53
54 // Define a struct for ARP header
55 typedef struct _arp_hdr arp_hdr;
56 struct _arp_hdr {
57   uint16_t htype;
58   uint16_t ptype;
59   uint8_t hlen;
60   uint8_t plen;
61   uint16_t opcode;
62   uint8_t sender_mac[6];
63   uint8_t sender_ip[4];
64   uint8_t target_mac[6];
65   uint8_t target_ip[4];
66 };
67   int
```

```
68 main (int argc, char **argv)
69 {
70   int i, status, datalen, frame_length, sd, bytes, *ip_flags;
71   char *interface, *target, *src_ip, *dst_ip;
72   struct ip iphdr;
73   arp_hdr arphdr, *arphdr_rec;
74   struct icmp icmphdr, icmphdr2;
75   uint8_t *data, *src_mac, *dst_mac, *ether_frame, *ether_frame_rec;
76   struct sockaddr_in *ipv4;
77   struct sockaddr_ll device;
78   struct ifreq ifr;
79   void *tmp;
80   int it;
81   uint16_t icmpseq;
82   datalen = 4;
83   if (argc >= 3) {
84     it = atoi(argv[1]);
85     datalen = atoi(argv[2]);
86   } else if (argc == 2) {
87     it = atoi(argv[1]);
88   } else {
89     //set iterations default to 1000
90     it = 1000;
91   }
92
93   // Allocate memory for various arrays.
94   src_mac = allocate_ustrmem (6);
95   dst_mac = allocate_ustrmem (6);
96   data = allocate_ustrmem (IP_MAXPACKET);
97   ether_frame = allocate_ustrmem (IP_MAXPACKET);
98   ether_frame_rec = allocate_ustrmem (IP_MAXPACKET);
99   interface = allocate_strmem (40);
100  target = allocate_strmem (40);
101  src_ip = allocate_strmem (INET_ADDRSTRLEN);
102  dst_ip = allocate_strmem (INET_ADDRSTRLEN);
103  ip_flags = allocate_intmem (4);
104
105  // Interface to send packet through.
106  strcpy (interface, "tap100");
107
108  // Submit request for a socket descriptor to look up interface.
```

```
109 if ((sd = socket (PF_PACKET, SOCK_RAW, htons (ETH_P_ALL))) < 0) {
110   perror ("socket() failed to get socket descriptor for using ioctl() ");
111   exit (EXIT_FAILURE);
112 }
113
114 // Use ioctl() to look up interface name and get its MAC address.
115 memset (&ifr, 0, sizeof (ifr));
116 snprintf (ifr.ifr_name, sizeof (ifr.ifr_name), "%s", interface);
117 if (ioctl (sd, SIOCGIFHWADDR, &ifr) < 0) {
118   perror ("ioctl() failed to get source MAC address ");
119   return (EXIT_FAILURE);
120 }
121 close (sd);
122
123 // Copy source MAC address.
124 memcpy (src_mac, ifr.ifr_hwaddr.sa_data, 6);
125
126 // Report source MAC address to stdout.
127 printf ("MAC address for interface %s is ", interface);
128 for (i=0; i<5; i++) {
129   printf ("%02x:", src_mac[i]);
130 }
131 printf ("%02x\n", src_mac[5]);
132
133 // Find interface index from interface name and store index in
134 // struct sockaddr_ll device, which will be used as an argument of sendto().
135 memset (&device, 0, sizeof (device));
136 if ((device.sll_ifindex = if_nametoindex (interface)) == 0) {
137   perror ("if_nametoindex() failed to obtain interface index ");
138   exit (EXIT_FAILURE);
139 }
140 printf ("Index for interface %s is %i\n", interface, device.sll_ifindex);
141
142 // Source IPv4 address: you need to fill this out
143 strcpy (src_ip, "10.0.0.1");
144
145 // Destination URL or IPv4 address: you need to fill this out
146 strcpy (dst_ip, "10.0.0.2");
147
148 // Fill out sockaddr_ll.
149 device.sll_family = AF_PACKET;
```

