Εθνικο Μετσοβιο Πολυτεχνειο
Σχολη Ηλεκτρολογων Μηχανικων και Μηχανικων Υπολογιστων

Τομεας Τεχνολογιας Πληροφορικης και Υπολογιστων

Αποδοτικός διαμοιρασμός αρχείων μεταξύ host και unikernel

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Φώτιος Ζαφείρης Μ. Ξενάκης

Επιβλέπων: Νεκτάριος Κοζύρης
Καθηγητής Ε.Μ.Π.

Αθήνα, Οκτώβριος 2020

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

Αποδοτικός διαμοιρασμός αρχείων μεταξύ host και unikernel

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Φώτιος Ζαφείρης Μ. Ξενάκης

Επιβλέπων: Νεκτάριος Κοζύρης
Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 30η Οκτωβρίου 2020.

.....	.....	.....
Νεκτάριος Κοζύρης	Γεώργιος Γκούμας	Διονύσιος Πνευματικάτος
Καθηγητής Ε.Μ.Π.	Επίκουρος καθηγητής Ε.Μ.Π.	Καθηγητής Ε.Μ.Π.

Αθήνα, Οκτώβριος 2020

.....

Φώτιος Ζαφείρης Μ. Ξενάκης
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Περίληψη

Το cloud computing είναι η κυρίαρχη προσέγγιση προς την υπολογιστική υποδομή, βασιζόμενο στην τεχνολογία της εικονικοποίησης (virtualization). Καθώς το cloud επεκτείνεται, καθίσταται επιτακτική η αποδοτική χρήση των υπολογιστικών του πόρων από το λογισμικό. Προς αυτό τον σκοπό, μια λύση είναι τα *unikernels*, πυρήνες λειτουργικών συστημάτων εξειδικευμένοι για να τρέχουν μία μόνο εφαρμογή, εξοικονομώντας πόρους σε σχέση με έναν γενικού σκοπού πυρήνα. Η αποδοτική πρόσβαση των virtualized guests στους πόρους του host συστήματος είναι μια μεγάλη πρόκληση για την εικονικοποίηση. Σε αυτόν τον τομέα, σημαντική συνεισφορά αποτελεί το *virtio*, μια προδιαγραφή paravirtualized συσκευών για την αποδοτική χρήση των πόρων του host. Για το διαμοιρασμό αρχείων ανάμεσα σε host και guest έχει προταθεί το *virtio-fs*, μια virtio συσκευή που προσφέρει στον guest πρόσβαση σε έναν κατάλογο του συστήματος αρχείων του host, παρέχοντας υψηλές επιδόσεις και σημασιολογία τοπικού συστήματος αρχείων.

Αντικείμενο της παρούσας διπλωματικής εργασίας είναι η υλοποίηση και αξιολόγηση της χρήσης του virtio-fs σε ένα unikernel, συγκεκριμένα στο OSv. Δείχνουμε ότι ο συνδυασμός αυτών των δύο έχει σημαντικά πλεονεκτήματα, τόσο από άποψη επιδόσεων, όπου επιτυγχάνονται αποτελέσματα συγκρίσιμα με τα τοπικά συστήματα αρχείων, όσο και από διαχειριστική άποψη στο πλαίσιο του cloud. Επίσης, τα παραπάνω γίνονται σε πλήρη ενσωμάτωση με το έργο ανοιχτού λογισμικού του unikernel στο οποίο βασιστήκαμε. Έτσι, το προϊόν της εργασίας αποκτά πρακτική αξία, καθώς αποτελεί χρήσιμη συνεισφορά στο έργο, επιτυγχάνοντας έτσι έναν κεντρικό, μη τεχνικό, στόχο της. Ταυτόχρονα, εξερευνούμε τον τρόπο λειτουργίας των έργων ανοιχτού λογισμικού και των κοινοτήτων που σχηματίζονται γύρω τους, καθώς γινόμαστε ενεργά μέλη μίας από αυτές.

Λέξεις κλειδιά

εικονικοποίηση, νέφος, σύστημα αρχείων, unikernel, virtio, OSv, virtio-fs, QEMU

Abstract

Cloud computing is the dominant approach to compute infrastructure, established on the technology of virtualization. As the cloud expands, efficient utilization of its compute resources by software becomes imperative. One solution towards that are *unikernels*, operating system kernels specialized to run a single application, sparing resources compared to a general-purpose kernel. Efficient access from virtualized guests to the underlying host's resources is a substantial challenge in virtualization. In this aspect, *virtio* has been a significant contribution, as a specification of paravirtual devices enabling efficient usage of the host's resources. For host-guest file sharing, *virtio-fs* has been proposed, as a virtio device offering guest access to a file system directory on the host, providing high performance and local file system semantics.

This thesis is concerned with the implementation and evaluation of virtio-fs in the context of the OSvunikernel. We demonstrate that combining the two offers great benefits, both with regard to performance achieved, which is comparable to local file systems, and the operational aspect in a cloud context. Moreover, the above are carried out fully within the open-source project behind the unikernel we based our work on. This way, the resulting product gains practical value, being a useful contribution to the project, thus achieving a pivotal, non-technical goal. Furthermore, we explore how open-source software projects and the communities around them work, as we become active members of one.

Keywords

virtualization, cloud, file system, unikernel, virtio, OSv, virtio-fs, QEMU

Ευχαριστίες

Για την παρούσα εργασία, που σηματοδοτεί την ολοκλήρωση μίας πορείας ετών, θα ήθελα να ευχαριστήσω τα μέλη του εργαστηρίου υπολογιστικών συστημάτων, υπό την αιγίδα του οποίου πραγματοποιήθηκε. Πιο πολύ όμως θα ήθελα να τους ευχαριστήσω, όπως και άλλα μέλη της σχολής ΗΜΜΥ, για τη διδασκαλία, το γνήσιο ενδιαφέρον και την καλλιέργεια της κουλτούρας του μηχανικού που μου προσέφεραν.

Επίσης, οφείλω ένα μεγάλο ευχαριστώ στους ανθρώπους των κοινοτήτων του OSv και του virtio-fs για την υποστήριξη, τις συμβουλές και το χρόνο τους, αλλά κυρίως για την ανοικτότητα, το ήθος και το έργο τους, που ενέπνευσαν αυτή τη συνεισφορά.

Τέλος, αυτοί για τους οποίους είμαι περισσότερο ευγνώμων είναι η οικογένεια και οι φίλοι μου, που πάντα βρίσκονται στο πλευρό μου, με ανέχονται και με στηρίζουν και χωρίς τους οποίους τίποτα δεν θα μπορούσε να επιτευχθεί.

Περιεχόμενα

Κατάλογος Σχημάτων

Κατάλογος Πινάκων

Κεφάλαιο 1

Εισαγωγή

Κεφάλαιο 2

Υπόβαθρο

Κεφάλαιο 3

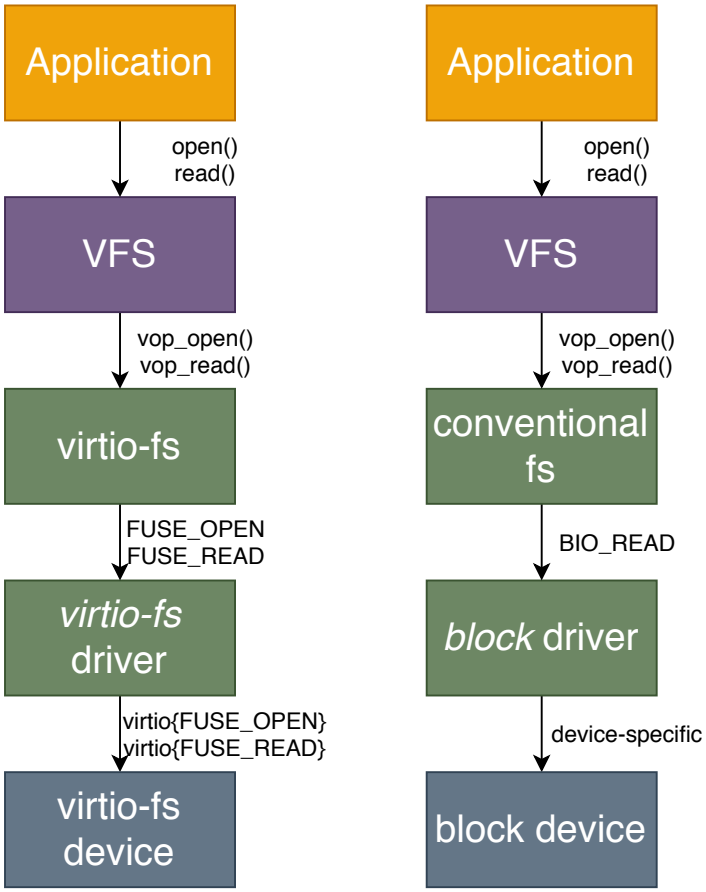
Υλοποίηση

Μία βασική υλοποίηση-σκελετός του virtio-fs προϋπήρχε στο OSv. Αυτή υποστήριζε απλή (μέσω της κλήσης FUSE_READ) ανάγνωση αρχείων (και όχι καταλόγων). Σε συνεννόηση με τα υπόλοιπα μέλη της κοινότητας του OSv, θέσαμε στόχο να αναπτύξουμε περαιτέρω τη λειτουργικότητα αυτής της υλοποίησης. Το κομβικό σημείο αυτής της προσπάθειας ήταν η υποστήριξη ανάγνωσης μέσω του virtio-fs DAX window, κάτι που είχε σχεδιαστεί και υλοποιηθεί (σε 'πειραματικό' ακόμα στάδιο) από τους ανθρώπους του virtio-fs με σκοπό την δραματική βελτίωση των επιδόσεων του συστήματος αρχείων. Εκτός αυτού, είχαμε την ευκαιρία να προσθέσουμε υποστήριξη για ανάγνωση καταλόγων, καθώς και εκκίνηση (boot) του OSv με το virtio-fs ως ριζικό (root) σύστημα αρχείων. Στην πορεία απαιτήθηκαν δομικές αλλαγές, διορθώσεις και βελτιώσεις στην προϋπάρχουσα υλοποίηση.

Σε αυτό το κεφάλαιο αναλύουμε την υλοποίηση των πιο ουσιαστικών από τις παραπάνω συμβολές: της ανάγνωσης αρχείων μέσω DAX window και του boot από virtio-fs.

3.1 DAX window στο virtio-fs

Το OSv ακολουθεί μία εσωτερική δομή συμβατή με αυτή των περισσότερων πυρήνων γενικού σκοπού, ορίζοντας χωριστά τις έννοιες του οδηγού (driver) και του συστήματος αρχείων. Οι οδηγοί (στο drivers/) είναι υπεύθυνοι για την παροχή μίας διεπαφής (interface) κοινής για κάθε οικογένεια συσκευών (αποθήκευσης block, δικτύου κλπ), η οποία επιτρέπει τη χρήση της εκάστοτε συσκευής από τα υπόλοιπα υποσυστήματα του πυρήνα. Τα συστήματα αρχείων (στο fs/) αποτελούν επίσης υλοποιήσεις μίας κοινής διεπαφής η οποία ορίζεται από το εικονικό σύστημα αρχείων (virtual file system) του OSv. Αυτή η διεπαφή αποτελείται από λειτουργίες όπως το άνοιγμα, το κλείσιμο, η ανάγνωση από αρχείο, και άλλες. Τα συστήματα αρχείων συνήθως είναι υλοποιημένα πάνω από μία συσκευή block και χρησιμοποιούν τη διεπαφή που αυτή παρέχει για να 'μεταφράσουν' ανάμεσα στις υψηλού επιπέδου λειτουργίες του συστήματος αρχείων και τις χαμηλού επιπέδου λειτουργίες της συσκευής απο-



Σχήμα 3.1: Συστατικά και εξαρτήσεις στον guest: virtio-fs σε σύγκριση με συμβατικά συστήματα τοπικά αρχείων.

