ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

Εικονικοποίηση Συσκευών Αποθήκευσης σε Περιβάλλοντα Υπολογιστικού Νέφους: μια Γρήγορη, Ασφαλής και Ευέλικτη Προσέγγιση με SPDK και virtio-vhost-user

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Νικόλαος Γ. Δράγαζης

Επιβλέπων Καθηγητής: Νεκτάριος Κοζύρης
Καθηγητής ΕΜΠ

Εργαστήριο Υπολογιστικών Συστημάτων

Αθήνα, Ιούλιος 2019

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

Εικονικοποίηση Συσκευών Αποθήκευσης σε Περιβάλλοντα Υπολογιστικού Νέφους: μια Γρήγορη, Ασφαλής και Ευέλικτη Προσέγγιση με SPDK και virtio-vhost-user

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Νικόλαος Γ. Δράγαζης

Επιβλέπων Καθηγητής: Νεκτάριος Κοζύρης
Καθηγητής ΕΜΠ

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 19η Ιουλίου 2019.

. . . . .	. . . . .	. . . . .
Νεκτάριος Κοζύρης	Νικόλαος Παπασπύρου	Γεώργιος Γκούμας
Καθηγητής ΕΜΠ	Καθηγητής ΕΜΠ	Επ. Καθηγητής ΕΜΠ

Εργαστήριο Υπολογιστικών Συστημάτων
Αθήνα, Ιούλιος 2019

National Technical University of Athens
School of Electrical and Computer Engineering
Division of Computer Science

Storage Virtualization in the Cloud: A Fast, Secure and Flexible Approach with SPDK and virtio-vhost-user

DIPLOMA THESIS

Nikolaos G. Dragazis

Computing Systems Laboratory
Athens, July 2019

. . . . . .

Νικόλαος Γ. Δράγαζης
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών ΕΜΠ

Περίληψη

Η παρούσα διπλωματική εργασία πραγματεύεται το πρόβλημα της εικονικοποίησης συσκευών αποθήκευσης σε περιβάλλοντα υπολογιστικού νέφους. Η εικονικοποίηση συσκευών αποθήκευσης αφορά στην υλοποίηση εικονικών συσκευών αποθήκευσης για εικονικές μηχανές. Αυτό το ζήτημα έχει μελετηθεί εκτενώς στο παρελθόν και έχουν προταθεί διάφορες υλοποιήσεις. Ωστόσο, η ραγδαία εξέλιξη της τεχνολογίας των μέσων αποθήκευσης έχει οδηγήσει σε μείωση των χρόνων καθυστέρησης (latency), καθιστώντας έτσι το συνολικό χρονικό διάστημα επεξεργασίας ενός αιτήματος I/O να είναι πρωτίστως υπολογιστικό κόστος αντί για κόστος επεξεργασίας στο μέσο αποθήκευσης. Αυτό το γεγονός μας ωθεί στο να στρέψουμε την προσοχή μας εκ νέου στο λογισμικό και να εργαστούμε για την περαιτέρω βελτίωσή του.

Σε αυτή τη διπλωματική παρουσιάζουμε μια νέα μέθοδο υλοποίησης εικονικών μέσων αποθήκευσης που λέγεται **"SPDK/VVU"**. Η βασική ιδέα είναι ο διαχωρισμός του μονοπατιού δεδομένων από τον επόπτη (hypervisor). Ο μηχανισμός μας αποτελείται από μια εικονική μηχανή που λειτουργεί σαν συσκευή αποθήκευσης και παρέχει υπηρεσίες αποθήκευσης σε άλλες τοπικές εικονικές μηχανές που πραγματοποιούν υπολογιστικές εργασίες. Όλη η κίνηση I/O δρομολογείται διαμέσου της συσκευής αποθήκευσης, αλλά χωρίς τη παρέμβαση του επόπτη στο μονοπάτι δεδομένων. Ο μηχανισμός επικοινωνίας των δύο εικονικών μηχανών βασίζεται σε μοιραζόμενη μνήμη. Αν ρυθμιστεί κατάλληλα, αυτή η λύση συνδυάζει υψηλό throughput, χαμηλή καθυστέρηση και κλιμακωσιμότητα. Είναι επίσης σχετικά ασφαλής υπό την έννοια ότι το λογισμικό προσομοίωσης της συσκευής τρέχει μέσα σε εικονική μηχανή. Τέλος, σε περιβάλλοντα υπολογιστικού νέφους, αυτή η λύση είναι ευέλικτη από της οπτική ματιά του χρήστη, διότι ο χρήστης έχει τον πλήρη έλεγχο του μονοπατιού δεδομένων. Στην ουσία, ο χρήστης μπορεί να προσαρμόζει δυναμικά το υλικό (hardware) των εικονικών του μηχανών. Αποκαλούμε αυτή την ιδιότητα ως "Αποθηκευτική Λειτουργικότητα οριζόμενη από το Χρήστη".

Η υλοποίηση του SPDK/VVU περιελάμβανε ενασχόληση με πολλαπλά έργα λογισμικού και αλληλεπίδραση με τις αντίστοιχες κοινότητες. Έχουμε υποβάλει αλλαγές στα έργα ανοικτού λογισμικού **SPDK**[1] [2] και **DPDK**[3]. Τα SPDK και DPDK συναποτελούν το λογισμικό προσομοίωσης που τρέχει μέσα στην εικονική μηχανή που αποτελεί τη συσκευή απόθήκευσης. Έχουμε επίσης υποβάλει αλλαγές στο **QEMU**[4] και στο **VIRTIO**[5] για μια νέα συσκευή παραεικονικοποίησης που λέγεται **"virtio-vhost-user"**, η οποία υλοποιεί το μηχανισμό επικοινωνίας των εικονικών μηχανών. Συνολικά, κάποιες αλλαγές μας έχουν συγχωνευτεί, ενώ κάποιες άλλες τελούν αυτή τη στιγμή υπό αξιολόγηση.

Λέξεις-Κλειδιά

εικονικοποίηση συσκευών αποθήκευσης, επιτάχυνση του μονοπατιού δεδομένων, υπερσελίδες, IOMMU, οδηγοί συσκευών στο χώρο χρήστη, DMA χωρίς αντίγραφα, DMA από συσκευή σε συσκευή, SPDK, vhost-user, μνήμη κατάλληλη για DMA, "κάρφωμα" σελίδων, PCI, MMIO, PIO, QEMU, KVM, ενεργός αναμονή, χωρίς κλειδώματα, eventfd, ioeventfd, irqfd, unix domain sockets

[1] `https://review.gerrithub.io/q/owner:+Dragazis+status:merged`
[2] `https://review.gerrithub.io/q/status:+open+owner:+Dragazis+repo:+spdk/spdk`
[3] `http://mails.dpdk.org/archives/dev/2019-June/135116.html`
[4] `https://lists.gnu.org/archive/html/qemu-devel/2019-04/msg03082.html`
[5] `https://lists.oasis-open.org/archives/virtio-dev/201906/msg00036.html`

Abstract

This diploma thesis deals with the problem of storage virtualization in cloud environments. Storage virtualization is the implementation of emulated storage devices for virtual machines. This concept has been extendedly studied in the past and various implementations have been proposed. However, the rapid evolution of the storage media technologies has led to significantly lower latencies, making the bulk of I/O processing time be software overhead rather than actual processing on the storage device. This forces us to turn our focus back to the software and prevent it from becoming a bottleneck.

In this diploma thesis, we are presenting a new storage virtualization solution called **"SPDK/VVU"**. The basic idea is to offload the I/O datapath from the hypervisor. Our setup consists of a Storage Appliance VM that offers storage services to local Compute VMs. All the I/O traffic goes through the Storage Appliance VM but without the intervention of the hypervisor in the datapath. The inter-VM communication mechanism is based on shared memory. If properly configured, this solution combines high throughput, low latency and scalability. It is also quite secure in the sense that the device emulation software runs inside the Storage Appliance VM. Last but not least, in cloud environments, this solution is flexible from a user's perspective, because the end user has full control of the datapath. Essentially, the user can adjust dynamically the underlying virtual hardware of his compute VMs. We call this property "User Defined Storage".

The implementation of SPDK/VVU required working on multiple projects and interacting with the corresponding communities. We have submitted patches to the **SPDK**[6] [7] and **DPDK**[8] open source projects. SPDK and DPDK make up the emulation software running inside the Storage Appliance VM. We have also submitted some patches on **QEMU**[9] and **VIRTIO**[10] on a new paravirtualized device called **"virtio-vhost-user"**, which implements the inter-VM communication mechanism. Overall, some of our patches have been merged while others are currently under review.