```
150  memcpy (device.sll_addr, src_mac, 6);
151  device. sll_halen  = 6;
152
153  // Before proceeding with the icmp request
154  // let's  first  decide the target  MAC address
155  // by sending an ARP request
156  // Prepare the ethernet frame (ARP request)
157
158  // Source IPv4 address (32 bits)
159  if  ((status  = inet_pton (AF_INET, src_ip, &(arphdr.sender_ip))) != 1) {
160     fprintf  (stderr , "inet_pton()  failed .\nError message: %s", strerror  (status));
161    exit (EXIT_FAILURE);
162  }
163
164  // Destination IPv4 address (32 bits)
165  if  ((status  = inet_pton (AF_INET, dst_ip, &(arphdr.target_ip))) != 1) {
166     fprintf  (stderr , "inet_pton()  failed .\nError message: %s", strerror  (status));
167    exit (EXIT_FAILURE);
168  }
169  // ARP header
170
171  // Hardware type (16 bits): 1 for ethernet
172  arphdr.htype = htons (1);
173
174  // Protocol type (16 bits): 2048 for IP
175  arphdr.ptype = htons (ETH_P_IP);
176
177  // Hardware address length (8 bits): 6 bytes for MAC address
178  arphdr.hlen = 6;
179
180  // Protocol address length (8 bits): 4 bytes for IPv4 address
181  arphdr.plen = 4;
182
183  // OpCode: 1 for ARP request
184  arphdr.opcode = htons (ARPOP_REQUEST);
185
186  // Sender hardware address (48 bits): MAC address
187  memcpy (&arphdr.sender_mac, src_mac, 6 * sizeof (uint8_t));
188
189  // Target hardware address (48 bits): zero,  since  we don't know it yet.
190  memset (&arphdr.target_mac, 0, 6 * sizeof (uint8_t));
```

```
191 // Fill out ethernet frame header.
192
193 // Ethernet frame length = ethernet header (MAC + MAC + ethernet type) +
      ethernet data (ARP header)
194 frame_length = 6 + 6 + 2 + ARP_HDRLEN;
195
196 // Set destination MAC address: broadcast address
197 memset (dst_mac, 0xff, 6 * sizeof (uint8_t));
198
199 // Destination and Source MAC addresses
200 memcpy (ether_frame, dst_mac, 6 * sizeof (uint8_t));
201 memcpy (ether_frame + 6, src_mac, 6 * sizeof (uint8_t));
202
203 // Next is ethernet type code (ETH_P_ARP for ARP).
204 // http://www.iana.org/assignments/ethernet-numbers
205 ether_frame[12] = ETH_P_ARP / 256;
206 ether_frame[13] = ETH_P_ARP % 256;
207
208 // Next is ethernet frame data (ARP header).
209
210 // ARP header
211 memcpy (ether_frame + ETH_HDRLEN, &arphdr, ARP_HDRLEN * sizeof
      (uint8_t));
212
213 // Submit request for a raw socket descriptor.
214 if ((sd = socket (PF_PACKET, SOCK_RAW, htons (ETH_P_ALL))) < 0) {
215   perror ("socket() failed ");
216   exit (EXIT_FAILURE);
217 }
218
219 printf("Sending an ARP Request..\n");
220 // Send ethernet frame to socket.
221 if ((bytes = sendto (sd, ether_frame, frame_length, 0, (struct sockaddr *)
      &device, sizeof (device))) <= 0) {
222   perror ("sendto() ARP failed");
223   exit (EXIT_FAILURE);
224 }
225
226 // Listen for incoming ethernet frame from socket sd.
227 // We expect an ARP ethernet frame of the form:
228 //      MAC (6 bytes) + MAC (6 bytes) + ethernet type (2 bytes)
```

```
229 //      + ethernet data (ARP header) (28 bytes)
230 // Keep at it until we get an ARP reply.
231 printf(" listener: Waiting for ARP reply...\n");
232 arphdr_rec = (arp_hdr *) (ether_frame_rec + 6 + 6 + 2);
233 while (((((ether_frame_rec[12]) << 8) + ether_frame_rec[13]) != ETH_P_ARP)
      || (ntohs (arphdr_rec->opcode) != ARPOP_REPLY)) {
234   if ((status = recv (sd, ether_frame_rec, IP_MAXPACKET, 0)) < 0) {
235     if (errno == EINTR) {
236       memset (ether_frame_rec, 0, IP_MAXPACKET * sizeof (uint8_t));
237       continue; // Something weird happened, but let's try again.
238     } else {
239       perror ("recv() ARP failed:");
240       exit (EXIT_FAILURE);
241     }
242   }
243 }
244 printf ("Sender hardware (MAC) address: ");
245 for (i=0; i<6; i++) {
246   printf ("%02x:", arphdr_rec->sender_mac[i]);
247   dst_mac[i] = arphdr_rec->sender_mac[i];
248   }
249 printf ("\nSender protocol (IPv4) address: %u.%u.%u.%u\n",
250 arphdr_rec->sender_ip[0], arphdr_rec->sender_ip[1], arphdr_rec->sender_ip[2],
      arphdr_rec->sender_ip[3]);
251
252 // Close socket descriptor.
253 close (sd);
254
255
256 printf ("Proceeding to ICMP test\n");
257 printf("Running for: %d iterations and %d bytes of icmp data\n", it, datalen);
258 // Report Target MAC address to stdout.
259 printf ("Target's MAC address is:");
260 for (i=0; i<5; i++) {
261   printf ("%02x:", dst_mac[i]);
262 }
263 printf ("%02x\n", dst_mac[5]);
264
265 // ICMP data
266 for (i=0; i<datalen; i++)
267   data[i] = 'K';
```