θήκευσης. Υπάρχουν βεβαίως εξαιρέσεις όπως το NFS, το οποίο υποστηρίζεται από το δικτυακό υποσύστημα.

Το virtio-fs είναι ιδιαίτερο στο γεγονός ότι, ως σύστημα αρχείων εξαρτάται από την ομώνυμη συσκευή, η οποία δεν μπορεί να αντιμετωπιστεί ως μέλος κάποιας από τις συνήθεις οικογένειες (πχ block ή δικτύου). Αυτό το χαρακτηριστικό το κληρονομεί βεβαίως από το FUSE. Έτσι, αναπόφευκτα υπάρχει ισχυρή, ρητή εξάρτηση του virtio-fs συστήματος αρχείων από τον οδηγό της virtio-fs συσκευής, κάτι που βλέπουμε και στην περίπτωση του linux, όπου αυτά τα δύο είναι υλοποιημένα μαζί (στο fs/fuse/virtio_fs.c). Αυτό σημειώνεται ενόψει της παρουσίασης της υλοποίησης που ακολουθεί, η οποία γίνεται υπό το πρίσμα της διάκρισης ανάμεσα σε οδηγό και σύστημα αρχείων, αν και αυτή η διάκριση (δηλαδή ο αντίστοιχος καταμερισμός των λειτουργιών) είναι εύκαμπτη.

3.1.1 Οδηγός

Ο οδηγός της virtio-fs συσκευής είναι επιφορτισμένος με την απευθείας επικοινωνία με αυτή και τη χαμηλού επιπέδου διαχείριση της. Πέρα από την αρχικοποίηση (ανακάλυψη virtqueues, διαπραγμάτευση παραμέτρων λειτουργίας) [15], το κύριο έργο του είναι η μεταφορά αιτημάτων (requests) και απαντήσεων (responses) σε αυτά από και προς τη συσκευή μέσω των virtqueues. Γι΄ αυτό το σκοπό παρέχει μία ασύγχρονη διεπαφή υποβολής αιτημάτων, στα οποία ενθυλακώνονται τα FUSE αιτήματα. Ο οδηγός παραμένει αγνωστικός ως προς το FUSE, χειριζόμενος τα αιτήματα του ως αδιαφανή.

Η αρχικά διαθέσιμη υλοποίηση του virtio-fs DAX window στο QEMU ήταν ως συσκευή PCI. Σύμφωνα με την προδιαγραφή του virtio [15] για τη συσκευή, το DAX window είναι η περιοχή κοινής μνήμης (shared memory region) με αναγνωριστικό ('shmid') 0. Στην περίπτωση του virtio PCI transport, οι περιοχές κοινής μνήμης υλοποιούνται ως PCI BARs (base address registers), κάθε ένα εκ των οποίων εκτίθεται ως ένα VIRTIO_PCI_CAP_SHARED_MEMORY_CFG PCI capability, έχοντας ένα μοναδικό αναγνωριστικό [15, 18, 6].

Το πρώτο βήμα για την προσθήκη του DAX window ήταν λοιπόν η ανακάλυψη (discovery) αυτού, εφόσον παρέχονταν από τη συσκευή. Αυτή ήταν μία ανώδυνη εργασία, χάρη στο πλήρες και καλά μοντελοποιημένο υποσύστημα PCI του OSv. Συγκεκριμένα, οι επιμέρους αλλαγές που χρειάστηκαν ήταν:

1. Προσθήκη στο PCI υποσύστημα ώστε να είναι δυνατή η ανακάλυψη πολλαπλών capabilities με τον ίδιο τύπο (στο drivers/pci-function.cc). Αυτή ήταν απαραίτητη διότι μέχρι πρότινος οι λειτουργίες ανακάλυψης στο OSv αναζητούσαν μέχρι και το πρώτο capability ενός τύπου, σταματώντας εκεί. Όμως, όπως αναφέραμε προηγουμένως, κάθε περιοχή κοινής μνήμης αντιστοιχεί σε ένα PCI capability και αυτές μπορεί να είναι περισσότερες από μία (αν και στην περίπτωση του virtio-fs υπάρχει μόνο μία).

2. Επέκταση στο μοντέλο των virtio PCI συσκευών (στο drivers/virtio-pci-device.cc) ώστε να ανακαλύπτει όλες τις περιοχές κοινής μνήμης κάθε συσκευής, αναζητώντας τα capabilities αντίστοιχου τύπου και αποθηκεύοντας τα στοιχεία (διεύθυνση και μέγεθος) των BARs που αυτά υποδεικνύουν.

3. Επέκταση στον οδηγό της virtio-fs συσκευής (στο drivers/virtio-fs.cc) ώστε να ανακαλύπτει την περιοχή κοινής μνήμης με αναγνωριστικό 0, δηλαδή το DAX window και να το εκθέτει απευθείας στη διεπαφή του, ως περιοχή MMIO (memory-mapped I/O) της συσκευής. Σημειώνεται ότι η απεικόνιση (mapping) της περιοχής στον εικονικό χώρο διευθύνσεων έχει ήδη πραγματοποιηθεί από το μοντέλο της συσκευής.

3.1.2 Σύστημα αρχείων

Το virtio-fs σύστημα αρχείων θεωρεί δεδομένο έναν μηχανισμό αποστολής FUSE αιτημάτων (ο οποίος παρέχεται από τον οδηγό) και πάνω σε αυτόν χτίζει τη λειτουργικότητα του. Είναι συνεπώς επιφορτισμένο με τη γνώση και τον χειρισμό του περιεχομένου των αιτημάτων που ο οδηγός απλώς μεταφέρει.

3.1.2.1 Απεικόνιση αρχείων

Η προσάρτηση (mounting) ενός virtio-fs συστήματος αρχείων συνίσταται στην έναρξη μίας FUSE συνεδρίας (session) με την αποστολή ενός FUSE_INIT αιτήματος [15]. Με αυτό πραγματοποιείται η διαπραγμάτευση των παραμέτρων της συνεδρίας, μεταξύ των οποίων και μίας που αφορά τη λειτουργία του DAX window: η ευθυγράμμιση των απεικονίσεων (map alignment). Η πρώτη τροποποίηση που απαιτούνταν στο σύστημα αρχείων λοιπόν ήταν η επέκταση της διαδικασίας προσάρτησης ώστε να γνωστοποιείται στη συσκευή ότι υποστηρίζεται από το λειτουργικό το DAX window και να λαμβάνεται η τιμή της παραμέτρου από την απάντηση.

Η παραπάνω παράμετρος περιορίζει τα mappings αρχείων στο DAX window ως εξής: τόσο η μετατόπιση (offset) της αρχής ενός mapping μέσα στο αρχείο όσο και στο DAX window πρέπει να είναι ευθυγραμμισμένες σύμφωνα με το map alignment. Αυτό στην πράξη επιβάλλεται στην παρούσα υλοποίηση της virtio-fs συσκευής από τη χρήση της `mmap()` [5] στον host (επίσης ο λόγος που το map alignment συνήθως ταυτίζεται με το μέγεθος της σελίδας μνήμης στον host).

Όπως ορίζεται από την προδιαγραφή, η παροχή του DAX window από τη συσκευή όπως και η χρησιμοποίηση του από τον guest είναι αμφότερα προαιρετικά. Επίσης, η χρήση του DAX window είναι ανεξάρτητη από τη χρήση της συμβατικής FUSE_READ (που είναι πάντοτε διαθέσιμη) για την ανάγνωση αρχείων. Έτσι, η υλοποίηση μας, εφόσον είναι διαθέσιμο το DAX window, επιχειρεί να ικανοποιήσει όλες τις αναγνώσεις αρχείων μέσω εκείνου. Σε περίπτωση όμως που αυτό δεν είναι διαθέσιμο ή αποτύχει η διαδικασία της ανάγνωσης από εκείνο, στρέφεται δυναμικά στην εφεδρική FUSE_READ, όπως φαίνεται και στο διάγραμμα 3.3.

Για την εγκαθίδρυση ενός νέου mapping, το virtio-fs επεκτείνει το πρωτόκολλο του FUSE συστήνοντας τη λειτουργία FUSE_SETUPMAPPING. Ένα αίτημα τέτοιου τύπου περιγράφεται από μία δομή (struct) fuse_setupmapping_in (ο ορισμός φαίνεται στον κώδικα 3.1). Για την ακύρωση υπαρχόντων απεικονίσεων έχει προστεθεί η FUSE_REMOVEMAPPING, με το σώμα της να αποτελείται από ένα struct fuse_removemapping_in ακολουθούμενο από το υποδεικνυόμενο πλήθος από struct fuse_removemapping_one.

Τέλος, σημειώνουμε ότι σύμφωνα με την προδιαγραφή, ένα νέο mapping μπορεί να ζητηθεί ενώ επικαλύπτει κάποιο υπάρχον. Εάν η κάλυψη είναι πλήρης, το προϋπάρχον αντικαθίσταται από το νέο, ενώ αν επικαλύπτεται μερικώς, διασπάται. Ένα αίτημα για νέο mapping μπορεί να αποτύχει σε περίπτωση εξάντλησης πόρων της συσκευής, οπότε προτείνεται να δοκιμαστεί ξανά κατόπιν απελευθέρωσης πόρων μέσω της ακύρωσης υπαρχουσών απεικονίσεων. Έτσι και στην υλοποίηση μας η λειτουργία

```
#define FUSE_SETUPMAPPING_FLAG_WRITE (1ull << 0)
struct fuse_setupmapping_in {
    /* An already open handle */
    uint64_t        fh;
    /* Offset into the file to start the mapping */
    uint64_t        foffset;
    /* Length of mapping required */
    uint64_t        len;
    /* Flags, FUSE_SETUPMAPPING_FLAG_* */
    uint64_t        flags;
    /* memory offset in to dax window */
    uint64_t        moffset;
};

struct fuse_removemapping_in {
    /* number of fuse_removemapping_one follows */
    uint32_t        count;
};

struct fuse_removemapping_one {
    /* Offset into the dax to start the unmapping */
    uint64_t        moffset;
    /* Length of mapping required */
    uint64_t        len;
};
```

Κώδικας 3.1: Ορισμοί του FUSE για τις λειτουργίες απεικονίσεων. Βλέπε παράρτημα A για σημείωμα πνευματικών δικαιωμάτων.

FUSE_REMOVEMAPPING δεν χρησιμοποιείται υπό φυσιολογικές συνθήκες.

3.1.2.2 Διαχειριστής (DAX window manager)

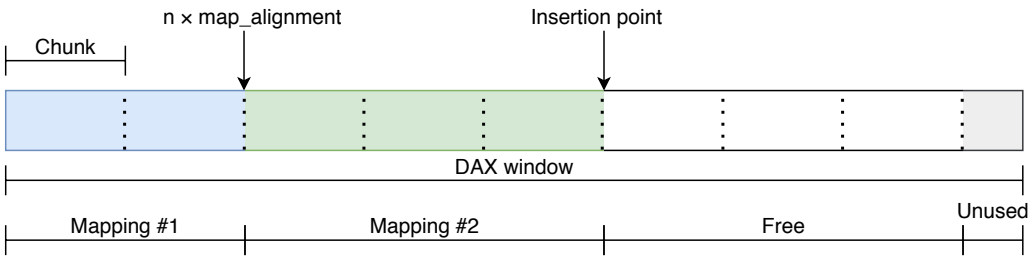
Από τη δυνατότητα λεπτού ελέγχου των DAX window mappings που προσφέρεται, προκύπτει το ζήτημα της βέλτιστης διαχείρισης του. Το τελευταίο έχει πολλά κοινά με το πρόβλημα της διαχείρισης μνήμης στο πλαίσιο ενός λειτουργικού συστήματος με σελιδοποίηση (paging), μιας και λόγω του map alignment οι απεικονίσεις γίνονται πάντα σε όρους πολλαπλασίων αυτού. Για να το αντιμετωπίσουμε λοιπόν χρειάστηκε να υλοποιήσουμε έναν διαχειριστή για το DAX window στο επίπεδο του συστήματος αρχείων, μέσω του οποίου συμβαίνουν όλες οι αναγνώσεις, επιτρέποντας του να εφαρμόσει την πολιτική που περιγράφουμε στη συνέχεια.