Keywords

storage virtualization, I/O datapath acceleration, hugepages, IOMMU, userspace drivers, zero-copy DMA, peer-to-peer DMA, SPDK, vhost-user, DMA-able memory, page pinning, PCI, MMIO, PIO, QEMU, KVM, polling, lock-free, eventfd, ioeventfd, irqfd, inter-VM communication, unix domain sockets

[6] https://review.gerrithub.io/q/owner:+Dragazis+status:merged
[7] https://review.gerrithub.io/q/status:+open+owner:+Dragazis+repo:+spdk/spdk
[8] http://mails.dpdk.org/archives/dev/2019-June/135116.html
[9] https://lists.gnu.org/archive/html/qemu-devel/2019-04/msg03082.html
[10] https://lists.oasis-open.org/archives/virtio-dev/201906/msg00036.html

Πρόλογος

Σε αυτό το σημείο, θα ήθελα να εκφράσω τη βαθιά μου ευγνωμοσύνη για την ανιδιοτελή συνεισφορά του συναδέλφου μου Βαγγέλη Κούκη, όχι μόνο στα πλαίσια αυτής της διπλωματικής εργασίας, αλλά και γενικότερα στην ατέρμονη προσπάθειά μου για την κατάκτηση της γνώσης. Επίσης, θα ήθελα να ευχαριστήσω τους καθηγητές μου και ιδιαίτερα τους κυρίους Κοζύρη, Παπασπύρου, Γκούμα και Φωτάκη για τις γνώσεις που μου μετέφεραν αλλά κυρίως για τον τρόπο σκέψης που μου μετέδωσαν στη γνωστική περιοχή της Επιστήμης των Υπολογιστών. Ειδική μνεία θα ήθελα να κάνω στον εκλιπόντα καθηγητή κύριο Μπάκα από τη ΣΕΜΦΕ για τον άρτιο τρόπο διδασκαλίας του, που θα αποτελεί σημείο αναφοράς στη μετέπειτα επαγγελματική μου καριέρα. Τέλος, θέλω να ευχαριστήσω την οικογένειά μου για όλη τη στήριξη και τους συμφοιτητές μου για τα όμορφα φοιτητικά χρόνια που περάσαμε μαζί στο Πολυτεχνείο.

Νίκος Δράγαζης
Ιούλιος 2019

Contents

List of figures

Εικονικοποίηση Συσκευών Αποθήκευσης με το Μηχανισμό SPDK/VVU

1 Εισαγωγή

1.1 Σκοπός & Κίνητρο

Στη παρούσα διπλωματική ασχολούμαστε με τη βελτίωση των μηχανισμών εικονικοποίησης συσκευών αποθήκευσης, ιδίως σε περιβάλλοντα υπολογιστικού νέφους (cloud). Το πρόβλημα αυτό είναι πολυδιάστατο. Μία από τις κατευθύνσεις στις οποίες κινούμαστε είναι η καλύτερη αξιοποίηση των δυνατοτήτων των σύγχρονων μέσων αποθήκευσης. Είναι γεγονός ότι τη σημερινή εποχή ολοένα και περισσότερες εφαρμογές που τρέχουν στο cloud χρειάζονται αποθηκευτική λειτουργικότητα υψηλής απόδοσης. Ωστόσο, λαμβάνοντας υπόψη τις υπάρχουσες προσεγγίσεις στο θέμα αυτό, φαίνεται ότι δεν έχει καταβληθεί αρκετή προσπάθεια προς της κατεύθυνση της βέλτιστης αξιοποίησης των σύγχρονων αποθηκευτικών μέσων, όπως οι NVMe[51] SSDs. Επίσης, μια εξίσου σημαντική παράμετρος στο πρόβλημα αυτό είναι η ασφάλεια. Στα περιβάλλοντα υπολογιστικού νέφους τα θέματα ασφαλείας είναι πιο σημαντικά, διότι έχουμε πολλούς χρήστες να χειρίζονται εικονικές μηχανές που τρέχουν στο ίδιο φυσικό μηχάνημα. Τέλος, μία πολύ σημαντική κατεύθυνση στην οποία κινούμαστε στη παρούσα διπλωματική είναι η παροχή περισσότερης ευελιξίας στον τελικό χρήστη. Στόχος είναι να μπορεί ο χρήστης να ρυθμίζει την αποθηκευτική λειτουργικότητα που παρέχει στις εικονικές του μηχανές. Αυτή τη στιγμή, στα δημόσια clouds που παρέχουν

υποδομές σαν υπηρεσία (Infrastructure as a Service - IaaS), ο χρήστης έχει ελάχιστη ευελιξία στο θέμα αυτό, καθώς ο πάροχος υπηρεσιών (Cloud Provider) έχει το πλήρη έλεγχο στο μονοπάτι δεδομένων.

Το κίνητρο για την ενασχόληση με εικονικές συσκευές αποθήκευσης στα πλαίσια της παρούσας διπλωματικης είναι διττό. Πρώτον, είναι γεγονός ότι το cloud εν γένει, αλλά και πιο ειδικά, η παροχή υποδομών σας υπηρεσία είναι μια ανερχόμενη τάση. Στο πλαίσιο αυτό αποκτά ιδιαίτερο ενδιαφέρον η βελτίωση των παρεχόμενων υπηρεσιών αποθήκευσης και η κάλυψη των αναγκών διαφόρων πελατών. Δεύτερον, τα τελευταία χρόνια παρατηρούμε μια σημαντική βελτίωση στην τεχνολογία, και συνεπώς και στη συνολική απόδοση, των μέσων αποθήκευσης. Έχουμε περάσει από την εποχή των μηχανικών δίσκων (HDDs) στην εποχή των δίσκων στερεάς κατάστασης (Solid State Drives - SSDs). Σήμερα, υπάρχουν πολύ αποδοτικές τεχνολογίες αποθήκευσης, όπως η τεχνολογία 3D crosspoint[1] και αντίστοιχα πολύ αποδοτικά πρωτόκολλα επικοινωνίας, όπως το NVMe. Αυτή η βελτίωση των μέσων αποθήκευσης καθιστά πιο δύσκολη της εικονικοποίησή τους με τρόπο που να αξιοποιεί πλήρως τις δυνατότητές τους.

1.2 Υπάρχουσες Προσεγγίσεις και SPDK/VVU

Αυτή τη στιγμή υπάρχουν διάφοροι μηχανισμοί εικονικοποίησης συσκευών αποθήκευσης και αντίστοιχα πολλά κριτήρια κατηγοριοποίησης. Θα μπορούσαμε να τους ταξινομήσουμε στις ακόλουθες κατηγορίες:

1. **Τεχνικές εικονικοποίησης**

    - **πλήρης εικονικοποίηση (trap and emulate)**:
      η συσκευή εικονικοποιείται από τον επόπτη (πχ. QEMU[16])
    - **παρα-εικονικοποίηση (VIRTIO[2])**:
      η συσκευή εξακολουθεί να υλοποιείται από τον επόπτη. Ωστόσο, η απόδοση I/O είναι καλύτερη, διότι οι drivers έχουν βελτιστοποιηθεί ώστε να μειώνεται το κόστος εικονικοποίησης.
    - **απευθείας ανάθεση (passthrough)**:
      η συσκευή είναι μια πραγματική PCI συσκευή που αναθέτεται εξ ολοκλήρου σε μία εικονική μηχανή. Η εικονική μηχανή έχει απευθείας πρόσβαση

στους πόρους της συσκευής (καταχωρητές, χώροι διευθύνσεων, διακοπές).

- **μεσολαβούμενη ανάθεση (mdev[4])**:

  η συσκευή παρουσιάζεται σαν πολλαπλές εικονικές συσκευές από το πυρήνα του host. Αυτή η τεχνική συνδυάζει τα πλεονεκτήματα της πλήρης εικονικοποίησης με αυτά της απευθείας ανάθεσης.

2. **Προσομοίωση του οπίσθιου μέρους της συσκευής (Μονοπάτι Δεδομένων)**

   - **Προσομοίωση στον επόπτη (QEMU)**:

     τα αιτήματα I/O προσομοιώνονται από τον επόπτη και υλοποιούνται σας κλήσεις συστήματος *read*/*write* σε μια εικόνα δίσκου (disk image), που είναι ένα αρχείο στο τοπικό σύστημα αρχείων του host.

   - **Προσομοίωση στο χώρο πυρήνα (vhost[5])**:

     το μονοπάτι δεδομένων αποσπάται από τον επόπτη και υλοποιείται από τον πυρήνα του host. Με αυτό τον τρόπο η συσκευή υλοποιείται εξ ολοκλήρου μέσα στον πυρήνα, χωρίς την παρέμβαση του επόπτη, γλυτώνοντας έτσι το κόστος από αλλαγές κατάστασης (context switches). Ο μηχανισμός vhost δουλεύει με μοιραζόμενη μνήμη.

   - **Προσομοίωση στο χώρο χρήστη (vhost-user[6])**:

     το μονοπάτι δεδομένων αποσπάται από τον επόπτη και υλοποιείται σε μια ξεχωριστή διεργασία στο χώρο χρήστη του host. Αυτός ο μηχανισμός είναι μια τροποποίηση του μηχανισμού vhost. Βασίζεται σε μοιραζόμενη μνήμη και μπορεί να επιτύχει καλύτερη απόδοση από τον πυρήνα αν παρακάμψουμε τον πυρήνα με drivers στο χώρο χρήστη (πχ. SPDK[72]).

3. **Πρωτόκολλο Αποθήκευσης**

   - **virtio-blk**:

     πρωτόκολλο ειδικά σχεδιασμένο για τεχνικές παραεικονικοποίησης. Παρακάμπτει το υποσύστημα SCSI του πυρήνα.