```
268
269 // IPv4 header
270
271 // IPv4 header length (4 bits): Number of 32-bit words in header = 5
272 iphdr.ip_hl = IP4_HDRLEN / sizeof (uint32_t);
273
274 // Internet Protocol version (4 bits): IPv4
275 iphdr.ip_v = 4;
276
277 // Type of service (8 bits)
278 iphdr.ip_tos = 0;
279
280 // Total length of datagram (16 bits): IP header + ICMP header + ICMP data
281 iphdr.ip_len = htons (IP4_HDRLEN + ICMP_HDRLEN + datalen);
282
283 // ID sequence number (16 bits): unused, since single datagram
284 iphdr.ip_id = htons (0);
285
286 // Flags, and Fragmentation offset (3, 13 bits): 0 since single datagram
287
288 // Zero (1 bit)
289 ip_flags [0] = 0;
290
291 // Do not fragment flag (1 bit)
292 ip_flags [1] = 0;
293
294 // More fragments following flag (1 bit)
295 ip_flags [2] = 0;
296
297 // Fragmentation offset (13 bits)
298 ip_flags [3] = 0;
299
300 iphdr.ip_off = htons ((ip_flags [0] << 15)
301     + (ip_flags [1] << 14)
302     + (ip_flags [2] << 13)
303     + ip_flags [3]);
304
305 // Time-to-Live (8 bits): default to maximum value
306 iphdr.ip_ttl = 255;
307
308 // Transport layer protocol (8 bits): 1 for ICMP
```

```
309  iphdr.ip_p = IPPROTO_ICMP;
310
311  // Source IPv4 address (32 bits)
312  if ((status = inet_pton (AF_INET, src_ip, &(iphdr.ip_src))) != 1) {
313    fprintf (stderr, "inet_pton() failed.\nError message: %s", strerror (status));
314    exit (EXIT_FAILURE);
315  }
316
317  // Destination IPv4 address (32 bits)
318  if ((status = inet_pton (AF_INET, dst_ip, &(iphdr.ip_dst))) != 1) {
319    fprintf (stderr, "inet_pton() failed.\nError message: %s", strerror (status));
320    exit (EXIT_FAILURE);
321  }
322
323  // IPv4 header checksum (16 bits): set to 0 when calculating checksum
324  iphdr.ip_sum = 0;
325  iphdr.ip_sum = checksum ((uint16_t *) &iphdr, IP4_HDRLEN);
326
327  // ICMP header
328
329  // Message Type (8 bits): echo request
330  icmphdr.icmp_type = ICMP_ECHO;
331
332  // Message Code (8 bits): echo request
333  icmphdr.icmp_code = 0;
334
335  // Identifier (16 bits): usually pid of sending process - pick a number
336  icmphdr.icmp_id = htons (1000);
337
338  // Sequence Number (16 bits): starts at 0
339  icmphdr.icmp_seq = htons (0);
340
341  // ICMP header checksum (16 bits): set to 0 when calculating checksum
342  icmphdr.icmp_cksum = icmp4_checksum (icmphdr, data, datalen);
343
344  // Fill out ethernet frame header.
345
346  // Ethernet frame length = ethernet header (MAC + MAC + ethernet type) +
       ethernet data (IP header + ICMP header + ICMP data)
347  frame_length = 6 + 6 + 2 + IP4_HDRLEN + ICMP_HDRLEN + datalen;
348
```

```
349  // Destination and Source MAC addresses
350  memcpy (ether_frame, dst_mac, 6);
351  memcpy (ether_frame + 6, src_mac, 6);
352
353  // Next is ethernet type code (ETH_P_IP for IPv4).
354  // http://www.iana.org/assignments/ethernet-numbers
355  ether_frame[12] = ETH_P_IP / 256;
356  ether_frame[13] = ETH_P_IP % 256;
357
358  // Next is ethernet frame data (IPv4 header + ICMP header + ICMP data).
359
360  // IPv4 header
361  memcpy (ether_frame + ETH_HDRLEN, &iphdr, IP4_HDRLEN);
362
363  // ICMP header
364  memcpy (ether_frame + ETH_HDRLEN + IP4_HDRLEN, &icmphdr,
       ICMP_HDRLEN);
365
366  // ICMP data
367  memcpy (ether_frame + ETH_HDRLEN + IP4_HDRLEN + ICMP_HDRLEN,
       data, datalen);
368
369  // Submit request for a raw socket descriptor.
370  if ((sd = socket (PF_PACKET, SOCK_RAW, htons (ETH_P_IP))) < 0) {
371    perror ("socket() failed ");
372    exit (EXIT_FAILURE);
373  }
374
375  // Bind socket to interface index.
376  if (bind (sd, (struct sockaddr *) &device, sizeof (device) ) < 0) {
377    perror ("bind() failed to bind to interface ");
378    exit (EXIT_FAILURE);
379  }
380
381
382  struct timespec tv1, tv2;
383  ssize_t numbytes;
384  int not_us, oooerror;
385  double average_us = 0;
386  long long roundtrip;
387  long long maxtrip = 0;
```