Για την διαμόρφωση της πολιτικής διαχείρισης λήφθηκαν υπόψη τα εξής:

- Στην περίπτωση μας, το σύστημα αρχείων είναι μόνο για ανάγνωση (read-only).
- Το κόστος σε χρόνο ενός αιτήματος FUSE_SETUPMAPPING είναι ανεξάρτητο του μεγέθους της απεικόνισης η οποία ζητείται.
- Μία σχετικά απλή πολιτική μεταφράζεται σε απλούστερη υλοποίηση, η οποία είναι ευκολότερο να είναι ορθή, αποδοτική και εύρωστη (robust).

Σύμφωνα με αυτά οδηγηθήκαμε σε ένα σχήμα διαχείρισης που συνοψίζεται από τα παρακάτω χαρακτηριστικά (απεικονίζονται μερικώς και στο σχήμα 3.2):

- Το DAX window χωρίζεται σε κομμάτια (*chunks*) σταθερού μεγέθους (2 MiB από προεπιλογή). Αυτά αποτελούν τους δομικούς λίθους με την έννοια ότι όλες οι λειτουργίες (απεικονίσεις) γίνονται σε όρους αυτών των chunks. Κάθε νέο mapping λοιπόν ξεκινά από μία μετατόπιση εντός του αρχείου και εντός του DAX window, που αμφότερες είναι πολλαπλάσια του μεγέθους του chunk, ενώ αφορά ακέραιο πλήθος chunks. Γίνεται έτσι σαφές ότι απαιτώντας το μέγεθος του chunk να είναι συμβατό με το map alignment, αυτόματα ικανοποιούνται όλες οι απαιτήσεις ευθυγράμμισης που τίθενται από τη virtio-fs συσκευή. Αυτό επελέγη αφενός για απλότητα και αφετέρου ώστε να λειτουργεί (με την επιλογή του κατάλληλου μεγέθους chunk) ως μονάδα προφόρτωσης (prefetching).
- Ένα νέο mapping ξεκινά πάντοτε από τη χαμηλότερη διαθέσιμη διεύθυνση εντός του DAX window. Εάν δεν υπάρχει αρκετός χώρος (το παράθυρο είναι γεμάτο), τότε γίνεται επικάλυψη με τις απεικονίσεις στις υψηλότερες διευθύνσεις, μόνο όσο χρειάζεται ώστε να χωρέσει η νέα. Έτσι, ακολουθείται ένα LIFO (Last In - First Out) μοντέλο, με το DAX window να προσιδιάζει μία στοίβα. Αυτό έγινε αφενός για απλότητα και αφετέρου ώστε στην περίπτωση μίας εφαρμογής που διαβάζει ένα αρχείο σειριακά με διαδοχικές κλήσεις, τα επιμέρους διαδοχικά chunks να βρίσκονται διατεταγμένα στο παράθυρο.



Σχήμα 3.2: Ενδεικτική εικόνα του DAX window υπό τον manager.

- Δεν γίνεται προφόρτωση (prefetching) των αρχείων πέρα του ενός chunk, προκειμένου να αποφευχθεί δυνητική σπατάλη του χώρου στο DAX window. Παρ' αυτά, σημειώνουμε ότι στην υλοποίηση περιλαμβάνεται η επιλογή (απενεργοποιημένη από προεπιλογή) για 'επιθετικό' prefetching, όπου απεικονίζεται ολόκληρο το αρχείο με την πρώτη πρόσβαση σε αυτό.

Ολόκληρη η διαδικασία ανάγνωσης σε ότι αφορά την υλοποίηση του virtio-fs απεικονίζεται ενδεικτικά στο διάγραμμα ροής 3.3.

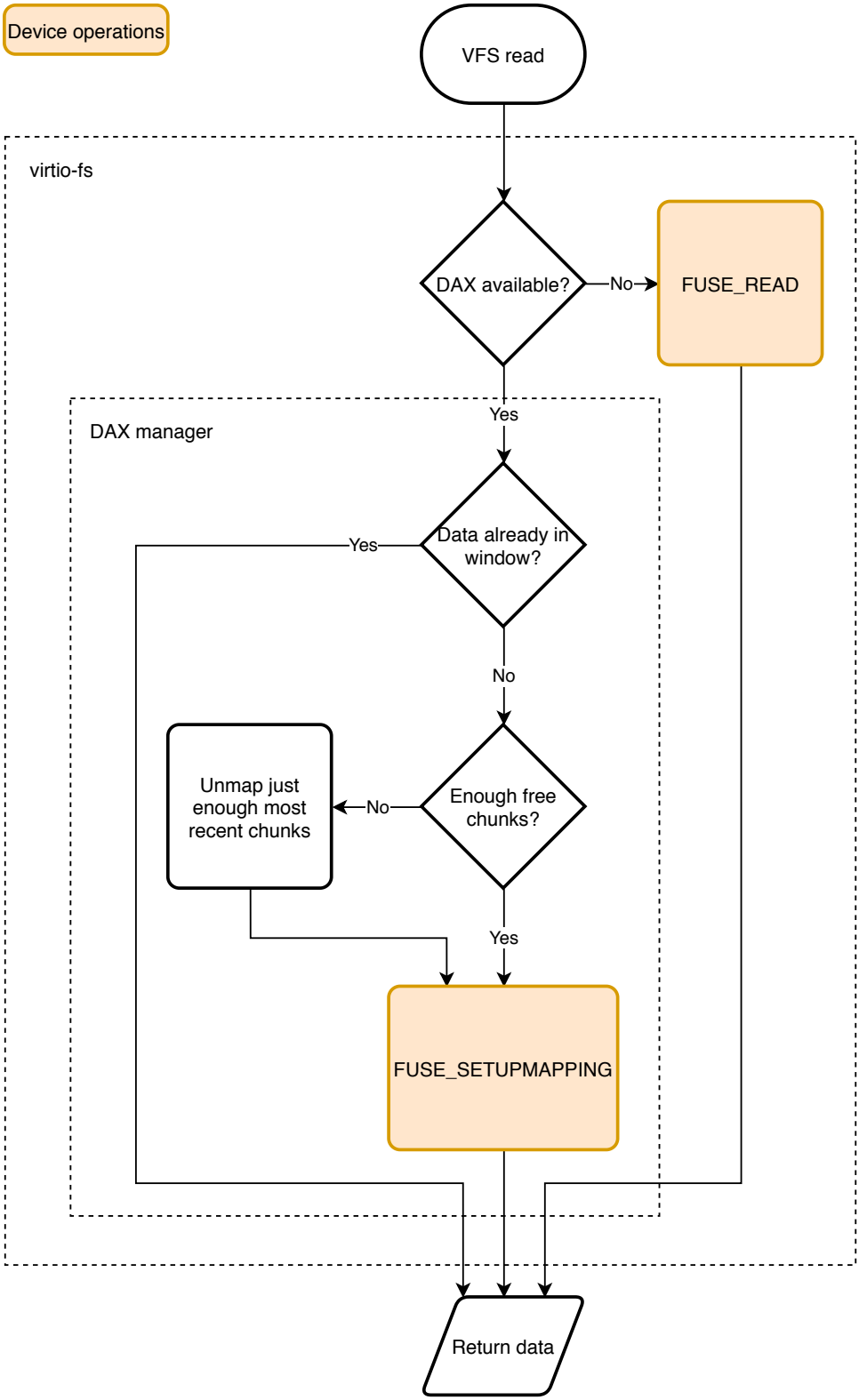
3.2 Boot από virtio-fs

Η προσθήκη υποστήριξης για εκκίνηση (boot) από το virtio-fs στο OSv είχε δύο συνιστώσες: την κυριότερη που αφορούσε τον πυρήνα του λειτουργικού συστήματος αλλά και μία βοηθητική που αφορούσε τα εργαλεία αυτοματισμού για τη δημιουργία και εκτέλεση των εικόνων. Αμφότερες βασίστηκαν σε προηγούμενες αντίστοιχες επεκτάσεις, μιας και το OSv ήδη μπορούσε να χρησιμοποιήσει είτε το ZFS είτε το rofs (και το ramfs βέβαια, που είναι ιδιαίτερη περίπτωση) για το κύριο (root) σύστημα αρχείων του.

Συνοπτικά, τα σημεία που αφορούν το root file system κατά τον κύκλο ζωής ενός OSv unikernel, ως είχαν πριν την επέκταση μας είναι:

1. Χτίσιμο της εικόνας (κεντρικό σημείο το scripts/build): σε αυτό το στάδιο επιλέγεται από τον χρήστη ο τύπος του συστήματος αρχείων και το root file system δημιουργείται με το περιεχόμενο που προκύπτει από την εφαρμογή. Επίσης, εδώ ξεκινά ο καθορισμός της εντολής εκκίνησης (command line) του (uni)kernel, η οποία περιέχει επιλογές για τον ίδιο τον πυρήνα, αλλά και το command line της εφαρμογής που πρόκειται να εκτελεστεί. Αυτή αποθηκεύεται σε ένα προσωρινό αρχείο, από όπου την παραλαμβάνει το επόμενο βήμα [7].

2. Εκτέλεση της εικόνας (κεντρικό σημείο το scripts/run.py): εδώ προαιρετικά εμπλουτίζεται το command line με εφήμερες επιλογές, για παράδειγμα το επίπεδο λεπτομέρειας (verbosity) στα μηνύματα του πυρήνα. Κατόπιν, το ολοκληρωμένο command line εγγράφεται στο image, προτού αυτό εκτελεστεί.