   - **SCSI**:

     πρωτόκολλο ευρείας υιοθέτησης για μεγάλο πλήθος συσκευών (πχ. συσκευές virtio-scsi)

- **NVMe**:

  πρωτόκολλο που αποδίδει καλύτερο απόδοση σε συνδυασμό με τα σύγχρονα μη-πτητικά μέσα, αξιοποιώντας τις ιδιότητές τους (χαμηλό latency, υψηλός βαθμός παραλληλισμού, υψηλό throughput)

Στα επόμενα κεφάλαια, θα μελετήσουμε μια νέα τεχνική εικονικοποίησης δίσκων που βασίζεται στο πρωτόκολλο vhost-user, στο λογισμικό SPDK και στη συσκευή virtio-vhost-user. Ονομάζουμε αυτό το νέο μηχανισμό **"SPDK/VVU"**. Η βασική ιδέα είναι να βασιστούμε στο μηχανισμό vhost-user, αλλά να απομονώσουμε το λογισμικό εικονικοποίησης μέσα σε μια ξεχωριστή εικονική μηχανή. Για την υλοποίηση του εν λόγω μηχανισμού, χρειάζεται να επεκτείνουμε τους υπάρχοντες μηχανισμούς επικοινωνίας, όπως αυτοί ορίζονται από το πρωτόκολλο vhost-user, ώστε το λογισμικό εικονικοποίησης να εξακολουθεί να έχει πρόσβαση στη μνήμη της εικονικής μηχανής. Αυτό πραγματοποιείται με τη βοήθεια της συσκευής virtio-vhost-user. Χρειάζεται επίσης να επεκτείνουμε τον κώδικα του SPDK ώστε να υποστηρίζει αυτό το νέο μηχανισμό.

Ο μηχανισμός SPDK/VVU συνδυάζει τρία βασικά πλεονεκτήματα. Αξιοποιεί την απόδοση του μηχανισμού vhost-user σε συνδυασμό με τους drivers χώρου χρήστη του SPDK. Είναι πιο ασφαλής μηχανισμός σε σχέση με το απλό πρωτόκολλο vhost-user, διότι το λογισμικό εικονικοποίησης τρέχει μέσα σε μία εικονική μηχανή και όχι απευθείας στο χώρο χρήστη του host. Τέλος, παρέχει έναν ευέλικτο μηχανισμό ελέγχου των εικονικών συσκευών αποθήκευσης στους τελικούς χρήστες, διότι το λογισμικό εικονικοποίησης "τρέχει" μέσα σε μια εικονική μηχανή, η οποία θα μπορούσε να ανήκει σε τελικούς χρήστες σε ένα περιβάλλον νέφους.

2 Θεωρητικό Υπόβαθρο

Σε αυτή την ενότητα παραθέτουμε όλες τις απαραίτητες γνώσεις για τη κατανόηση της παρούσης διπλωματικής.

2.1 QEMU/KVM

Το QEMU[16] (Quick Emulator) είναι ένα πρόγραμμα εικονικοποίησης ενός πλήρους υπολογιστικού συστήματος. Μπορεί να προσομοιώσει τον επεξεργαστή, τη μνήμη, το chipset και τις περιφερειακές συσκευές. Το QEMU μπορεί να χρησιμοποιηθεί για να εκτελέσουμε κώδικα μεταγλωττισμένο σε κάποια αρχιτεκτονική σε επεξεργαστή διαφορετικής αρχιτεκτονικής. Στην ειδική περίπτωση που θέλουμε να εκτελέσουμε κώδικα σε επεξεργαστή της ίδιας αρχιτεκτονικής (για παράδειγμα κώδικα x86 σε επεξεργαστή αρχιτεκτονικής x86), τότε μπορούμε να το κάνουμε αυτό με πιο αποδοτικό τρόπο, υπό την προϋπόθεση ότι ο επεξεργαστής έχει επεκτάσεις εικονικοποίησης (πχ. Intel VT-x[78], VT-d[55]). Για το σκοπό αυτό, το QEMU χρησιμοποιεί το module KVM του πυρήνα.

Το KVM[18] (Kernel Virtual Machine) είναι ένα module στο πυρήνα του Linux που αξιοποιεί τις επεκτάσεις εικονικοποίησης των σύγχρονων επεξεργαστών, με σκοπό να εκτελεί guest κώδικα απευθείας στο φυσικό επεξεργαστή με ασφαλή τρόπο. Οι σύχρονοι επεξεργαστές με επεκτάσεις εικονικοποίησης υποστηρίζουν ξεχωριστές καταστάσεις λειτουργίας. Στην αρχιτεκτονική x86 υπάρχουν 4 διαφορετικές καταστάσεις/επίπεδα λειτουργίας που είναι τα εξής:

- κατάσταση root - χώρος χρήστη
- κατάσταση root - χώρος πυρήνα
- κατάσταση non-root - χώρος χρήστη
- κατάσταση non-root - χώρος πυρήνα

Η κατάσταση root αντιστοιχεί σε host κώδικα ενώ η κατάσταση non-root αντιστοιχεί σε guest κώδικα. Κάποιες λειτουργίες και κάποιες εντολές προκαλούν trap και αλλαγή

κατάστασης όταν εκτελούνται σε κατάσταση non-root και με αυτό το τρόπο επιτυγχάνεται η εικονικοποίηση του υλικού. Το KVM εικονικοποιεί κάποια μέρη ενός συστήματος όπως την MMU και τους ελεγκτές διακοπών (interrupt controllers). Επίσης, χερίζεται τη κατάσταση του εικονικού επεξεργαστή (δομή Virtual Machine Control Structure ή VMCS εν συντομία) και παράγει εικονικές διακοπές. Ωστόσο, το KVM δεν μπορεί να προσομοιώσει ένα πλήρες υπολογιστικό σύστημα, υπό την έννοια ότι δεν μπορεί να προσομοιώσει περιφερειακές συσκευές. Γι' αυτο χρησιμοποιείται σε συνδυασμό με το QEMU. Η προγραμματιστική διεπαφή του KVM είναι το character device file */dev/kvm* και ένα σύνολο από κλήσεις συστήματος ioctls[20].

Με βάση αυτές τις πληροφορίες, η αλληλεπίδραση των QEMU και KVM γίνεται ως εξής[23]:

Πριν την εκκίνηση της εικονικής μηχανής, το QEMU απεικονίζει το BIOS (ή απευθείας το guest πυρήνα αν θέλουμε να παραλήψουμε το BIOS) στο φυσικό χώρο διευθύνσεων του εικονικού επεξεργαστή. Αυτό γίνεται με το *KVM_SET_USER_MEMORY_REGION* KVM ioctl. Στη συνέχεια, δημιουργεί ένα σύνολο από εικονικούς επεξεργαστές (vCPUs) με το *KVM_CREATE_VCPU* KVM ioctl. Κάθε εικονικός επεξεργαστής αντιστοιχεί σε ένα ξεχωριστό νήμα της διεργασίας QEMU. Προτού ξεκινήσει να τρέχει guest κώδικα, το QEMU ορίζει την αρχική κατάσταση των καταχωρητών των εικονικών επεξεργαστών με τα *KVM_GET_SREGS* και *KVM_SET_SREGS* KVM ioctls. Στη συνέχεια, το QEMU δίνει εντολή στο KVM να ξεκινήσει να τρέχει guest κώδικα με το ***KVM_RUN*** ioctl. Το KVM, σε απόκριση του αιτήματος αυτού, ξεκινάει να τρέχει guest κώδικα με την ειδική εντολή *VMXON* (αυτή η εντολή είναι ειδική για της αρχιτεκτονική x86, αλλά αντίστοιχες εντολές υπάρχουν και για τις άλλες αρχιτεκτονικές). Με αυτή την εντολή, ο επεξεργαστής αλλάζει κατάσταση από root σε non-root - εναλλακτικά λέμε ότι πραγματοποιεί *VMENTRY* - και ξεκινά να τρέχει guest κώδικα. Σε περίπτωση που ο guest πυρήνας επιχειρήσει να εκτελέσει μια προνομιούχα ενέργεια, ο επεξεργαστής θα κάνει trap, θα επανέλθει σε κατάσταση root και θα συνεχίσει να τρέχει κώδικα του KVM. Αυτή η αλλαγή κατάστασης από non-root σε root λέγεται "**world switch**". Το KVM ελέγχει το αίτιο της αλλαγής κατάστασης μέσω των καταχωρητών της δομής VMCS. Σε περίπτωση που το KVM μπορεί να εξυπηρετήσει το λόγο εξόδου της εικονικής μηχανής, το πράττει και πραγματοποιεί *VMENTRY*. Ένα τέτοιο γεγονός λέγεται "**ελαφρύ VMEXIT**". Στη περίπτωση όμως που το KVM δεν μπορεί να χεριστεί την έξοδο (διότι για παράδειγμα οφείλεται σε προσπέλαση

ενός καταχωρητή μιας εικονικής συσκευής ελεγχόμενης από το QEMU), επανέρχεται στο QEMU με return από το *ioctl(KVM_RUN)*. Αυτό το γεγονός λέγεται "**βαρύ VMEXIT**". Το QEMU χειρίζεται το γεγονός εξόδου και επαναλαμβάνει τη διαδικασία εκτέλεσης guest κώδικα, όπως αυτή περιγράφηκε παραπάνω.

Ανακεφαλαιώνοντας, η δομή του βρόχου στο κώδικα του QEMU που πραγματοποιεί τις ανωτέρω λειτουργίες μοιάζει ως εξής:

```
1  open("/dev/kvm")
2  ioctl(KVM_CREATE_VM)
3  ioctl(KVM_CREATE_VCPU)
4  for (;;) {
5       ioctl(KVM_RUN)
6       switch (exit_reason) {
7       case KVM_EXIT_IO:  /* ... */
8       case KVM_EXIT_HLT: /* ... */
9       }
10 }
```

2.2 Περιγραφείς Αρχείων για Συμβάντα - Event File Descriptors

Ένας περιγραφέας αρχείου για συμβάντα[24] (eventfd) είναι ένας μηχανισμός του πυρήνα για διενέργεια ειδοποιήσεων του πυρήνα προς διεργασίες στο χώρο χρήστη ή μεταξύ διεργασιών. Μπορεί να ελέγχεται για ειδοποιήσεις μαζί άλλους περιγραφείς αρχείων με τις συναρτήσεις *select()*, *poll()* και *epoll()*.