```
388  long long mintrip = 10000000000000;
389  double iterations;
390  iterations = it;
391  oooerror = 0;
392  not_us = 0;
393 send:
394  // Send ethernet frame to socket.
395  if (it < 0)
396    goto end;
397  it = it - 1;
398  clock_gettime(CLOCK_MONOTONIC, &tv1);
399  if ((bytes = sendto (sd, ether_frame, frame_length, 0, (struct sockaddr *)
       &device, sizeof (device))) <= 0) {
400    perror ("sendto() failed");
401    exit (EXIT_FAILURE);
402  }
403
404 receive:
405  numbytes = recv (sd, ether_frame_rec, frame_length, 0);
406  clock_gettime(CLOCK_MONOTONIC, &tv2);
407  if (ether_frame_rec[0] == src_mac[0] &&
408      src_mac[1] == ether_frame_rec[1] &&
409      src_mac[2] == ether_frame_rec[2] &&
410      src_mac[3] == ether_frame_rec[3] &&
411      src_mac[4] == ether_frame_rec[4] &&
412      src_mac[5] == ether_frame_rec[5]) {
413    //The reply is definitely addressed to us
414    //but we still have to match the sequence number
415    //extract the icmp header from the ethernet frame
416    memcpy (&icmphdr2, ether_frame_rec + ETH_HDRLEN + IP4_HDRLEN,
       ICMP_HDRLEN);
417    icmpseq = ntohs(icmphdr2.icmp_seq);
418    if (icmphdr2.icmp_seq != icmphdr.icmp_seq) {
419      printf("Wrong sequence number!!\n");
420      fprintf(stderr, "Ending now");
421      exit (EXIT_FAILURE);
422    }
423
424    //calculate the total request-reply time
425    roundtrip = ((tv2.tv_sec - tv1.tv_sec) * 1000000000) + (tv2.tv_nsec -
       tv1.tv_nsec);
```

```
426     average_us = average_us + roundtrip;
427
428     //also keep track of min/max roundtrip values
429     if (roundtrip > maxtrip)
430       maxtrip = roundtrip;
431     if (roundtrip < mintrip)
432       mintrip = roundtrip;
433     icmpseq = ntohs(icmphdr.icmp_seq);
434     icmphdr.icmp_seq = htons(icmpseq + 1);
435
436     //recalculate icmp checksum.
437     icmphdr.icmp_cksum = icmp4_checksum (icmphdr, data, datalen);
438
439     //Fill in the new ICMP header + the target MAC address.
440     memcpy (ether_frame + ETH_HDRLEN + IP4_HDRLEN, &icmphdr,
        ICMP_HDRLEN);
441     memcpy (ether_frame, ether_frame_rec + 6, 6);
442     goto send;
443   } else {
444     //notify that the dst MAC was not us and go to listener
445
446     printf("with dst MAC: %x:%x:%x:%x:%x:%x ",
447         ether_frame_rec [0],
448         ether_frame_rec [1],
449         ether_frame_rec [2],
450         ether_frame_rec [3],
451         ether_frame_rec [4],
452         ether_frame_rec [5]) ;
453
454     printf("and src MAC: %x:%x:%x:%x:%x:%x\n",
455         ether_frame_rec [6],
456         ether_frame_rec [7],
457         ether_frame_rec [8],
458         ether_frame_rec [9],
459         ether_frame_rec [10],
460         ether_frame_rec [11]) ;
461     not_us++;
462     goto receive ;
463
464   }
465
```