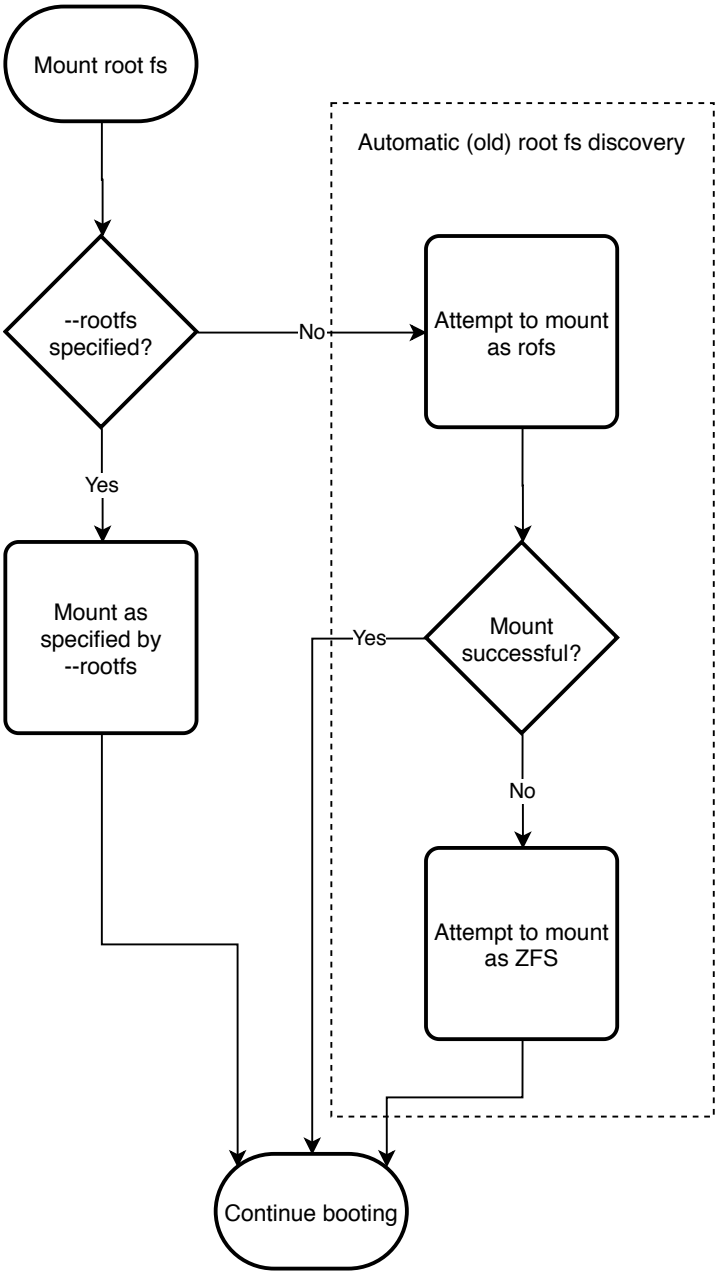


Σχήμα 3.3: Διαδικασία ανάγνωσης από το virtio-fs υπό τον manager.

3. Φόρτωση του πυρήνα (κεντρικό σημείο το loader.cc): μετά από, μεταξύ άλλων, την αρχικοποίηση του πυρήνα, των συσκευών και των οδηγών, αλλά και το διάβασμα του command line, το OSv επιχειρεί να περάσει από το αρχικό, ενσωματωμένο σύστημα αρχείων του (ramfs) στο κύριο σύστημα αρχείων (root file system) [10]. Η διαδικασία ετούτη συνίσταται στην προσάρτηση (mounting) του συστήματος αρχείων σε κάποιο σημείο του υπάρχοντος, αρχικού ramfs, ακολουθούμενη από το πέρασμα της ρίζας (pivoting) του εικονικού συστήματος αρχείων σε αυτό. Τα τελευταία γίνονται απλά, μέσω του εικονικού συστήματος αρχείων, αφού όλα τα προαπαιτούμενα για τη χρήση του έχουν ολοκληρωθεί.

Η επιλογή του συστήματος αρχείων που θα χρησιμοποιηθεί ως root κατά τη φόρτωση μέχρι πρότινος ήταν δυναμική: γινόταν απόπειρα προσάρτησης από την πρώτη συσκευή block αρχικά του rofs και εάν αυτό αποτύγχανε, του ZFS. Εάν αμφότερα αποτύγχαναν, η εκτέλεση συνεχιζόταν με το ramfs αρχικό σύστημα αρχείων. Αν και η προηγούμενη διαδικασία θα μπορούσε εύκολα να επεκταθεί ώστε να συμπεριλάβει το virtio-fs, γινόταν εμφανές ότι βασιζόταν σε πολλές σιωπηλές υποθέσεις, χωρίς να αφήνει μεγάλο περιθώριο ελέγχου στον χρήστη. Έτσι αποφασίσαμε να πάμε ένα βήμα παραπέρα, ώστε να δοθεί περισσότερος έλεγχος επί της διαδικασίας: προσθέσαμε μία επιλογή `--rootfs` στη γραμμή εντολών του πυρήνα (command line option), η οποία επιτρέπει να καθοριστεί ρητά ο τύπος του root file system (ανάμεσα σε ZFS, rofs, virtio-fs και ramfs). Όπως φαίνεται και στο διάγραμμα 3.4, αυτή η επιλογή είναι προαιρετική και εάν δεν έχει καθοριστεί χρησιμοποιείται η παλαιότερη, δυναμική διαδικασία, για συμβατότητα. Τέλος, κάναμε τις απαραίτητες αλλαγές στη διαδικασία χτισίματος (scripts/build) ώστε να θέτει αυτή την επιλογή ανάλογα με το σύστημα αρχείων που έχει καθορίσει ο χρήστης κατά το χτίσιμο. Εξάλλου, με αυτόν τον τρόπο έγινε πιο προβλέψιμη και τεκμηριώθηκε καλύτερα η διαδικασία καθώς και οι διαθέσιμες επιλογές για το root file system.

Τέλος, όσον αφορά καθεαυτό το virtio-fs, η χρήση του ως root file system δεν ενείχε κάποια αξιοσημείωτη πρόκληση: εάν επιλέγονταν μέσω του `--rootfs`, γινόταν προσάρτηση του από την πρώτη virtio-fs συσκευή (στο OSv, σε αντίθεση με το linux, αυτές εκτίθενται στο devfs, αριθμημένες σειριακά αντί να αναγνωρίζονται από το virtio-fs tag τους). Προκειμένου να συμπληρωθεί αυτόματα με τα κατάλληλα περιεχόμενα για την εκάστοτε εφαρμογή ένας κατάλογος στον host (ο οποίος στη συνέχεια θα χρησιμοποιηθεί ως το virtio-fs shared directory), είναι διαθέσιμες οι παράμετροι `export` και `export_dir` του scripts/build.



Σχήμα 3.4: Διαδικασία προσάρτησης του root file system.

Κεφάλαιο 4

Αξιολόγηση

Για την αξιολόγηση του νέου συστήματος αρχείων στο OSv, διεξάγαμε δοκιμές συγκρίνοντας με τα υπόλοιπα διαθέσιμα συστήματα αρχείων αλλά και το virtio-fs στο linux, όπου αυτή η σύγκριση είχε νόημα. Οι δοκιμές περιλαμβάνουν τρία σενάρια, όπως αναλύονται στη συνέχεια: ένα συνθετικό μετροπρόγραμμα (synthetic benchmark ή microbenchmark), μια δοκιμή χρόνου εκκίνησης και τέλος μία πραγματική εφαρμογή (application benchmark).

4.1 Μεθοδολογία

Όλες οι δοκιμές πραγματοποιήθηκαν σε προσωπικό υπολογιστή με τα στοιχεία που αναφέρονται στον πίνακα 4.1, ενώ ο πίνακας 4.2 περιλαμβάνει τις προδιαγραφές των OSv και linux guests. Ως προς τη διεξαγωγή τους πρέπει να αναφερθούν τα εξής:

- Για την εκτέλεση όλων των δοκιμών χρησιμοποιήθηκε ένα προσωρινό σύστημα αρχείων (tmpfs) στον host, τόσο για τις εικόνες του OSv όσο και για τους κοινόχρηστους καταλόγους στην περίπτωση των virtio-fs και NFS. Αυτό έγινε ώστε να μην επηρεάζονται οι επιδόσεις από τη συσκευή αποθήκευσης του host.

- Η δυναμική προσαρμογή της συχνότητας της CPU ήταν απενεργοποιημένη, χρησιμοποιώντας τον "performance" CPU scaling governor στον host.

- Η διεργασία του QEMU ήταν απομονωμένη από τις υπόλοιπες (virtiofsd, perf, vegeta) ως προς τις CPUs στις οποίες εκτελούνταν (CPU pinning). Συγκεκριμένα, στο QEMU αφιερώνονταν επεξεργαστικοί πυρήνες ισάριθμοι με το πλήθος των CPUs του guest, ενώ οι υπόλοιπες προαναφερόμενες διεργασίες δεσμεύονταν στους υπόλοιπους διαθέσιμους. Αυτό έγινε με σκοπό την ελαχιστοποίηση των παρεμβολών που θα μπορούσαν να επηρεάσουν τα αποτελέσματα.

- Για τη μέτρηση της χρησιμοποιούμενης επεξεργαστικής ισχύος (CPU usage) επελέγη το perf [12], στην έκδοση 5.7.g3d77e6a8804a. Συγκεκριμένα, χρησιμοποιήθηκε η εντολή `perf stat`, με το "task-clock" perf event. Η μέτρηση

Επεξεργαστής	Intel Core i7-6700 @3.4GHz
Μνήμη	2×8 GiB @2666MHz
Swap	Όχι
linux kernel	5.8.13-arch1-1
QEMU	5.1.50 @ c37a890d12e57a3d28c3c7ff50ba6b877f6fc2cc [17]

Πίνακας 4.1: Προδιαγραφές του host όπου διεξήχθησαν οι δοκιμές.

έγινε για τη διεργασία του QEMU, το virtiofsd, το vhost kernel thread [3] και τα NFS kernel threads (για το καθένα όπου ήταν εφαρμόσιμο).

- Όλες οι δοκιμές επαναλήφθηκαν 10 φορές, από τις οποίες στη συνέχεια παρουσιάζονται η μέση τιμή (mean) και η τυπική απόκλιση (standard deviation). Στην περίπτωση του OSv κάθε επανάληψη ήταν μία εκ νέου εκκίνηση της εικονικής μηχανής, ενώ στο linux όλες οι επαναλήψεις έγιναν στην ίδια εκτέλεση της εικονικής μηχανής.

- Το virtiofsd εκτελούνταν με το cache mode απενεργοποιημένο (`cache=none`) και ένα thread (`--thread-pool-size=1`). Το δεύτερο επελέγη διότι οδηγούσε σε ελαφρώς πιο συνεπείς μετρήσεις, χωρίς όμως να παρατηρείται καλύτερη επίδοση όπως είχε αναφερθεί σε συζήτηση στη mailing list του virtio-fs[1].

- Για την εκτέλεση του OSv έγινε χρήση του βοηθητικού script (scripts/run.py) που παρέχει, το οποίο τροποποιήθηκε για την διεξαγωγή των δοκιμών, ώστε να ενορχηστρώνει την εκτέλεση όλων των υπόλοιπων εργαλείων (virtiofsd, perf, vegeta). Ως προς τη δικτύωση της εικονικής μηχανής, χρησιμοποιήθηκε το tap backend του QEMU με το vhost ενεργοποιημένο, ενώ στο VM δινόταν στατική διεύθυνση IP.

- Ως NFS server χρησιμοποιήθηκε η αντίστοιχη υλοποίηση του linux στον host. Όλες οι δοκιμές έγιναν με την έκδοση 3 του NFS, καθώς η έκδοση 4 δεν ήταν λειτουργική από την πλευρά του OSv. Το readahead στον NFS client (libnfs), αντίστοιχο του μεγέθους των chunks στην υλοποίηση μας του DAX window manager, ήταν ίσο με 2 MiB.

4.2 Microbenchmark

4.2.1 Περιγραφή

Για τη μέτρηση των επιδόσεων του συστήματος αρχείων χρησιμοποιήσαμε το flexible I/O tester (fio) [1] στην έκδοση 3.23. Προκειμένου να εκτελεστεί στο OSv χρει-

[1] `https://www.redhat.com/archives/virtio-fs/2020-September/msg00068.html`

CPUs	4
Μνήμη	4 GiB
DAX window	4 GiB
OSv	5372a230ce0abf0dc72e92ec1116208145e595c5 [11]
linux kernel	5.8.0-rc4-33261-gfaa931f16f27 [16]

Πίνακας 4.2: Προδιαγραφές των guests.

άστηκαν μόνο δύο τροποποιήσεις[2] και τα κατάλληλα ορίσματα στο configure script του fio ώστε να απενεργοποιηθούν χαρακτηριστικά που δεν παρέχονται από το OSv. Σημειώνουμε ότι το ίδιο ακριβώς εκτελέσιμο (με τα παραπάνω απενεργοποιημένα) χρησιμοποιήθηκε και στο linux, εκτός από το OSv.