2.3 ioeventfd/irqfd

Οι μηχανισμοί ioeventfd και irqfd παρέχονται από το KVM με σκοπό την παραγωγή πιο "ελαφρών" *VMEXITs*. Είναι δύο αποδοτικοί μηχανισμοί για ειδοποιήσεις του guest προς τον host και του host προς τον guest αντίστοιχα.

Ο μηχανισμός ioeventfd αντιστοιχίζει διευθύνσεις I/O σε eventfds. Οποτεδήποτε ο guest προσπελάζει μια τέτοια διεύθυνση I/O, το KVM "βαράει" τον αντίστοιχο eventfd.

Ο μηχανισμός irqfd είναι το ακριβώς αντίθετο. Αντιστοιχίζει eventfds σε διακοπές. Οποτεδήποτε κάποιος "βαράει" έναν τέτοιο eventfd, το KVM στέλνει την αντίστοιχη διακοπή στην εικονική μηχανή.

Η προγραμματιστική διεπαφή για τη χρήση αυτών των μηχανισμών είναι τα ioctls τύπου *KVM_IOEVENTFD* και *KVM_IRQFD*, που αποτελούν μέρος της διεπαφής του KVM[20].

2.4 VFIO

Το vfio[3] είναι ένας driver στο πυρήνα του Linux που καθιστά εφικτή τη δημιουργία οδηγών συσκευών στο χώρο χρήστη. Στην ουσία εξάγει όλους τους πόρους των συσκευών PCI σε διεργασίες στο χώρο χρήστη. Οι πόροι μιας συσκευής PCI είναι οι καταχωρητές του PCI Configuration Space, οι χώροι διευθύνσεων MMIO και PIO και οι διακοπές (interrupts). Το vfio επίσης χρησιμοποιεί την IOMMU του συστήματος, σε περίπτωση που υπάρχει, για να παρέχει τη δυνατότητα για DMA με ασφαλή τρόπο από μνήμη στο χώρο χρήστη. Η προγραμματιστική διεπαφή του vfio αποτελείται από character device files στο κατάλογο */dev/vfio/* και ένα σύνολο από κλήσεις συστήματος ioctls. Οι πόροι κάθε συσκευής απεικονίζονται σε συγκεκριμένα τμήματα ενός character device file. Μια διεργασία έχει πρόσβαση σε αυτούς κάνοντας *read()*, *write()* ή *mmap()* στα συγκεκριμένα τμήματα του αρχείου. Ο πυρήνας ειδοποιεί το χώρο χρήστη για διακοπές από τη συσκευή με τη χρήση ενός eventfd. Το vfio υποστηρίζει και τους δύο μηχανισμούς του διαύλου PCI για διακοπές, δηλαδή legacy και MSI διακοπές.

Σε περίπτωση που μια διεργασία θέλει να ζητήσει DMA από μνήμη χώρου χρήστη, χρησιμοποιεί το *VFIO_IOMMU_MAP_DMA* ioctl για να δηλώσει τη συγκεκριμένη περιοχή μνήμης. Ο vfio driver αναλαμβάνει να κάνει τη συγκεκριμένη περιοχή μνήμης κατάλληλη για DMA. Με αυτό το ioctl, μια διεργασία δίνει σαν παραμέτρους στο πυρήνα ένα συνεχόμενο εύρος εικονικών διευθύνσεων που αντιστοιχούν στη μνήμη για DMA και μια διεύθυνση IOVA (IO Virtual Address). Ο vfio driver εξυπηρετεί αυτό το ioctl με τις ακόλουθες ενέργειες:

- "καρφώνει" τη συγκεκριμένη περιοχή μνήμης, ώστε να είναι κατάλληλη για DMA. Αυτό σημαίνει ότι τα αντίστοιχα πλαίσια μνήμης δε θα γίνουν swap-out

από το πυρήνα και δε θα αλλάξει η απεικόνιση VA-to-PA (Virtual Address to Physical Address).

- προγραμματίζει την IOMMU, χρησιμοποιώντας τη διεύθυνση IOVA που έδωσε η διεργασία. Συγκεκριμένα, εισάγει τις κατάλληλες μεταφράσεις στο IOVA domain της συσκευής, ώστε μια εντολή DMA από το εύρος των IOVA διευθύνσεων να μεταφράζεται από την IOMMU σε DMA από τα αντίστοιχα φυσικά πλαίσια μνήμης.

Το vfio διαθέτει επίσης τη δυνατότητα να δουλεύει χωρίς IOMMU, παρακάμπτοντάς την, αν αυτή είναι ενεργοποιημένη στο σύστημα. Λέμε ότι το vfio είναι σε no-IOMMU mode. Σε αυτή τη περίπτωση, δεν υποστηρίζεται το *VFIO_IOMMU_MAP_DMA* ioctl και η μόνη λύση για DMA από μνήμη χώρου χρήστη είναι η χρήση υπερσελίδων.

2.5 Υπερσελίδες (Hugepages)

Οι σύγχρονοι επεξεργαστές υποστηρίζουν πολλαπλά μεγέθη σελίδων. Οι υπερσελίδες είναι συνεχόμενες περιοχές φυσικής μνήμης που είναι μεγαλύτερες από τις κλασσικές σελίδες των 4KB. Το μέγεθος των υπερσελίδων εξαρτάται από την αρχιτεκτονική. Για παράδειγμα, η αρχιτεκτονική x86 υποστηρίζει μεγέθη υπερσελίδων 2MB και 1GB.

Οι υπερσελίδες έχουν τρία δομικά χαρακτηριστικά[27]:

- είναι συνεχόμενες περιοχές φυσικής μνήμης. Αυτό επιτρέπει να έχουμε μία και μοναδική απεικόνιση για 2MB διευθύνσεις στο πίνακα σελίδων μιας διεργασίας αντί για 512 απεικονίσεις των 4KB. Απόρια αυτού είναι ότι με μία καταχώρηση στην TLB καλύπτουμε μεγαλύτερο εύρος μνήμης, μειώνοντας έτσι τον αριθμό των TLB misses.

- δε γίνονται ποτέ swap-out από το πυρήνα.

- η απεικόνιση VA-to-PA είναι σταθερή. Αυτό σημαίνει ότι ο πυρήνας δε μετακινεί τη φυσική τοποθεσία της υπερσελίδας στη μνήμη.

Οι υπερσελίδες έχουν τρία βασικά πλεονεκτήματα:

- το γεγονός ότι αντιστοιχούν σε συνεχόμενη φυσική μνήμη επιτρέπει να έχουμε λιγότερα TLB misses, όπως αναφέραμε παραπάνω.

- το κόστος για κάθε TMB miss είναι μικρότερο σε σχέση με τις κανονικές σελίδες, διότι απαιτείται η διάσχιση λιγότερων επιπέδων στο πίνακα σελίδων (οι σύγχρονοι επεξεργαστές χρησιμοποιούν πίνακες σελίδων πολλαπλών επιπέδων). Με προϋπόθεση ότι η κλίμακα (granularity) του τελευταίου επιπέδου είναι 4KB, για υπερσελίδες 2MB γλυτώνουμε ένα επίπεδο, ενώ για υπερσελίδες 1GB γλυτώνουμε 2 επίπεδα.

- η απεικόνιση VA-to-PA για κάθε υπερσελίδα είναι σταθερή. Με άλλα λόγια, η φυσική διεύθυνση μιας υπερσελίδας δεν αλλάζει. Η ιδιότητα αυτή είναι γνωστή ως "κάρφωμα σελίδας (page pinning)". Το γεγονός αυτό τις καθιστά κατάλληλες για λειτουργίες DMA. Αυτή τη στιγμή, οι υπερσελίδες είναι ο μοναδικός μηχανισμός για DMA από μνήμη χώρου χρήστη χωρίς τη παρεμβολή του πυρήνα. Γι' αυτή την ιδιότητα χρησιμοποιούνται από το SPDK συνδυαστικά με τον uio driver. Ο uio driver χρησιμοποιείται για δημιουργία drivers στο χώρο χρήστη, αλλά, εν αντιθέσει με τον vfio driver, δεν υποστηρίζει λειτουργίες DMA.

2.6 VIRTIO

Το VIRTIO[2] είναι ένα πρότυπο για δημιουργία συσκευών παραεικονικοποίησης. Ορίζει τις προδιαγραφές των συσκευών καθώς και των drivers για τον έλεγχο αυτών των συσκευών. Σκοπός του VIRTIO είναι ο ορισμός ενός μηχανισμού αλληλεπίδρασης ανάμεσα στο driver και στον επόπτη που υλοποιεί τη συσκευή, ο οποίος θα ελαχιστοποιεί το κόστος εικονικοποίησης. Στην ουσία, αυτό σημαίνει ελαχιστοποίηση των VMEXITs που προκαλούνται από λειτουργίες PIO ή MMIO.