```
466end:
467   printf ( "Total time for %d iterations of %d datalen: %f ns\n",
        (int) iterations , datalen, average_us);
468   average_us = average_us / iterations ;
469   printf ( "AVERAGE / Max / Min = %f ns / %d ns / %d ns \n", average_us,
        maxtrip, mintrip);
470   printf ( "AVERAGE packets per second = %f /sec\n", (1 / average_us) *
        1000000000);
471   if (not_us > 0) {
472     fprintf (stderr, "Received %d frames not addressed to us\n", not_us);
473   }
474   // Close socket descriptor.
475   close (sd);
476
477   // Free allocated memory.
478   free (src_mac);
479   free (dst_mac);
480   free (data);
481   free (ether_frame);
482   free (ether_frame_rec);
483   free ( interface );
484   free (target);
485   free ( src_ip );
486   free (dst_ip);
487   free ( ip_flags );
488
489   return (EXIT_SUCCESS);
490}
491
492
493// Build IPv4 ICMP pseudo-header and call checksum function.
494   uint16_t
495icmp4_checksum (struct icmp icmphdr, uint8_t *payload, int payloadlen)
496{
497   char buf[IP_MAXPACKET];
498   char *ptr;
499   int chksumlen = 0;
500   int i;
501
502   ptr = &buf[0];  // ptr points to beginning of buffer buf
503
```

```
504  // Copy Message Type to buf (8 bits)
505  memcpy (ptr, &icmphdr.icmp_type, sizeof (icmphdr.icmp_type));
506  ptr += sizeof (icmphdr.icmp_type);
507  chksumlen += sizeof (icmphdr.icmp_type);
508
509  // Copy Message Code to buf (8 bits)
510  memcpy (ptr, &icmphdr.icmp_code, sizeof (icmphdr.icmp_code));
511  ptr += sizeof (icmphdr.icmp_code);
512  chksumlen += sizeof (icmphdr.icmp_code);
513
514  // Copy ICMP checksum to buf (16 bits)
515  // Zero, since we don't know it yet
516  *ptr = 0; ptr++;
517  *ptr = 0; ptr++;
518  chksumlen += 2;
519
520  // Copy Identifier to buf (16 bits)
521  memcpy (ptr, &icmphdr.icmp_id, sizeof (icmphdr.icmp_id));
522  ptr += sizeof (icmphdr.icmp_id);
523  chksumlen += sizeof (icmphdr.icmp_id);
524
525  // Copy Sequence Number to buf (16 bits)
526  memcpy (ptr, &icmphdr.icmp_seq, sizeof (icmphdr.icmp_seq));
527  ptr += sizeof (icmphdr.icmp_seq);
528  chksumlen += sizeof (icmphdr.icmp_seq);
529
530  // Copy payload to buf
531  memcpy (ptr, payload, payloadlen);
532  ptr += payloadlen;
533  chksumlen += payloadlen;
534
535  // Pad to the next 16-bit boundary
536  for (i=0; i<payloadlen%2; i++, ptr++) {
537    *ptr = 0;
538    ptr++;
539    chksumlen++;
540  }
541
542  return checksum ((uint16_t *) buf, chksumlen);
543}
544
```

```
545// Computing the internet checksum (RFC 1071).
546// Note that the internet checksum does not preclude collisions.
547 uint16_t
548checksum (uint16_t *addr, int len)
549{
550  int count = len;
551  register uint32_t sum = 0;
552  uint16_t answer = 0;
553
554  // Sum up 2-byte values until none or only one byte left.
555  while (count > 1) {
556    sum += *(addr++);
557    count -= 2;
558  }
559
560  // Add left-over byte, if any.
561  if (count > 0) {
562    sum += *(uint8_t *) addr;
563  }
564
565  // Fold 32-bit sum into 16 bits; we lose information by doing this,
566  // increasing the chances of a collision.
567  // sum = (lower 16 bits) + (upper 16 bits shifted right 16 bits)
568  while (sum >> 16) {
569    sum = (sum & 0xffff) + (sum >> 16);
570  }
571
572  // Checksum is one's compliment of sum.
573  answer = ~sum;
574
575  return (answer);
576}
577
578// Allocate memory for an array of chars.
579  char *
580allocate_strmem (int len)
581{
582  void *tmp;
583
584  if (len <= 0) {
585    fprintf (stderr, "ERROR: Cannot allocate memory because len = %i in
```

```
        allocate_strmem().\n", len);
586     exit (EXIT_FAILURE);
587   }
588
589   tmp = (char *) malloc (len * sizeof (char));
590   if (tmp != NULL) {
591     memset (tmp, 0, len * sizeof (char));
592     return (tmp);
593   } else {
594     fprintf (stderr, "ERROR: Cannot allocate memory for array
        allocate_strmem().\n");
595     exit (EXIT_FAILURE);
596   }
597}
598
599// Allocate memory for an array of unsigned chars.
600   uint8_t *
601allocate_ustrmem (int len)
602{
603   void *tmp;
604
605   if (len <= 0) {
606     fprintf (stderr, "ERROR: Cannot allocate memory because len = %i in
        allocate_ustrmem().\n", len);
607     exit (EXIT_FAILURE);
608   }
609
610   tmp = (uint8_t *) malloc (len * sizeof (uint8_t));
611   if (tmp != NULL) {
612     memset (tmp, 0, len * sizeof (uint8_t));
613     return (tmp);
614   } else {
615     fprintf (stderr, "ERROR: Cannot allocate memory for array
        allocate_ustrmem().\n");
616     exit (EXIT_FAILURE);
617   }
618}
619
620// Allocate memory for an array of ints.
621   int *
622allocate_intmem (int len)
```