Στη σύγκριση περιλαμβάνονται τα ZFS, rofs, ramfs, virtio-fs (με και χωρίς DAX window και με ramfs root file system) και NFS στο OSv, το virtio-fs (με και χωρίς DAX window και με ext4 root file system) στο linux, καθώς και το tmpfs στον host, το οποίο συμπεριλάβαμε για πληρότητα, ως σημείο αναφοράς (baseline). Συγκεκριμένα για τη διαδικασία έχουμε:

- Σε όλες τις περιπτώσεις εκτός του ramfs, τα αρχεία ελέγχου (test files) του fio έχουν παραχθεί εκ των προτέρων από το ίδιο και έχουν τοποθετηθεί στην εκάστοτε τοποθεσία (εικονικό δίσκο ή κοινόχρηστο κατάλογο). Αυτό γίνεται παρότι το fio μπορεί να τα παράξει δυναμικά κατά το χρόνο εκτέλεσης, διότι κάποια από τα συστήματα αρχείων είναι read-only. Επειδή το ramfs είναι περιορισμένο ως προς το μέγεθος μεμονωμένων αρχείων στο image του, στην περίπτωση του τα αρχεία παράγονται κατά το χρόνο εκτέλεσης.

- Εξετάζονται δύο περιπτώσεις ως προς τα αρχεία ελέγχου:

  - Ένα μεγάλο αρχείο (1 GiB).
  - Πολλαπλά (10) μικρότερα αρχεία (80-100 MiB). Σημειώνουμε ότι τα αρχεία σε όλες τις δοκιμές είναι πανομοιότυπα (το αντίστοιχο αρχείο έχει το ίδιο μέγεθος πάντα).

  Και στις δύο περιπτώσεις το συνολικό μέγεθος των αρχείων δεν ξεπερνά το 1 GiB. Αυτό εν μέρει γίνεται διότι το συγκεκριμένο μέγεθος είναι περιορισμένο από τη μνήμη του guest για τα rofs και ramfs και το μέγεθος του DAX window για το virtio-fs με DAX.

- Εξετάζονται δύο περιπτώσεις ως προς το πρότυπο (pattern) ανάγνωσης των αρχείων: σειριακό (`rw=read`) και τυχαίο (`rw=randread`).

- Σε όλες τις περιπτώσεις υπάρχει μόνο ένα thread ανάγνωσης (`numjobs=1`).

[2]Όλες οι αλλαγές βρίσκονται στο 'fio-3.23-osv' tag του git repository `https://github.com/foxeng/fio`.

- Στο linux, για την αυτοματοποίηση των δοκιμών βασιστήκαμε στο σχετικό βοήθημα του Vivek Goyal[3] [2]. Το συγκεκριμένο ακυρώνει τις page, inode και dentry caches χρησιμοποιώντας το sysctl (/proc/sys/vm/drop_caches) πριν από κάθε εκτέλεση του fio.

- Στην περίπτωση του linux (guest και host) δεν έγινε μέτρηση του CPU usage γιατί μία τέτοια σύγκριση κρίθηκε ότι δεν έχει αξία, δεδομένου του διαφορετικού χαρακτήρα του από το OSv (λειτουργικό γενικού σκοπού από τη μία και unikernel από την άλλη).

4.2.2 Αποτελέσματα

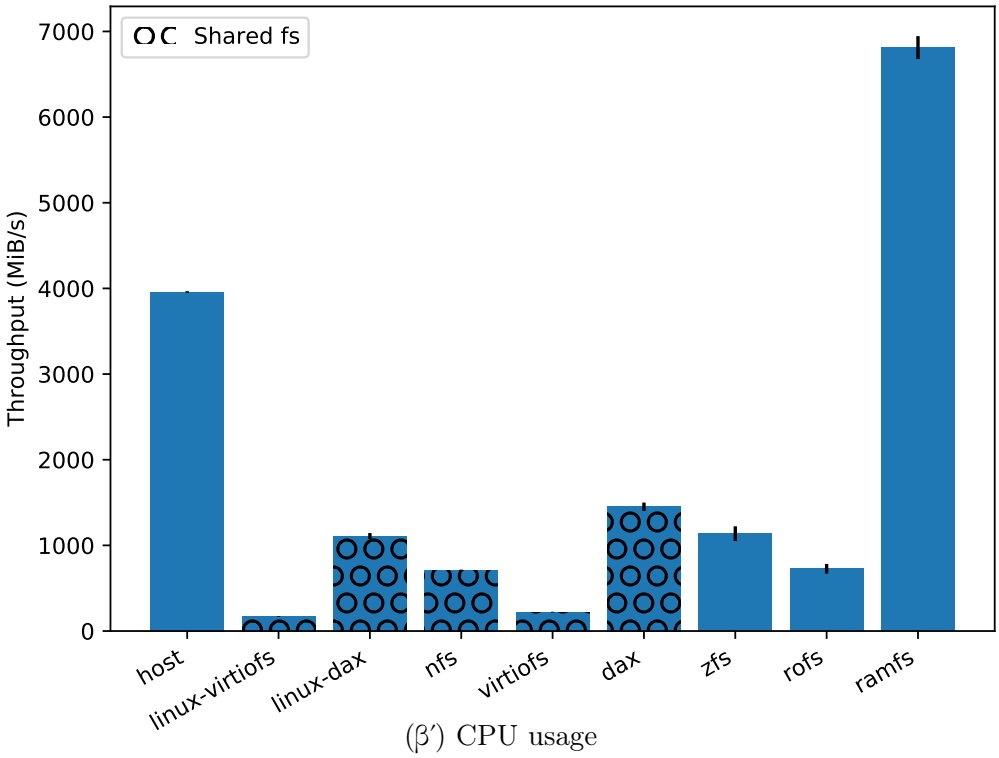
Όπως βλέπουμε στα σχήματα 4.1 έως 4.4, το υψηλότερο throughput επιτυγχάνεται από το ramfs στο OSv, υψηλότερο και από αυτό του tmpfs στον host. Αυτό συμβαίνει διότι, με τα δεδομένα στη μνήμη το virtualization overhead ελαχιστοποιείται, ενώ ταυτόχρονα η απλοποιημένη υλοποίηση του συστήματος αρχείων και η έλλειψη mode switches λόγω κλήσεων συστήματος στο OSv φαίνεται ότι κάνουν τη διαφορά. Το virtio-fs με DAX προσφέρει τις αμέσως καλύτερες επιδόσεις στο OSv, σε όλες τις περιπτώσεις, και παράλληλα έχει τη μικρότερη επιβάρυνση σε όρους επεξεργαστικής ισχύος, και πάλι μετά το ramfs.

Εστιάζοντας στο virtio-fs και συγκρίνοντας ανάμεσα σε OSv και linux, διακρίνουμε συνεπή συμπεριφορά σε όλες τις περιπτώσεις: σε αμφότερα τα λειτουργικά το DAX window υπερτερεί του virtio-fs χωρίς αυτό με μεγάλη διαφορά ($> 6\times$ throughput), ενώ στο OSv βλέπουμε $20 - 30\%$ καλύτερες επιδόσεις από ότι στο linux. Το δεύτερο είναι αναμενόμενο, αφενός λόγω της απλούστερης οπότε και αποδοτικότερης υλοποίησης στο OSv, που υποστηρίζει πολύ λιγότερες λειτουργίες και αφετέρου λόγω των συγκριτικών πλεονεκτημάτων ενός unikernel.

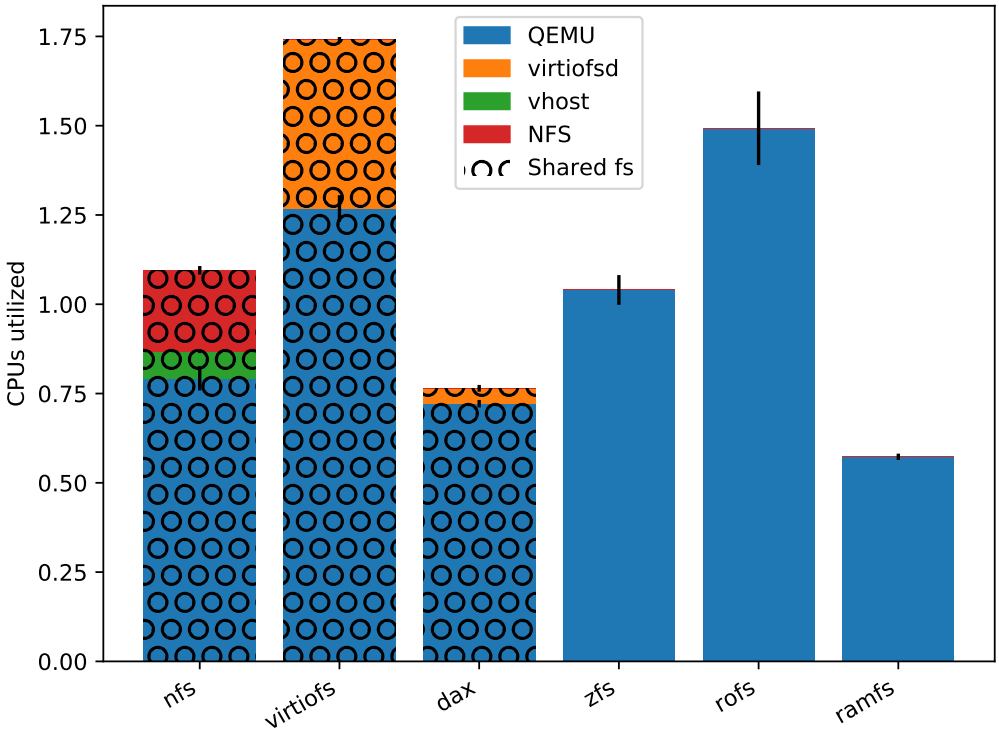
Ιδιαίτερη μνεία οφείλουμε στη σύγκριση ανάμεσα σε virtio-fs και NFS, τα μόνα κοινόχρηστα συστήματα αρχείων στο OSv. Εδώ το virtio-fs με DAX window έχει το προβάδισμα στις επιδόσεις, με το NFS να ακολουθεί και το virtio-fs χωρίς DAX να βρίσκεται τελευταίο, στις δοκιμές με σειριακή ανάγνωση. Όπως φαίνεται και στον πίνακα 4.3, στις δοκιμές τυχαίας ανάγνωσης τα αποτελέσματα διαφοροποιούνται, με το NFS να έχει το χειρότερο throughput και μεγαλύτερο CPU usage. Φαίνεται λοιπόν ότι έχει επηρεαστεί σαφώς περισσότερο σε σχέση με το virtio-fs από την αλλαγή στο random access pattern, που θέτει πίεση στους μηχανισμούς πρόβλεψης και caching όλων των συστημάτων αρχείων.