Το πρότυπο ορίζει ότι οι drivers χωρίζονται σε δύο μέρη. Υπάρχει ο frontend driver που τρέχει στο guest πυρήνα, και ο backend driver που είναι μέρος του επόπτη. Η επικοινωνία ανάμεσα στον frontend και τον backend driver γίνεται με ειδικές δομές δεδομένων που λέγονται "virtqueues". Οι virtqueues είναι κυκλικές ουρές αποθηκευμένες στη μνήμη του guest και οι οποίες δημιουργούνται από τον frontend driver. Οι virtqueues δε χρησιμοποιούνται για τη μεταφορά δεδομένων αλλά για τη μεταφορά δεικτών στα δεδομένα. Οι μηχανισμοί ειδοποιήσεων ανάμεσα στον frontend και τον

backend driver αξαρτώνται από το τύπο του διαύλου. Το VIRTIO υποστηρίζει τρεις τύπους διαύλων: PCI, MMIO, CCW.

2.7 IOMMU

Η IOMMU είναι ένα τμήμα υλικού που παρεμβάλλεται ανάμεσα στις περιφερειακές συσκευές και τη φυσική μνήμη. Η λειτουργία του είναι παρόμοια με αυτή της MMU στον επεξεργαστή, αλλά αφορά συσκευές αντί για διεργασίες. Συγκεκριμένα, μεταφράζει τις διευθύνσεις των λειτουργιών DMA που πραγματοποιούνται από τις συσκευές. Ο μηχανισμός αυτός είναι διαφανής για τις συσκευές. Κάθε συσκευή αντιλαμβάνεται έναν εικονικό χώρο διευθύνσεων, το **"IOVA domain"**. Το λογισμικό είναι υπεύθυνο για το προγραμματισμό της IOMMU με τον ορισμό των μεταφράσεων IOVA-to-PA.

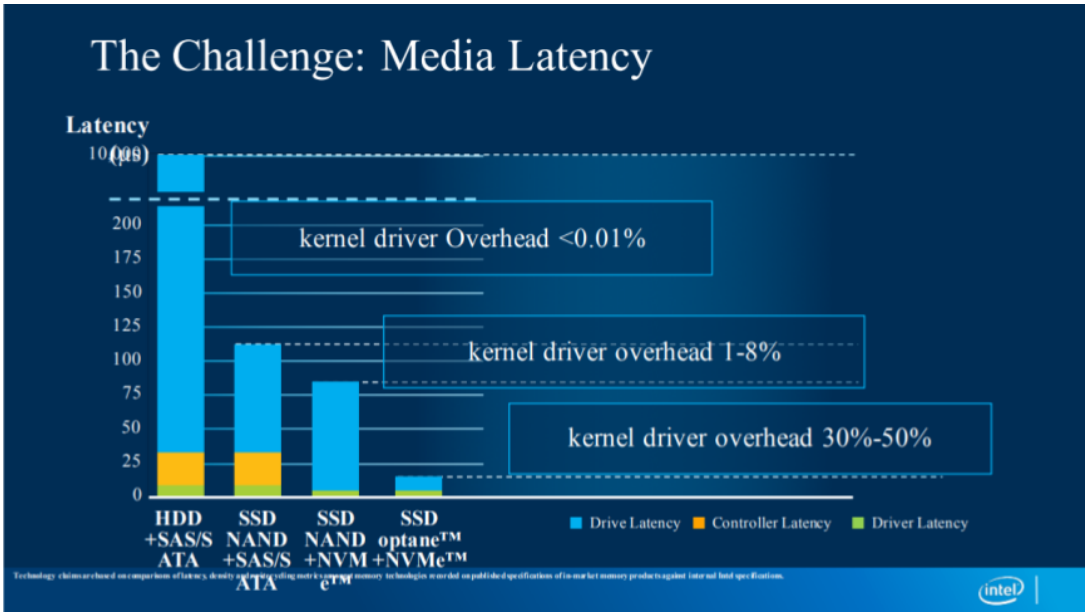
Η Intel ενσωματώνει την IOMMU στο chipset, δηλαδή στο υλικό που συνδέει τον επεξεργαστή, τη μνήμη και τις περιφερειακές συσκευές. Η Intel χρησιμοποιεί των όρο "DMA Remapping (DMAR)" για την IOMMU και την εντάσσει ως τμήμα των επεκτάσεων εικονικοποίησης Intel VT-d[55].

Η IOMMU έχει δύο πλεονεκτήματα. Πρώτον, εισάγει ένα επίπεδο ασφαλείας ανάμεσα στις συσκευές και τη φυσική μνήμη. Με άλλα λόγια, προστατεύει τη φυσική μνήμη από επιθέσεις DMA. Οι συσκευές έχουν πρόσβαση σε ένα περιορισμένο κομμάτι της φυσικής μνήμης. Δεύτερον, καθιστά δυνατή την απευθείας ανάθεση (passthrough) συσκευών σε εικονικές μηχανές. Το πρόβλημα που ανακύπτει σε περιπτώσεις απευθείας ανάθεσης είναι ότι οι drivers στο guest πυρήνα αντιλαμβάνονται τον guest χώρο διευθύνσεων, ενώ οι φυσικές συσκευές αντιλαμβάνονται τον host χώρο διευθύνσεων. Η απευθείας αλληλεπίδρασή τους καθίσταται δυνατή με την είσοδο της IOMMU, η οποία διαφανώς κάνει την ανάλογη μετάφραση διευθύνσεων. Ο προγραμματισμός της IOMMU με τις απεικονίσεις GPA-to-HPA γίνεται από τον επόπτη.[14]

3 Εισαγωγή στο SPDK

Το SPDK (Storage Performance Development Kit) [62] είναι ένα σύνολο βιβλιοθηκών και οδηγών συσκευών (drivers) για συσκευές αποθήκευσης στο χώρο χρήστη. Είναι ένα έργο ανοικτού κώδικα της Intel. Είναι εμπνευσμένο από το προϋπάρχον και παρεμφερές έργο DPDK (Data Plane Develompent Kit) [63] που αφορά κάρτες δικτύου αντί για συσκευές αποθήκευσης. Το SPDK δουλεύει αποκλειστικά σε χώρο χρήστη παρακάμπτοντας πλήρως το πυρήνα.

Σκοπός του SPDK είναι η παροχή ενός αποδοτικού, γρήγορου και κλιμακώσιμου λογισμικού ελέγχου των σύγχρονων συσκευών αποθήκευσης. Τα σύγχρονα μέσα αποθήκευσης και συγκεκριμένα οι διάφορες τεχνολογίες συσκευών NVMe παρουσιάζουν χαμηλό latency, υψηλό throughput και έχουν υψηλό βαθμό παραλληλισμού. Το SPDK, εν αντιθέσει με τον γενικού σκοπού πυρήνα, έχει σχεδιαστεί ώστε να επιτυγχάνει τη μέγιστη δυνατή αξιοποίηση των δυνατοτήτων των σύγχρονων μέσων αποθήκευσης. Η επανασχεδίαση του λογισμικού ελέγχου των συσκευών αποθήκευσης αποδεικνύεται κρίσιμη για τη συνολική απόδοση του συστήματος, καθώς τα σύγχρονα μέσα αποθήκευσης επιτυγχάνουν πολύ χαμηλό latency, συγκρίσιμο με αυτό του λογισμικού. Το διάγραμμα 1 δίνει μια αίσθηση επ' αυτού.



Σχήμα 1: *Καθυστερήσεις Συσκευών Αποθήκευσης για Αιτήματα I/O*

Η λειτουργία του SPDK διακρίνεται από τα ακόλουθα τρία βασικά χαρακτηριστικά:

1. το SPDK δουλεύει αποκλειστικά στο χώρο χρήστη[66]. Αυτό σημαίνει ότι παρακάμπτει τη στοίβα αποθήκευσης του πυρήνα. Το SPDK είναι εφοδιασμένο με οδηγούς συσκευών για διάφορους τύπους συσκευών αποθήκευσης. Οι οδηγοί συσκευών έχουν πρόσβαση στους πόρους των συσκευών μέσω δύο μηχανισμών του πυρήνα: το uio και το vfio. Το SPDK παρακάμπτει το πυρήνα για δύο λόγους:

   (a) με αυτό το τρόπο αποφεύγεται το κόστος από τα context switches

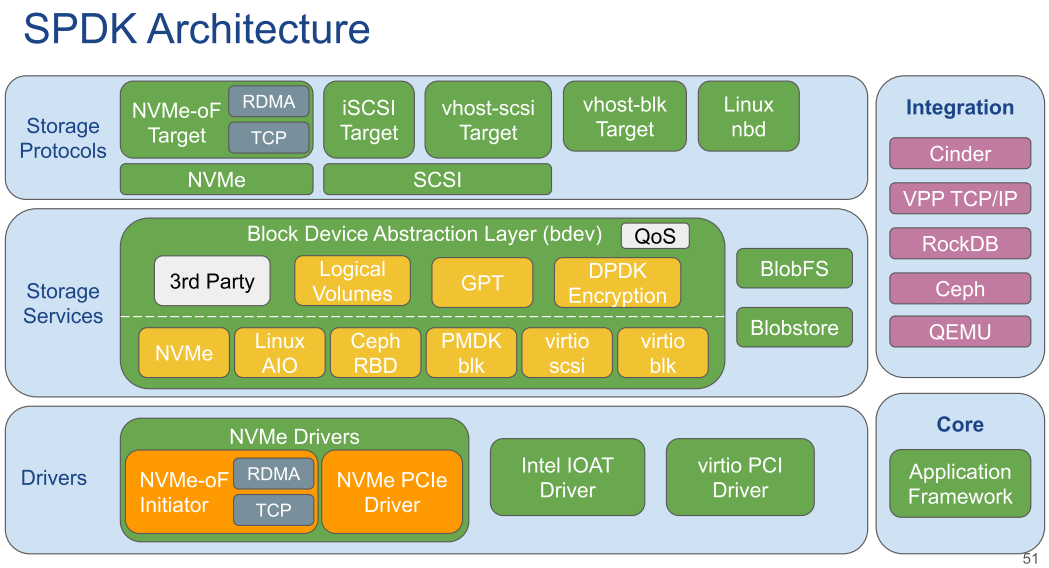
   (b) η στοίβα αποθήκευσης του πυρήνα έχει σχεδιαστεί για να έχει ικανοποιητική απόδοση για κάθε τύπο συσκευής αποθήκευσης, μη επιτυγχάνοντας τη βέλτιση δυνατή απόδοση για κάποιο συγκεκριμένο τύπο συσκευής αποθήκευσης