```
623{
624   void *tmp;
625
626   if (len <= 0) {
627      fprintf (stderr, "ERROR: Cannot allocate memory because len = %i in
      allocate_intmem().\n", len);
628      exit (EXIT_FAILURE);
629   }
630
631   tmp = (int *) malloc (len * sizeof (int));
632   if (tmp != NULL) {
633      memset (tmp, 0, len * sizeof (int));
634      return (tmp);
635   } else {
636      fprintf (stderr, "ERROR: Cannot allocate memory for array
      allocate_intmem().\n");
637      exit (EXIT_FAILURE);
638   }
639}
```

icmp–test.c

A.2 Μετρο-πρόγραμμα επιδόσεων I/O του Virtual Block Device - Linux Guest

```
1/**
2 * Test to measure read time from a block device in Linux
3 * Compile and run with:
4 * gcc test_readblk.c -o test_readblk
5 */
6#include <sys/types.h>
7#include <sys/stat.h>
8#include <fcntl.h>
9#include <unistd.h>
10#include <stdio.h>
11#include <linux/fs.h>
12#include <sys/ioctl.h>
13#include <errno.h>
14#include <time.h>
15#include <stdlib.h>
16
17int main() {
```

```
18
19  long long size, total_time;
20  struct timespec tv1, tv2;
21
22  char *rbuf;
23  posix_memalign(&rbuf, 512, 512);
24  int j = 0;
25  ssize_t bytes_read = 0;
26
27  int fd = open("/dev/vda", O_RDWR|O_DIRECT);
28  off_t offset = 0;
29  clock_gettime(CLOCK_MONOTONIC, &tv1);
30  for (int i = 0; i < 100000 ; i++) {
31
32    if ( pread(fd, rbuf, 512, offset ) == 512)
33      j++;
34    offset += 512;
35  }
36  clock_gettime(CLOCK_MONOTONIC, &tv2);
37  total_time = ((tv2.tv_sec - tv1.tv_sec) * 1000000000) + (tv2.tv_nsec -
      tv1.tv_nsec);
38  fprintf(stderr, "pwrite = %d\n", pwrite(fd, rbuf, 512, offset));
39  printf("Read %d blocks of 512 bytes\n",j);
40  printf("aka %lld total KBs\n", j * 512 / (1024));
41  printf("Total time %lld ns\n", total_time);
42
43  float read_speed;
44
45  read_speed = (j * 512.0 * 1000000000)/((double)total_time * 1024);
46  printf("Read Speed %.02f KB/s\n", read_speed);
47  close(fd);
48  return 0;
49}
```

test_readblk.c

A.3 Μετρο-πρόγραμμα επιδόσεων I/O του Virtual Block Device - Solo5 Guest

```
1 #include "solo5.h"
2 #include "../../kernel/lib.c"
3 #define NOPRINTF
```

```
4 #include "../../kernel/ee_printf.c"
5
6 static void puts(const char *s)
7 {
8   solo5_console_write(s, strlen(s));
9 }
10
11 int solo5_app_main(const struct solo5_start_info *si __attribute__((unused)))
12 {
13   puts("\n**** Solo5 standalone block_read benchmark ****\n\n");
14
15   struct solo5_block_info bi;
16   solo5_block_info(&bi);
17
18
19   uint8_t rbuf_all[bi.block_size + (PAGE_SIZE - 1)];
20
21   unsigned long mask = (PAGE_SIZE - 1);
22   unsigned long addr = (unsigned long) rbuf_all;
23   uint8_t *rbuf = (uint8_t*) ((addr + mask) & ~(mask));
24
25   solo5_time_t t1=0, t2=0;
26
27   solo5_off_t offset = 0;
28   t1 = solo5_clock_monotonic();
29   for (int i = 0; i < 3 ; i++) {
30     if (solo5_block_read(offset, rbuf, bi.block_size) != SOLO5_R_OK)
31       return -1;
32     offset += bi.block_size;
33   }
34   t2 = solo5_clock_monotonic();
35
36   t1 = t2 - t1;
37
38   char dt [(sizeof t1) + 2];
39   memset(dt, 0, sizeof dt);
40   puts("total Solo5 read ");
41   snprintf(dt, sizeof dt, "%lu", t1);
42   puts(dt);
43   puts("\n");
44
```