[3]commit hash 8c99f50c878cb39db76abec7e0882fd83c99f4b3

(α′) Throughput

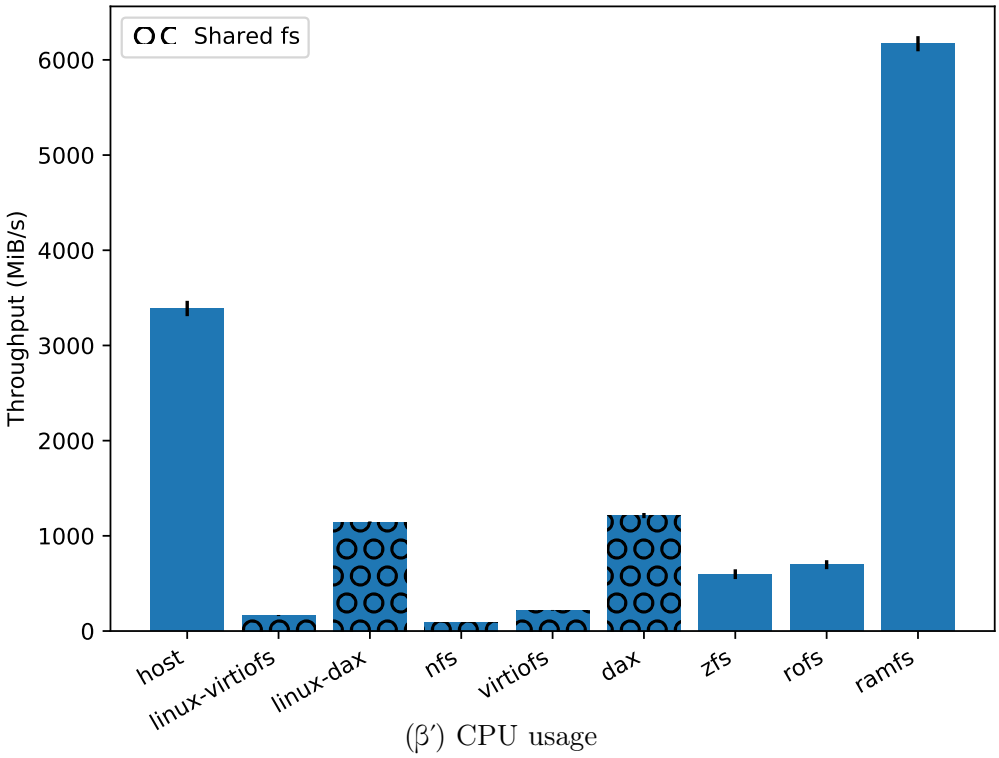


(β′) CPU usage

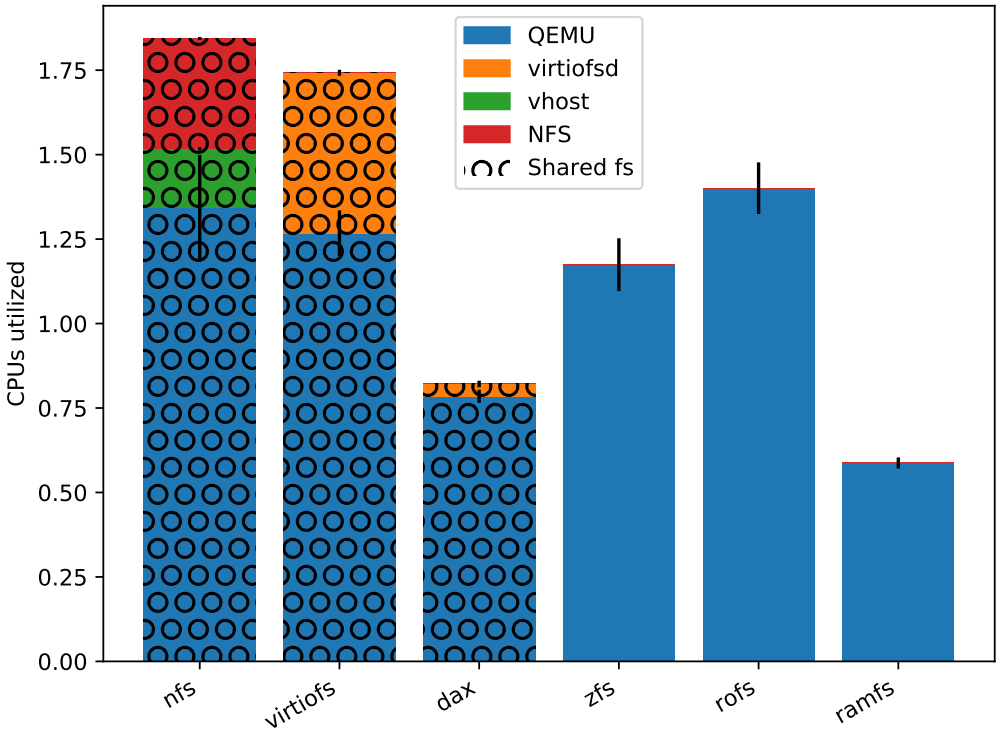


Σχήμα 4.1: fio, ένα αρχείο, σειριακή ανάγνωση

(α′) Throughput

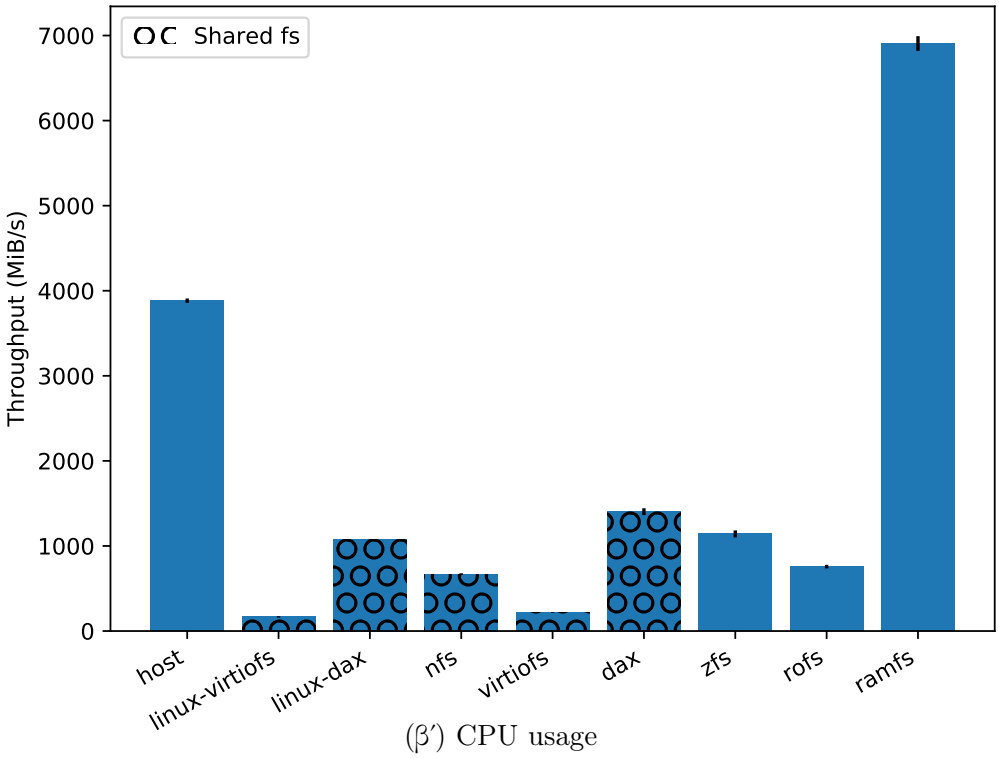


(β′) CPU usage

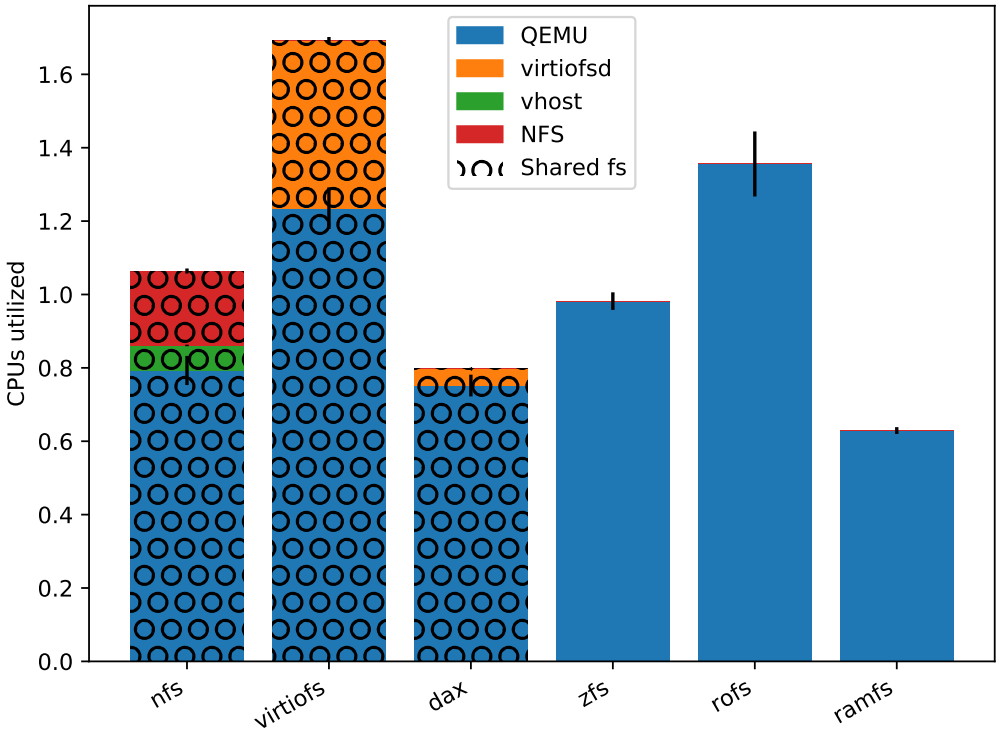


Σχήμα 4.2: fio, ένα αρχείο, τυχαία ανάγνωση

(α′) Throughput

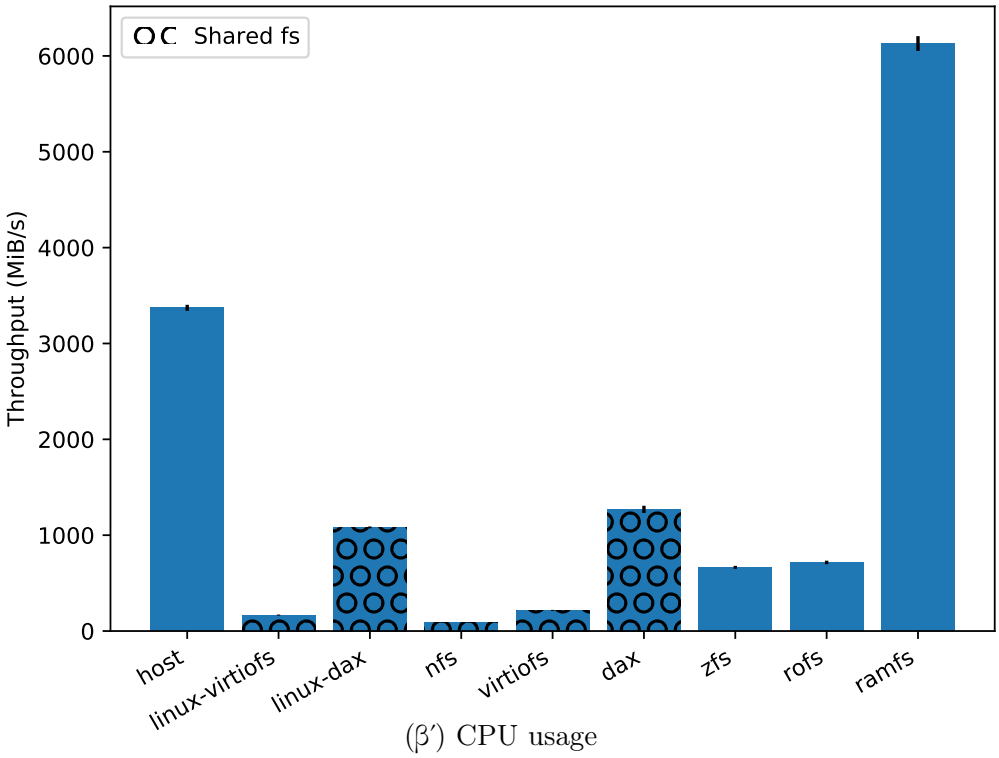


(β′) CPU usage

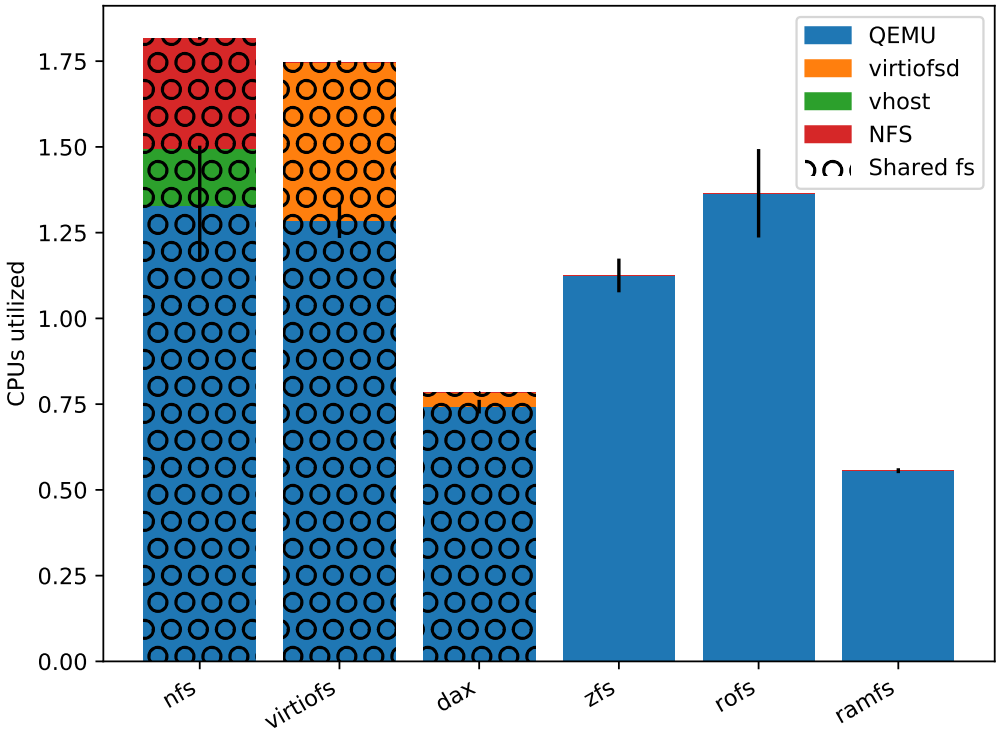


Σχήμα 4.3: fio, πολλαπλά αρχεία, σειριακή ανάγνωση

(α′) Throughput



(β′) CPU usage



Σχήμα 4.4: fio, πολλαπλά αρχεία, τυχαία ανάγνωση

pattern	virtio-fs	NFS	virtio-fs DAX
σειριακό	1,0	3,1	6,5
τυχαίο	1,0	0,4	5,7

Πίνακας 4.3: Κανονικοποιημένο fio throughput virtio-fs και NFS στο OSv.

4.3 Χρόνος εκκίνησης

4.3.1 Περιγραφή

Προκειμένου να αξιολογήσουμε τον χρόνο εκκίνησης στο virtio-fs επιλέξαμε μία εφαρμογή από αυτές που έχουν ήδη μεταφερθεί στο OSv ως παραδείγματα [8]. Συγκεκριμένα, επελέγη το spring-boot-example, μια απλή web εφαρμογή από το [4] χτισμένη με το spring boot framework, στην έκδοση του 2.3.4.[4] Ως Java runtime επιλέξαμε και πάλι από την ίδια συλλογή εφαρμογών το openjdk8-zulu-full. Η επιλογή της συγκεκριμένης εφαρμογής έγινε καθώς θεωρήθηκε αντιπροσωπευτική, ως stateless web app, εφαρμογής που θα γινόταν deploy ως unikernel, σε πλαίσιο cloud.

Στις δοκιμές συμμετείχαν όλα τα συστήματα αρχείων του OSv τα οποία μπορούν να χρησιμοποιηθούν ως root file systems: ZFS, rofs, ramfs και virtio-fs. Η διαδικασία περιελάμβανε την εκκίνηση του OSv με το εκάστοτε σύστημα αρχείων να χρησιμοποιείται ως root file system. Αυτό αφηνόταν να τρέξει για ένα εύλογο χρονικό διάστημα κάποιων δευτερολέπτων, μέσα στο οποίο ολοκληρωνόταν η πλήρης αρχικοποίηση της εφαρμογής και στη συνέχεια τερματιζόταν από το μηχανισμό (script) ενορχήστρωσης των δοκιμών, τερματίζοντας τη διεργασία του QEMU.

4.3.2 Αποτελέσματα

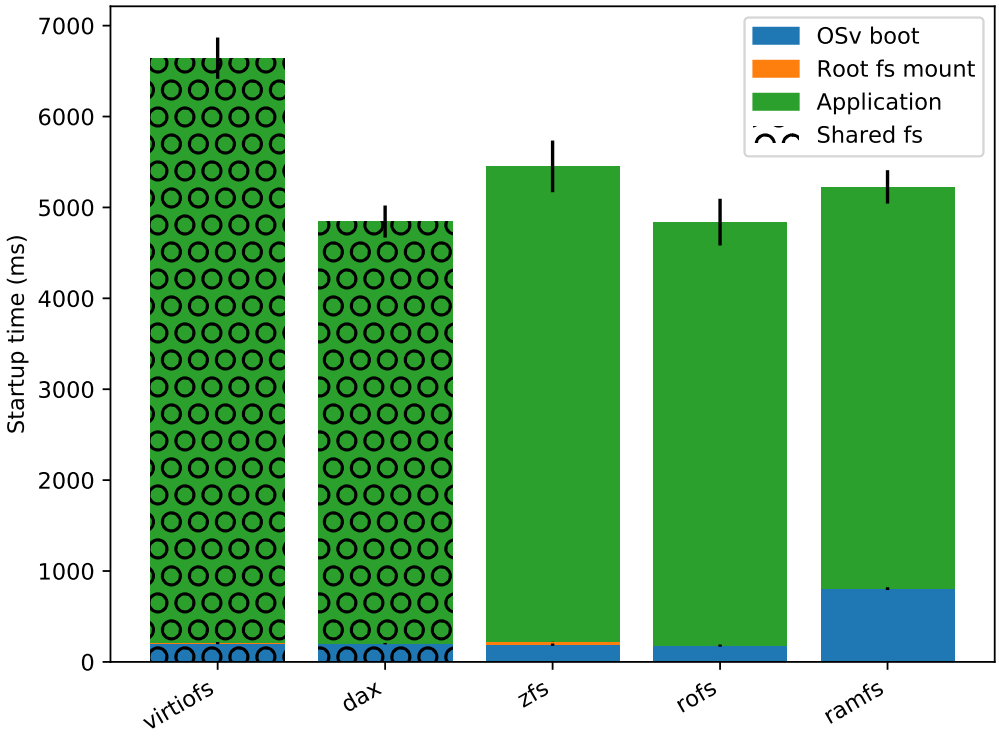
Στο σχήμα 4.5 απεικονίζεται μία ανάλυση του συνολικού χρόνου εκκίνησης ως εξής:

OSv boot είναι ο χρόνος εκκίνησης (boot) του συστήματος του OSv, δηλαδή του μέρους που είναι ανεξάρτητο της εκάστοτε εφαρμογής.

Root fs mount είναι ο χρόνος προσάρτησης (mount) του root file system και της αλλαγής της 'ρίζας' (/) του εικονικού συστήματος αρχείων σε αυτό (pivot). Επισημαίνεται ότι αυτό είναι ένα στάδιο του προηγούμενου, αλλά εν προκειμένω έχει αφαιρεθεί από εκείνο και παρουσιάζεται χωριστά.

Application είναι ο χρόνος αρχικοποίησης της ίδιας της εφαρμογής, όπως καταγράφεται από αυτήν.