2. το SPDK δε χρησιμοποιεί κλειδώματα στο μονοπάτι δεδομένων. Αντιθέτως, χρησιμοποιεί ανταλλαγή μηνυμάτων [67] όπου υπάρχει ανάγκη για συγχρονισμό. Ο λόγος είναι ότι τα κλειδώματα αποδεδειγμένα δε κλιμακώνουν λόγω των cache invalidations που προκαλούνται από το πρωτόκολλο για cache coherence.

3. το SPDK χρησιμοποιεί ενεργό αναμονή (polling) αντί για διακοπές (interrupts). Αποδεικνύεται ότι αυτή είναι η καλύτερη επιλογή για τις σύγχρονες συσκευές αποθήκευσης. Αυτό συμβαίνει για δύο λόγους. Ο πρώτος λόγος είναι ότι για την εξυπηρέτηση μιας διακοπής απαιτείται η συμμετοχή του πυρήνα. Στη περίπτωση του vfio για παράδειγμα, ο πυρήνας πρέπει να ειδοποιήσει τη διεργασία χώρου χρήστη μέσω ενός eventfd για τη λήψη μιας διακοπής. Η συμμετοχή του πυρήνα συνεπάγεται context switches και υπολογιστικό κόστος από την εκτέλεση της συνάρτησης χειρισμού της διακοπής. Ο δεύτερος λόγος είναι ότι το κόστος ανάκτησης μιας διεργασίας που είναι σε κατάσταση ύπνου περιμένοντας μια διακοπή ή, ακόμη χειρότερα, το κόστος αλλαγής της κατάστασης ενός επεξεργαστή που βρίσκεται σε κατάσταση ύπνου περιμένοντας μια διακοπή είναι ασύμφορο στη περίπτωση των σύγχρονων συσκευών αποθήκευσης. Οι σύγχρονες συσκευές αποθήκευσης έχουν χαμηλό και σχετικά ντετερμινιστικό latency, γεγονός που καθιστά την ενεργό αναμονή συμφέρουσα στη περίπτωση του SPDK, όπου αναμένεται να έχουμε πολλά αιτήματα I/O.

Η αρχιτεκτονική του SPDK μπορεί να χωριστεί σε τρία δομικά επίπεδα[64]. Το ακό-

λουθο σχήμα παρουσιάζει την αρχιτεκτονική δομή του SPDK και όλα τα δομικά στοιχεία που απαρτίζουν το έργο.



Σχήμα 2: *Αρχιτεκτονική του SPDK*

Στο κατώτερο επίπεδο υπάρχουν οι drivers των συσκευών (NVMe, I/OAT, virtio-pci). Στο ενδιάμεσο επίπεδο βρίσκεται το Block Layer. Το Block Layer απαρτίζεται από τα bdev modules. Τα bdev modules έχουν αντίστοιχη λειτουργικότητα με τους block device drivers στο πυρήνα. Συγκεκριμένα, υλοποιούν μια γενική διεπαφή για I/O από συσκευές και εξάγουν συσκευές τύπου block προς το ανώτερο επίπεδο. Το Block Layer διαθέτει επίσης bdev modules που υλοποιούν εικονικές συσκευές block πάνω από άλλες συσκευές block. Ο σκοπός τους είναι η παροχή υπηρεσιών όπως συμπίεση, κρυπτογράφηση, λογικοί τόμοι (logical volumes), κοκ. Τέλος, στο ανώτερο επίπεδο υπάρχουν βιβλιοθήκες που υλοποιούν τερματικά (targets) για διάφορα πρωτόκολλα αποθήκευσης. Το SPDK διαθέτει NVMf target, iSCSI target και vhost target.

4 Εισαγωγή στο πρωτόκολλο Vhost

4.1 Γενική Περιγραφή

Το vhost είναι ένας μηχανισμός για να μπορούμε να υλοποιούμε το μονοπάτι δεδομένων για συσκευές I/O εκτός του επόπτη (hypervisor). Είναι ανεξάρτητο του τύπου της συσκευής, αλλά αφορά εικονικές συσκευές που υπακούουν στο πρότυπο Virtio. Ο μηχανισμός είναι απόλυτα διαφανής για τον οδηγό της συσκευής, δηλαδή δεν χρειάζεται κάποιου είδους τροποποίηση του οδηγού. Ο σκοπός του μηχανισμού είναι να μπορούμε να υλοποιήσουμε πιο αποδοτικά μονοπάτια δεδομένων από αυτά του επόπτη.

Υπάρχουν δύο παραλλαγές του μηχανισμού. Υπάρχει η υλοποίηση στο χώρο πυρήνα και η υλοποίηση στο χώρο χρήστη. Αποτελεί σύμβαση η πρώτη να λέγεται "vhost" και η δεύτερη "vhost-user", αν και στη βιβλιογραφία η σύμβαση αυτή δεν τηρείται κατά κανόνα. Ιστορικά, η υλοποίηση στο χώρο πυρήνα προϋπήρξε της υλοποίησης στο χώρο χρήστη. Για την ακρίβεια, η υλοποίηση στο χώρο χρήστη βασίστηκε στην υλοποίηση στο χώρο πυρήνα. Η ειδοποιός διαφορά ανάμεσα στις δύο παραλλαγές είναι ο μηχανισμός επικοινωνίας με τον επόπτη για την εγκαθίδρυση του μονοπατιού δεδομένων. Ο μηχανισμός vhost χρησιμοποιεί character device files και τη κλήση συστήματος *ioctl()*. Ο μηχανισμός vhost-user χρησιμοποιεί unix domain sockets και ανταλλαγή μηνυμάτων πάνω από το socket. Για το μηχανισμό vhost-user υπάρχει ένα ανεπίσημο πρωτόκολλο στα έγγραφα του QEMU[6].

Στη παρούσα διπλωματική χρησιμοποιούμε την υλοποίηση σε χώρο χρήστη, γι' αυτό και αναλύουμε το συγκεκριμένο μηχανισμό στην ακόλουθη υποενότητα.

4.2 Τρόπος Λειτουργίας του πρωτοκόλλου vhost-user

Το πρωτόκολλο vhost-user ορίζει δύο πλευρές στη διαδιεργασιακή επικοινωνία: τον **master** και τον **slave**. Εναλλακτικά, χρησιμοποιείται και η ορολογία **initiator** και **target**. Ο master είναι η διεργασία που χρησιμοποιεί την εικονική συσκευή, δηλαδή ο επόπτης. Ο slave είναι η διεργασία που υλοποιεί την εικονική συσκευή. Ο μηχανισμός χωρίζεται σε δύο μέρη: το **μονοπάτι ελέγχου** και το **μονοπάτι δεδομένων**.

Το μονοπάτι ελέγχου είναι ο μηχανισμός για της εγκαθίδρυση του μονοπατιού δεδομένων. Το πρωτόκολλο vhost-user ορίζει ότι αυτό συμβαίνει μέσα από μία σειρά από μηνύματα ελέγχου. Τα μηνύματα ανταλλάσσονται μεταξύ των master και slave μέσω ενός unix domain socket. Τα περισσότερα μηνύματα παράγονται από τον master. Ωστόσο, υπάρχουν περιπτώσεις μηνυμάτων που παράγονται από τον slave ή είναι απάντηση σε μηνύματα του master. Ο master μπορεί να λειτουργεί είτε σαν πελάτης (client) είτε σαν εξυπηρετητής (server) στην επικοινωνία μέσω των sockets. Συνήθως, ο slave λειτουργεί σαν εξυπηρετητής. Η ανταλλαγή μηνυμάτων πυροδοτείται από τη σύνδεση ενός πελάτη στο socket. Η ανταλλαγή μηνυμάτων γίνεται ως εξής:

- Ο master στέλνει ένα μήνυμα τύπου *VHOST_USER_GET_FEATURES*, ζητώντας από τον slave να του κοινοποιήσει τη λίστα με τα features που υποστηρίζει.
- Ο slave στέλνει ως απάντηση τα features που υποστηρίζει.
- Ο master συγκρίνει τα features του slave με τα features που υποστηρίζει ο guest οδηγός συσκευής. Στη συνέχεια, διαλέγει τη τομή τους και ενεργοποιεί αυτά τα features στον slave με ένα μήνυμα τύπου *VHOST_USER_SET_FEATURES*.
- Αν ο slave υποστηρίζει το feature *VHOST_USER_F_PROTOCOL_FEATURES*, αυτό σημαίνει ότι ο slave υποστηρίζει κάποια features που είναι εξειδικευμένα για το μηχανισμό vhost-user. Σε αυτή τη περίπτωση ο master μαθαίνει αυτά τα features από τον slave με ένα μήνυμα τύπου *VHOST_USER_GET_PROTOCOL_FEATURES* και, όπως προηγουμένως, ενεργοποιεί τη τομή τους με ένα μήνυμα τύπου *VHOST_USER_SET_PROTOCOL_FEATURES*.
- Στη συνέχεια, ο master δημοσιοποιεί στον slave τη μνήμη της εικονικής μηχανής master. Η μνήμη του master πρέπει να είναι μοιραζόμενη. Γι' αυτό συνήθως είναι ένα αρχείο στο tmpfs ή στο hugetlbfs. Επίσης, η μνήμη του master μπορεί να είναι κατακερματισμένη σε πολλαπλά μη-συνεχόμενα τμήματα. Το πρωτόκολλο vhost-user ορίζει ως περιορισμό για το μέγιστο πλήθος τους τον αριθμό 8. Ο master στέλνει όλη τη πληροφορία για όλα τα τμήματα μνήμης με ένα μήνυμα τύπου *VHOST_USER_SET_MEM_TABLE*. Με το μήνυμα αυτό, ο master περνά στον slave δικαιώματα για τα ανοικτά αρχεία που αντιστοιχούν στη μνήμη του ώστε ο slave, εν τέλει, να έχει πρόσβαση σε όλη τη μνήμη του master. Επίσης, για κάθε τμήμα μνήμης στέλνει και τις ακόλουθες πληροφορίες:

  - host εικονική διεύθυνση - αντιστοιχεί στην εικονική διεύθυνση στον εικονικό χώρο διευθύνσεων του επόπτη όπου είναι απεικονισμένο το συγκεκριμένο κομμάτι μνήμης του master
  - guest φυσική διεύθυνση - αντιστοιχεί στη φυσική διεύθυνση στο φυσικό χώρο διευθύνσεων της guest CPU της εικονικής μηχανής master όπου είναι απεικονισμένο το συγκεκριμένο κομμάτι μνήμης του master
  - offset - offset όπου ο slave πρέπει να κάνει *mmap()* το εν λόγω ανοικτό αρχείο

  Ο slave κάνει *mmap()* κάθε ανοικτό αρχείο, δηλαδή κάθε κομμάτι μνήμης του master, όπως αυτά περιγράφονται στο μήνυμα *VHOST_USER_SET_MEM_TABLE*.

- Αφότου ο slave αποκτήσει πρόσβαση στη μνήμη της εικονικής μηχανής master, ο master στέλνει πληροφορίες σχετικά με τις virtqueues του guest οδηγού συσκευής. Αυτό είναι απαραίτητο ώστε ο slave να μπορεί να εντοπίσει τις virtqueues στη μνήμη του master. Για κάθε virtqueue, ο master στέλνει τις εξής πληροφορίες:

  - το μέγεθός της με το μήνυμα *VHOST_USER_SET_VRING_NUM*
  - το δείκτη στην αρχή της ουράς με το μήνυμα *VHOST_USER_SET_VRING_BASE*
  - τη guest φυσική διεύθυνση όπου είναι αποθηκευμένη η virtqueue στη μνήμη του master με το μήνυμα *VHOST_USER_SET_VRING_ADDR*

- Τέλος, για κάθε virtqueue, ο master στέλνει δύο eventfds, έναν kickfd και έναν callfd όπως χαρακτηριστικά λέγονται, με τα μηνύματα *VHOST_USER_SET_VRING_KICK* και *VHOST_USER_SET_VRING_CALL* αντίστοιχα. Αυτά χρησιμοποιούνται για τις ειδοποιήσεις ανάμεσα στον master και τον slave.

Το μονοπάτι δεδομένων αφορά στους μηχανισμούς για την μεταφορά των δεδομένων από και προς τη μνήμη της εικονικής μηχανής. Ο slave έχει πλήρη πρόσβαση στη μνήμη του master και, επομένως, έχει απευθείας πρόσβαση στις virtqueues και στα δεδομένα, όπου κι αν αυτά βρίσκονται μέσα στη μνήμη του master. Επομένως, η

μεταφορά των δεδομένων γίνεται με άμεσο τρόπο, χωρίς αντίγραφα εντός της μνήμης του master, όπως ακριβώς δηλαδή δουλεύει και ο επόπτης για την εξυπηρέτηση αιτημάτων I/O. Οι ειδοποιήσεις μεταξύ των master και slave για την υποβολή νέων αιτημάτων I/O και για την ολοκλήρωση επεξεργασίας αιτημάτων I/O γίνεται με τη χρήση των kickfds και callfds αντίστοιχα.

5 Σχεδιασμός

5.1 Γενική Περιγραφή της Σχεδίασης

Σε αυτό το κεφάλαιο θα περιγράψουμε το σχεδιασμό του μηχανισμού εικονικοποίησης συσκευών αποθήκευσης **"SPDK/VVU"**. Ο μηχανισμός SPDK/VVU βασίζεται στον vhost target του SPDK και στη συσκευή virtio-vhost-user. Ο μηχανισμός SPDK/VVU αποτελεί επέκταση του μηχανισμού vhost-user, όπως αυτός ορίζεται στο εν λόγω πρωτόκολλο. Εν συντομία, ο μηχανισμός SPDK/VVU δουλεύει ως εξής: αντί να "τρέχουμε" τον SPDK vhost target στο χώρο χρήστη του host, τον "τρέχουμε" απομονωμένα μέσα σε μια εικονική μηχανή. Έτσι, έχουμε μια εικονική μηχανή που παρέχει αποθηκευτική λειτουργικότητα σε άλλες εικονικές μηχανές.

Το γεγονός ότι τρέχουμε το λογισμικό εικονικοποίησης μέσα σε μια εικονική μηχανή συνεπάγεται ότι πρέπει να επεκτείνουμε το μηχανισμό vhost-user, ώστε ο SPDK vhost target να εξακολουθεί να έχει πρόσβαση στη μνήμη της προς εξυπηρέτηση εικονικής μηχανής. Αυτό υλοποιείται μέσω της συσκευής virtio-vhost-user. Θα δούμε στις ακόλουθες ενότητες πώς επιτυγχάνεται αυτό. Επίσης, πρέπει να επεκτείνουμε τον κώδικα του SPDK και του DPDK (το SPDK χρησιμοποιεί το DPDK ως submodule), ώστε ο SPDK vhost target να μπορεί να λειτουργήσει πάνω από το νέο μηχανισμό. Ο νέος μηχανισμός επικοινωνίας λέγεται "virtio-vhost-user transport", διότι βασίζεται στην ομώνυμη συσκευή. Ο αρχικός μηχανισμός, όπως αυτός ορίζεται από το πρωτόκολλο vhost-user, λέγεται "AF_UNIX transport". Θα δούμε σε επόμενες ενότητες πώς θα τροποποιήσουμε τον κώδικα των SPDK και DPDK ώστε να υποστηρίζουν και τα δύο transports.

Σκοπός της παρούσης διπλωματικής είναι, όχι απλώς η σχεδίαση και υλοποίηση του μηχανισμού SPDK/VVU, αλλά και η ενσωμάτωσή του στα σχετιζόμενα έργα ανοιχτού λογισμικού. Ο σχεδιασμός που θα αναλύσουμε σε αυτό το κεφάλαιο είναι προς αυτή την κατεύθυνση και έχει προκύψει από αλληλεπίδραση με τις αντίστοιχες κοινότητες. Συγκεκριμένα, έχουμε επικοινωνήσει[11] με τον Stefan Hajnoczi (Software Engineer στην ομάδα εικονικοποίησης της RedHat), ο οποίος είναι ο εμπνευστής της συσκευής virtio-vhost-user. Ο Stefan μας έχει εξουσιοδοτήσει[12] ώστε να συνεχίσουμε τη δουλειά

[11] https://lists.01.org/pipermail/spdk/2018-September/002488.html
[12] https://lists.01.org/pipermail/spdk/2018-December/002822.html

του όσον αφορά την ολοκλήρωση του προσδιορισμού των χαρακτηριστικών της συσκευής (VIRTIO spec) και την ολοκλήρωση του κώδικα υλοποίησης της συσκευής στο QEMU. Εκτός από τον Stefan, έχουμε επίσης επικοινωνήσει και με τον Darek Stojaczyk (Software Engineer στην Intel και core maintainer του SPDK) και έχουμε καταλήξει σε ένα πλάνο εργασίας[13] σχετικά με τις αλλαγές που πρέπει να γίνουν στο SPDK και στο DPDK. Για περισσότερες πληροφορίες σχετικά με την τρέχουσα κατάσταση και τα επόμενα βήματα, σας παραπέμπουμε στην ενότητα Επίλογος.

5.2 Η συσκευή virtio-vhost-user

Η συσκευή virtio-vhost-user είναι μια συσκευή παραεικονικοποίησης που υπακούει στο πρότυπο Virtio. Σκοπός της είναι να επεκτείνει το μηχανισμό vhost-user ώστε να μπορούμε να τρέχουμε vhost targets μέσα σε εικονικές μηχανές. Θα δούμε στις επόμενες ενότητες πώς επιτυγχάνεται αυτό. Η συσκευή είναι εφοδιασμένη με ένα ζευγάρι ουρών virtio (virtqueues) για την επικοινωνία με τον guest driver. Επίσης, υποστηρίζει MSI-X interrupts. Αυτή τη στιγμη, οι προδιαγραφές της συσκευής περιγράφουν πώς μπορεί να υλοποιηθεί πάνω από δίαυλο PCI.