```
45  double dt_sec = t1 / 1000000000.0;
46  double rate;
47  rate = 100000 / (2.0 * dt_sec);
48  char rate_st [(sizeof rate) + 2];
49  snprintf(rate_st, sizeof rate_st, "%Lu", (int)rate);
50  puts("Rate in KB / s = ");
51  puts(rate_st);
52  puts("\n");
53
54  char block_size [(sizeof bi.block_size) + 2];
55  snprintf(block_size, sizeof block_size, "%lu", bi.block_size);
56  puts(" 100000 read calls happened, of ");
57  puts(block_size);
58  puts(" bytes each\n");
59  if (solo5_block_write(offset, rbuf, bi.block_size) != SOLO5_R_OK)
60    return false;
61
62  puts("SUCCESS\n");
63  return SOLO5_EXIT_SUCCESS;
64}
```

test_blk_bench.c

Παράρτημα Β′

Λοιπά

B.1 Αλλαγές στο QEMU και το Solo5

```
diff --git a/block/file-posix.c b/block/file-posix.c
index fe83cbf0eb..22c577e2a1 100644
--- a/block/file-posix.c
+++ b/block/file-posix.c
@@ -1217,11 +1217,23 @@ static ssize_t
     handle_aiocb_rw_vector(RawPosixAIOData *aiocb)
  * Returns the number of bytes handles or -errno in case of an error. Short
  * reads are only returned if the end of the file is reached.
  */
+static long long total_time = 0;
+static int it = 1;
+static char *prev_buf = NULL;
 static ssize_t handle_aiocb_rw_linear(RawPosixAIOData *aiocb, char *buf)
 {
     ssize_t offset = 0;
     ssize_t len;
-
+    if (buf != prev_buf) {
+ //fprintf(stderr, "current buf = %p, prev buf = %p \n", buf, prev_buf);
+ it=1;
+ total_time = 0;
+ prev_buf = buf;
+    } else {
+ it++;
+    }
+    // fprintf(stderr, "**INSIDE handle_linear_rw**\n");
+    struct timespec tv1, tv2;
```

```
     while (offset < aiocb->aio_nbytes) {
         if (aiocb->aio_type & QEMU_AIO_WRITE) {
             len = pwrite(aiocb->aio_fildes,
@@ -1229,10 +1241,17 @@ static ssize_t
    handle_aiocb_rw_linear(RawPosixAIOData *aiocb, char *buf)
                            aiocb->aio_nbytes - offset,
                            aiocb->aio_offset + offset);
         } else {
+    //       fprintf(stderr, "**calling pread**\n");
+             clock_gettime(CLOCK_MONOTONIC, &tv1);
              len = pread(aiocb->aio_fildes,
                          buf + offset,
                          aiocb->aio_nbytes - offset,
                          aiocb->aio_offset + offset);
+             clock_gettime(CLOCK_MONOTONIC, &tv2);
+    //       fprintf(stderr, "**pread len=%ld\n",len);
+             total_time += ((tv2.tv_sec - tv1.tv_sec) * 1000000000) +
    (tv2.tv_nsec - tv1.tv_nsec);
+         // fprintf(stderr, "**it %d time=%ld\n",it, ((tv2.tv_sec - tv1.tv_sec) *
    1000000000) + (tv2.tv_nsec - tv1.tv_nsec));
+         // fprintf(stderr, "**total time=%lld\n", total_time);
          }
          if (len == -1 && errno == EINTR) {
              continue;
@@ -1856,6 +1875,7 @@ static int coroutine_fn raw_co_pwritev(BlockDriverState
    *bs, uint64_t offset,
                                           int flags)
 {
     assert(flags == 0);
+    fprintf(stderr, "pread time for %d iterations = %lld ns\n", it, total_time);
     return raw_co_prw(bs, offset, bytes, qiov, QEMU_AIO_WRITE);
 }

@@ -1913,7 +1933,6 @@ static void raw_aio_attach_aio_context(BlockDriverState
    *bs,
 static void raw_close(BlockDriverState *bs)
 {
     BDRVRawState *s = bs->opaque;
-
     if (s->fd >= 0) {
         qemu_close(s->fd);
```