[4] Όλες οι αλλαγές που έγιναν για τις δοκιμές μας βρίσκονται στο 'virtio-fs-tests' tag του git repository `https://github.com/foxeng/osv-apps`.



Σχήμα 4.5: Spring boot example, χρόνοι εκκίνησης

ZFS	rofs	ramfs	virtio-fs	virtio-fs DAX
1,13	1,00	1,08	1,37	1,00

Πίνακας 4.4: Κανονικοποιημένος συνολικός χρόνος εκκίνησης spring-boot-example στο OSv.

Όπως βλέπουμε καλύτερα στον πίνακα 4.4, με όρους συνολικού χρόνου εκκίνησης, το virtio-fs χωρίς DAX window υστερεί έναντι των υπολοίπων, ενώ το virtio-fs με DAX window έχει την καλύτερη επίδοση, μαζί με το rofs (τα ramfs και ZFS είναι ελαφρώς πιο αργά).

Όσον αφορά τον χρόνο προσάρτησης (mount time), για όλα τα συστήματα αρχείων είναι πρακτικά αμελητέος (< 2% του OSv boot time), εκτός από το ZFS. Στην περίπτωση του τελευταίου, ο χρόνος προσάρτησης αποτελεί περίπου το 14% του χρόνου εκκίνησης του συστήματος, κάτι αναμενόμενο δεδομένης της πολυπλοκότητας του συγκεκριμένου συστήματος αρχείων, η οποία μεταφράζεται σε μία σαφώς πιο μακρά διαδικασία αρχικοποίησης.

Τέλος, αξίζει να αναφερθούμε στον αισθητά αυξημένο χρόνο εκκίνησης του OSv στην περίπτωση του ramfs. Αυτή η διαφοροποίηση δικαιολογείται εάν αναλογιστούμε ότι στην περίπτωση του ramfs, το root file system ταυτίζεται με το boot file system (σε ορολογία OSv, το αντίστοιχο initramfs στο linux). Αυτό είναι μέρος του ELF object που περιέχει το kernel, το οποίο φορτώνεται και αποσυμπιέζεται κατά τα πρώιμα στάδια του boot [9], σε μια διαδικασία με χαμηλό throughput, λόγω του περιορισμένων δυνατοτήτων του αρχικού περιβάλλοντος κατά την εκκίνηση με BIOS στην αρχιτεκτονική x86. Έτσι, όταν στην περίπτωση του ramfs, το εν λόγω ELF object είναι έως και μία τάξη μεγέθους μεγαλύτερο από τα υπόλοιπα, αυτό είναι υπαίτιο για την αύξηση του χρόνου εκκίνησης. Βέβαια, εν προκειμένω έχουμε κάνει κατάχρηση του ramfs, το οποίο δεν είναι προορισμένο για τόσο μεγάλα images.

4.4 Application benchmark

4.4.1 Περιγραφή

Για να αξιολογήσουμε το virtio-fs με όρους μιας ολοκληρωμένης, πραγματικής εφαρμογής επιλέξαμε και πάλι από το πεδίο των stateless εφαρμογών σε πλαίσιο cloud το σενάριο ενός στατικού web εξυπηρετητή (server). Συγκεκριμένα, χρησιμοποιήσαμε τον nginx [14] (στην έκδοση του 1.19.2), έναν από τους πλέον δημοφιλείς web servers ελεύθερου λογισμικού, ο οποίος είναι επίσης διαθέσιμος στη συλλογή εφαρμογών του OSv.[5]

Στις δοκιμές συμμετείχαν όλα τα συστήματα αρχείων του OSv (στην περίπτωση του virtio-fs, ως root file system χρησιμοποιήθηκε το ramfs) εκτός από το NFS, με το

[5]Όλες οι αλλαγές που έγιναν για τις δοκιμές μας βρίσκονται στο 'virtio-fs-tests' tag του git repository `https://github.com/foxeng/osv-apps`.