```
            s->fd = -1;
```

qemu.diff

```
diff --git a/ukvm/ukvm_module_blk.c b/ukvm/ukvm_module_blk.c
index 4c5f81e..c1c9ada 100644
--- a/ukvm/ukvm_module_blk.c
+++ b/ukvm/ukvm_module_blk.c
@@ -30,7 +30,7 @@
 #include <stdio.h>
 #include <string.h>
 #include <unistd.h>
-
+#include <time.h>
 #include "ukvm.h"

 static struct ukvm_blkinfo blkinfo;
@@ -90,9 +90,12 @@ static void hypercall_blkread(struct ukvm_hv *hv,
     ukvm_gpa_t gpa)
         rd->ret = -1;
         return;
     }
-
+    struct timespec tv1, tv2;
+    clock_gettime(CLOCK_MONOTONIC, &tv1);
     ret = pread(diskfd, UKVM_CHECKED_GPA_P(hv, rd->data, rd->len),
     rd->len,
             pos);
+    clock_gettime(CLOCK_MONOTONIC, &tv2);
+    printf("Syscall time %ld ns\n", ((tv2.tv_sec - tv1.tv_sec) * 1000000000) +
     (tv2.tv_nsec - tv1.tv_nsec));
     assert(ret == rd->len);
     rd->ret = 0;
 }
@@ -112,7 +115,8 @@ static int setup(struct ukvm_hv *hv)
         return -1;

     /* set up virtual disk */
-    diskfd = open(diskfile, O_RDWR);
+    diskfd = open(diskfile, O_RDWR|O_DIRECT);
+    ///diskfd = open(diskfile, O_RDWR);
     if (diskfd == -1)
```

```
            err(1, "Could not open disk: %s", diskfile);
```

solo5.diff

B.2 Εκκίνηση QEMU και Solo5

```
#QEMU boot x86_64 Guest with virtual block device and vhost-net
qemu-system-x86_64 -enable-kvm -m 2048 -kernel bzImage -append
    console=ttyS0 -nographic -serial stdio -nodefaults -initrd rootfs.cpio.gz
    -drive if=virtio,file=/dev/loop0,format=raw,cache=none -net
    nic,model=virtio,macaddr=52:54:00:97:1b:24,netdev=nic-0 -netdev
    tap,id=nic-0,ifname=tap100,vhost=on

#QEMU boot aarch64 Guest with virtual block device and vhost-net
qemu-system-aarch64 -M virt -cpu host -enable-kvm -m 2048 -kernel
    /root/Image-4.17 -nographic -serial stdio -nodefaults -initrd
    /root/rootfs.cpio.gz -net
    nic,model=virtio,macaddr=00:16:3e:00:01:01,netdev=nic-0 -netdev
    tap,id=nic-0,ifname=tap100,vhost=on -drive
    if=virtio,file=/dev/loop0,format=raw,cache=none

#Solo5 boot Unikernel and attach virtual block device
./ukvm-bin --disk=/dev/loop0 test_blk_bench.ukvm

#Solo5 boot Unikernel and attach tap interface
./ukvm-bin --tap=/dev/tap100 test_ping_serve.ukvm
```

boot_commands

Γλωσσάριο

Αγγλικός Όρος	Ελληνικός όρος
access control	έλεγχος πρόσβασης
application layer	στρώμα εφαρμογής (στη στοίβα του λογισμικού)
architecture specific	συγκεκριμένος προς την αρχιτεκτονική
backend	σύστημα υποστήριξης
bootstrap	διαδικασία εκκίνησης
Cloud Computing	Υπολογιστικό Νέφος
compile	μεταγλώττιση
components	συστατικά στοιχεία
cpu time	χρόνος στον επεξεργαστή
Datacenter	Κέντρο Δεδομένων
development stages	στάδια ανάπτυξης
devices	συσκευές
emulation	εξομοίωση
host machine	μηχάνημα οικοδεσπότης
hypervisor	πρόγραμμα υπερεπόπτης
Infrastructure as a Service (IaaS)	Υποδομή ως Υπηρεσία
interaction	αλληλεπίδραση
Internet of Things (IoT)	Διαδίκτυο των Πραγμάτων)
isolation	απομόνωση
guest system	φιλοξενούμενο σύστημα
kernel space	χώρος πυρήνα
low-power	χαμηλή ισχύς
middleware	ενδιάμεσο στρώμα
modular	δομοστοιχειωτός
monitor	πρόγραμμα επόπτης
node	κόμβος
Platform as a Service (PaaS)	Πλατφόρμα ως Υπηρεσία
platform virtualization	εικονικοποίηση πλατφόρμας
porting	μεταφορά
reduced	μειωμένος
registers	καταχωρητές

runtime	χρόνος εκτέλεσης
scale	κλιμάκωση
service	υπηρεσία
single address space	ενιαίος χώρος διευθύνσεων
Software as a Service (SaaS)	Λογισμικό ως Υπηρεσία
storage	αποθήκευση
system calls (syscalls)	κλήσεις συστήματος
tailored	προσαρμοσμένη
User Interface	Διεπαφή Χρήστη
user space	χώρος χρήστη
virtual machine	εικονική μηχανή
workload	φόρτος εργασίας