οποίο το σενάριο μας αποτύγχανε. Συγκεκριμένα, κατά την εξυπηρέτηση του πρώτου αιτήματος (request) από τον server, μετά την αποστολή λίγων αρχικών δεδομένων της απάντησης (response), η διαδικασία σταματούσε και ο guest φαινόταν 'παγωμένος'. Αυτό διαπιστώθηκε ότι δεν οφειλόταν στον nginx, καθώς την ίδια ακριβώς συμπεριφορά επιδείκνυε το σύστημα με τον lighttpd (επίσης περιλαμβάνεται στη συλλογή του 'osv-apps') στη θέση του. Περαιτέρω διερεύνηση απαιτείται για να εντοπιστεί και πιθανώς να διορθωθεί η αιτία, η οποία από την εμπειρία μας φαίνεται να μοιάζει με κάποιου είδους αδιέξοδο (deadlock) που πιθανώς να εντοπίζεται στην υλοποίηση του NFS ή της στοίβας δικτύωσης του OSv.

Τα αρχεία τα οποία εξέθετε ο web server ήταν συνολικά 10, με μέγεθος που κυμαίνονταν από περίπου 500 KiB μέχρι και 12 MiB. Το μέγεθος κάθε αντίστοιχου αρχείου ήταν σταθερό σε όλες τις δοκιμές.

Για την διεξαγωγή των δοκιμών, που είχαν τη μορφή δοκιμών φόρτου HTTP (HTTP load tests), από τη μεριά του πελάτη (HTTP client) χρησιμοποιήσαμε το vegeta [13], ένα δημοφιλές εργαλείο γι' αυτό το σκοπό, στην έκδοση του 12.8.3. Η διαδικασία (αυτοματοποιημένη από το script ενορχήστρωσης) είχε ως εξής:

1. Αρχικά εκκινούνταν ο OSv guest, στον οποίο δίνονταν περιθώριο ενός δευτερολέπτου προκειμένου να ολοκληρωθεί η αρχικοποίηση του nginx server.

2. Στον host εκκινούσε το vegeta, ρυθμισμένο να παράγει HTTP 1.1 GET requests για όλα τα αρχεία του server, με το μέγιστο δυνατό ρυθμό, χρησιμοποιώντας 20 'εργάτες' και έως 4 CPUs, με την επαναχρησιμοποίηση (keepalive) των συνδέσεων TCP ενεργοποιημένη.

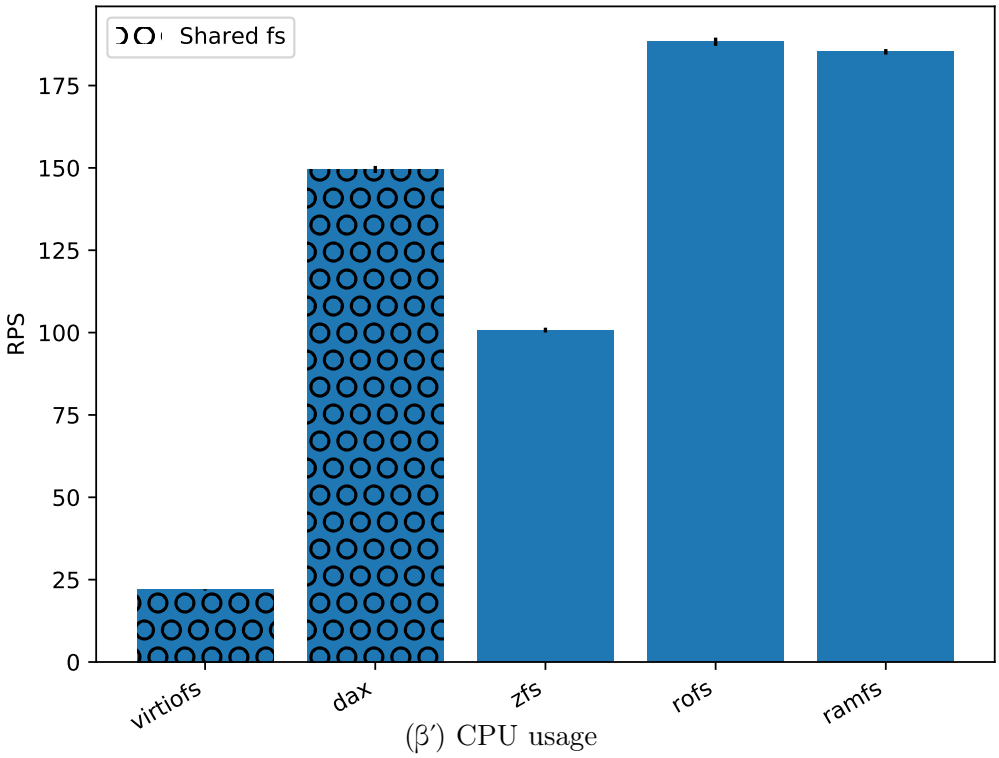
3. Μετά από δέκα δευτερόλεπτα, το vegeta ολοκλήρωνε το έργο του και τερματιζόταν, οπότε και ο αυτοματισμός τερμάτιζε τον guest μέσω της διεργασίας του QEMU.

4.4.2 Αποτελέσματα

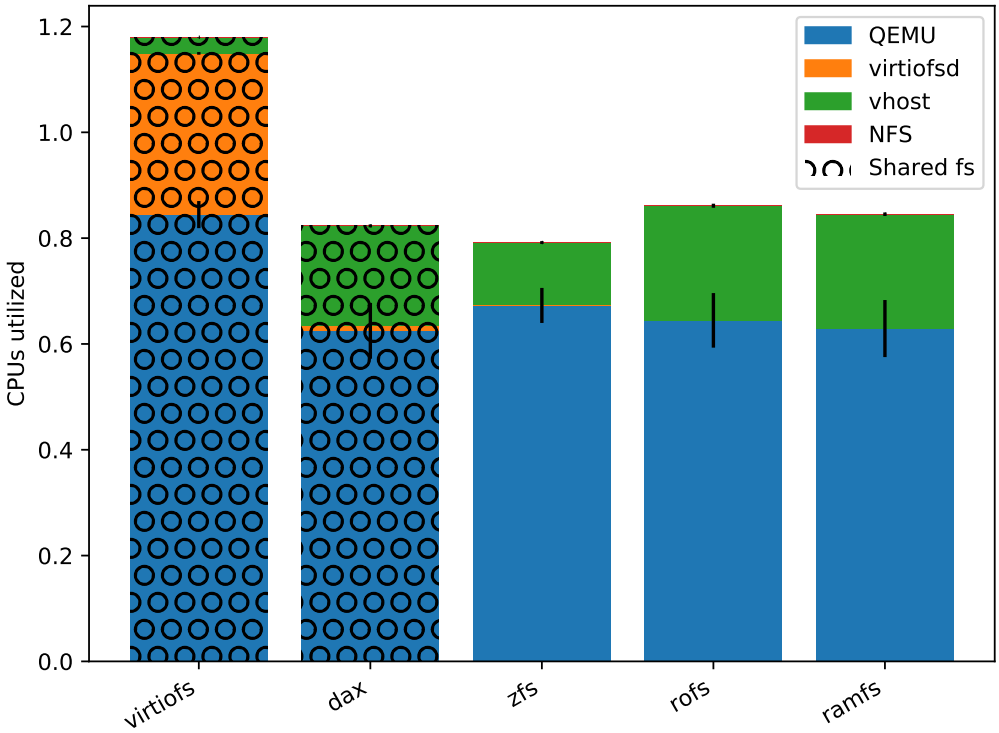
Όπως βλέπουμε στο σχήμα 4.6α', το virtio-fs με DAX window προσφέρει throughput (σε όρους εξυπηρετούμενων αιτημάτων ανά δευτερόλεπτο) που υπολείπεται αυτού των rofs και ramfs (τα οποία προηγούνται) κατά $\sim 20\%$. Δεδομένου ότι το CPU usage είναι οριακά χαμηλότερο από των άλλων δύο, αυτή η διαφορά χρήζει περαιτέρω μελλοντικής διερεύνησης (μέσω profiling), με εστίαση στην υλοποίηση του virtio-fs με DAX read datapath στο OSv, ως κύριο ύποπτο.

Όπως είναι αναμενόμενο σε αυτό το σενάριο χρήσης, το virtio-fs χωρίς DAX υστερεί με διαφορά έναντι όλων των υπόλοιπων συστημάτων αρχείων. Αυτό διότι είναι το μόνο που δεν χρησιμοποιεί οποιασδήποτε μορφής κρυφή μνήμη (cache), οπότε κάθε λειτουργία ανάγνωσης συνεπάγεται έξοδο και επεξεργασία στον host (όπως φαίνεται και από το υψηλό CPU usage), ενώ αυτές οι λειτουργίες είναι πολύ συχνές, πολλαπλασιάζοντας τις συνέπειες αυτού του overhead.

(α′) Requests ανά δευτερόλεπτο



(β′) CPU usage



Σχήμα 4.6: nginx HTTP load test

Κεφάλαιο 5

Συμπεράσματα και επεκτάσεις

Appendix A

FUSE copyright notice

Βιβλιογραφία

[1] Axboe, Jens και fio contributors: *Flexible I/O tester repository*. `https://github.com/axboe/fio`.

[2] Goyal, Vivek: *virtiofs-tests repository*. `https://github.com/rhvgoyal/virtiofs-tests`.

[3] Hajnoczi, Stefan: *QEMU Internals: vhost architecture*. `https://blog.vmsplice.net/2011/09/qemu-internals-vhost-architecture.html`, accessed 24/10/2020.

[4] in28minutes contributors: *spring-boot-examples repository*. `https://github.com/in28minutes/spring-boot-examples`.

[5] The Linux man-pages project: *mmap(2) Linux Programmer's Manual*, 5.08 έκδοση. `https://man7.org/linux/man-pages/man2/mmap.2.html`.

[6] OSDev contributors: *PCI*. `https://wiki.osdev.org/index.php?title=PCI&oldid=25084`, accessed 24/10/2020.

[7] OSv contributors: *Components of OSv*. `https://github.com/cloudius-systems/osv/wiki/Components-of-OSv`, accessed 24/10/2020.

[8] OSv contributors: *osv-apps repository*. `https://github.com/cloudius-systems/osv-apps`.

[9] OSv contributors: *OSv early boot (MBR)*. `https://github.com/cloudius-systems/osv/wiki/OSv-early-boot-(MBR)`, accessed 18/10/2020.

[10] OSv contributors: *OSv lzloader and early loader*. `https://github.com/cloudius-systems/osv/wiki/OSv-lzloader-and-early-loader`, accessed 24/10/2020.

[11] OSv contributors: *OSv repository*. `https://github.com/cloudius-systems/osv`.

[12] perf contributors: *Perf Wiki*. `https://perf.wiki.kernel.org`, accessed 18/10/2020.

[13] Senart, Tomás και vegeta contributors: *vegeta repository*. https://github.com/tsenart/vegeta.

[14] Sysoev, Igor και nginx contributors: *nginx*. http://nginx.org.

[15] Tsirkin, Michael S. και Cornelia Huck: *Virtual I/O Device (VIRTIO) working draft*. https://github.com/oasis-tcs/virtio-spec/commit/af6b93bfd9a0ea1b19147e5357d4434b5075b3c8.

[16] virtio-fs contributors: *virtio-fs linux repository*. https://gitlab.com/virtio-fs/linux.

[17] virtio-fs contributors: *virtio-fs qemu repository*. https://gitlab.com/virtio-fs/qemu.

[18] Wikipedia contributors: *PCI configuration space — Wikipedia, the free encyclopedia*. https://en.wikipedia.org/w/index.php?title=PCI_configuration_space&oldid=976450463, accessed 24/10/2020.