Σε αντίθεση με τις υπόλοιπες συσκευές virtio, η εν λόγω συσκευή είναι εφοδιασμένη με κάποιους επιπλέον πόρους. Αυτοί οι πόροι είναι τυποποιημένοι πάνω από διαύλο PCI με virtio PCI capabilities. Ονομαστικά, αυτοί οι επιπλέον πόροι είναι οι καταχωρητές-κουδούνια (doorbells), οι καταχωρητές για ειδοποιήσεις (notifications) και η μοιραζόμενη μνήμη (shared memory). Οι καταχωρητές-κουδούνια χρησιμοποιούνται από τον vhost target ώστε να μπορεί να ειδοποιεί τον master όταν ολοκληρώνεται η επεξεργασία ενός αιτήματος I/O. Είναι απεικονισμένοι σε διευθύνσεις MMIO του χώρου διευθύνσεων της συσκευής και είναι συσχετισμένοι με callfds. Αυτό σημαίνει ότι όταν ο guest driver "βαράει" ένα κουδούνι, το QEMU θα "βαρέσει" τον κατάλληλο callfd ώστε να ειδοποιηθεί ο master. Οι καταχωρητές για ειδοποιήσεις είναι το ακριβώς αντίθετο από τους καταχωρητές-κουδούνια. Χρησιμοποιούνται για να απεικονίζει ο guest driver MSI-X vectors σε kickfds. Οπότε, όταν η εικονική μηχανή master "βαράει" έναν kickfd, το QEMU στέλνει μια διακοπή με το κατάλληλο vector στην εικονική μηχανή slave. Τέλος, η μοιραζόμενη μνήμη είναι η μνήμη της εικονικής μηχανής master απεικονισμένη στο χώρο διευθύνσεων της συσκευής virtio-vhost-user.

[13] https://lists.01.org/pipermail/spdk/2019-March/003163.html

5.3 Επεκτείνοντας το μονοπάτι ελέγχου vhost-user

Σε αυτή την ενότητα θα δούμε πώς η συσκευή virtio-vhost-user επεκτείνει το μονοπάτι ελέγχου του πρωτοκόλλου vhost-user.

Το μονοπάτι ελέγχου του μηχανισμού vhost-user είναι ένα unix domain socket από το οποίο ο master στέλνει μηνύματα στον slave. Η ανταλλαγή μηνυμάτων λαμβάνει χώρα ως επί το πλείστον κατά την αρχικοποίηση του vhost target. Βασικός σκοπός του μονοπατιού ελέγχου είναι ο slave να αποκτήσει πρόσβαση στις virtqueues και στους I/O buffers στη μνήμη του master, η εγκαθίδρυση δηλαδή του μονοπατιού δεδομένων.

Η συσκευή virtio-vhost-user επεκτείνει το μονοπάτι ελέγχου ώστε ο vhost target, που τρέχει μέσα στην εικονική μηχανή slave, να λαμβάνει τα μηνύματα που στέλνει ο master. Η συσκευή virtio-vhost-user είναι ένα character device στο QEMU, οπότε είναι συσχετισμένη με το unix domain socket. Η συσκευή διαβάζει εισερχόμενα μηνύματα από το socket και τα προωθεί στον vhost target μέσω των virtqueues της συσκευής. Αυτό ισχύει για τα περισσότερα μηνύματα, αλλά όχι για όλα. Υπάρχουν συγκεκριμένα μηνύματα που απαιτούν περαιτέρω επεξεργασία από τη συσκευή πριν προωθηθούν στον vhost target. Ανάλογη διαδικασία ακολουθείται και για τα μηνύματα που στέλνει ο vhost target στον master, δηλαδή η συσκευή λαμβάνει μηνύματα μέσω των virtqueues και τα γράφει στο unix domain socket.

5.4 Επεκτείνοντας το μονοπάτι δεδομένων vhost-user

Σε αυτή την ενότητα θα δούμε πώς η συσκευή virtio-vhost-user επεκτείνει το μονοπάτι δεδομένων του πρωτοκόλλου vhost-user.

Το μονοπάτι δεδομένων του μηχανισμού vhost-user βασίζεται σε μοιραζόμενη μνήμη. Δηλαδή ο vhost target έχει πρόσβαση σε όλη τη μνήμη του master και, επομένως, στις virtqueues και στους I/O buffers που έχει δεσμεύσει ο οδηγός της συσκευής στη μνήμη της εικονικής μηχανής master. Πιο συγκεκριμένα, η μνήμη της εικονικής μηχανής master είναι ένα αρχείο στο tmpfs ή στο hugetlbfs, δηλαδή ένα αρχείο αποθηκευμένο στη φυσική μνήμη του μηχανήματος. Η εικονική μηχανή slave κάνει *mmap()* όλη τη μνήμη της εικονικής μηχανής master.

Η συσκευή virtio-vhost-user επεκτείνει το μονοπάτι δεδομένων ώστε ο vhost target,

που τρέχει μέσα στην εικονική μηχανή slave, να έχει πρόσβαση στη μνήμη της εικονικής μηχανής master. Αυτό υλοποιείται ως εξής:

ο master στέλνει ένα μήνυμα τύπου *VHOST_USER_SET_MEM_TABLE* μέσω του unix domain socket. Με αυτό το μήνυμα ο master δίνει πρόσβαση σε όλη τη μνήμη του στον slave (το Linux επιτρέπει σε μια διεργασία να περνά δικαιώματα σε ανοικτά αρχεία της σε άλλη διεργασία μέσω unix domain sockets [61]). Η συσκευή virtio-vhost-user λαμβάνει το μήνυμα και κάνει *mmap()* τη μνήμη του master. Στη συνέχεια, παρουσιάζει στον guest τη μνήμη του master σαν ένα μέρος του χώρου διευθύνσεων (MMIO) της συσκευής. Με αυτό τον τρόπο, ο vhost target μπορεί να έχει απευθείας πρόσβαση στη μνήμη του master χωρίς παρεμβολή του QEMU.

5.5 Επεκτείνοντας τους μηχανισμούς ειδοποιήσεων vhost-user

Σε αυτή την ενότητα θα δούμε πώς η συσκευή virtio-vhost-user επεκτείνει τους μηχανισμούς ειδοποιήσεων του πρωτοκόλλου vhost-user.

Οι μηχανισμοί ειδοποιήσεων αφορούν τόσο σε ειδοποιήσεις του master προς τον slave για νέα αιτήματα I/O, όσο και ειδοποιήσεις του slave προς τον master για την περάτωση των αιτημάτων I/O. Το πρωτόκολλο vhost-user ορίζει ότι και στις δύο περιπτώσεις χρησιμοποιούνται eventfds. Οι eventfds λέγονται kickfds και callfds για τις δύο περιπτώσεις. Οι eventfds δημιουργούνται από τον master και στέλνονται στον slave μέσω του unix domain socket. Συνήθως, η χρήση τους συνοδεύεται από τους μηχανισμούς ioeventfd/irqfd του KVM, ώστε το κόστος ανά VMEXIT να είναι μικρότερο.

Είναι προφανές ότι στην περίπτωση που ο vhost target τρέχει μέσα σε μια εικονική μηχανή, δεν μπορεί να χρησιμοποιεί απευθείας τους eventfds. Η συσκευή virtio-vhost-user παρέχει έναν εναλλακτικό μηχανισμό για τις ειδοποιήσεις. Συγκεκριμένα, η συσκευή λαμβάνει τα μηνύματα που κοινοποιούν στον slave τους eventfds και τους σώζει σε εσωτερικές δομές της συσκευής. Στη συνέχεια, απεικονίζει τους callfds σε καταχωρητές-κουδούνια (doorbells). Οπότε, όποτε ο vhost target “βαράει” ένα κουδούνι, η συσκευή “βαράει” τον callfd που αντιστοιχεί σε αυτόν τον καταχωρητή. Στη περίπτωση των kickfds, η συσκευή παρέχει ένα virtio PCI capability στον οδηγό συσκευής ώστε να μπορεί ο οδηγός συσκευής να απεικονίσει callfds σε MSI-X interrupt vectors. Στη περίπτωση που έχει γίνει αυτή η αντιστοίχιση, η συσκευή παράγει μια διακοπή MSI με

το κατάλληλο vector όποτε ο master “βαράει” έναν kickfd.

5.6 Αλλαγές στο SPDK και στο DPDK

Για την υποστήριξη του virtio-vhost-user transport, πρέπει να κάνουμε αλλαγές στον κώδικα των SPDK και DPDK.

5.6.1 Αρχιτεκτονική του vhost κώδικα στο SPDK

Το SPDK περιλαμβάνει μια βιβλιοθήκη “vhost” που υλοποιεί έναν vhost target. Η αρχιτεκτονική του κώδικα χωρίζεται σε τρία μέρη:

- η υλοποίηση του μονοπατιού ελέγχου (*lib/vhost/rte_vhost/*)
- η υλοποίηση του γενικού μονοπατιού δεδομένων (*lib/vhost/vhost.c*)
- η υλοποίηση του εξειδικευμένου μονοπατιού δεδομένων ανάλογα με το πρωτόκολλο αποθήκευσης (*lib/vhost/vhost_scsi[blk,nvme].c*)

5.6.2 Λίστα Αλλαγών

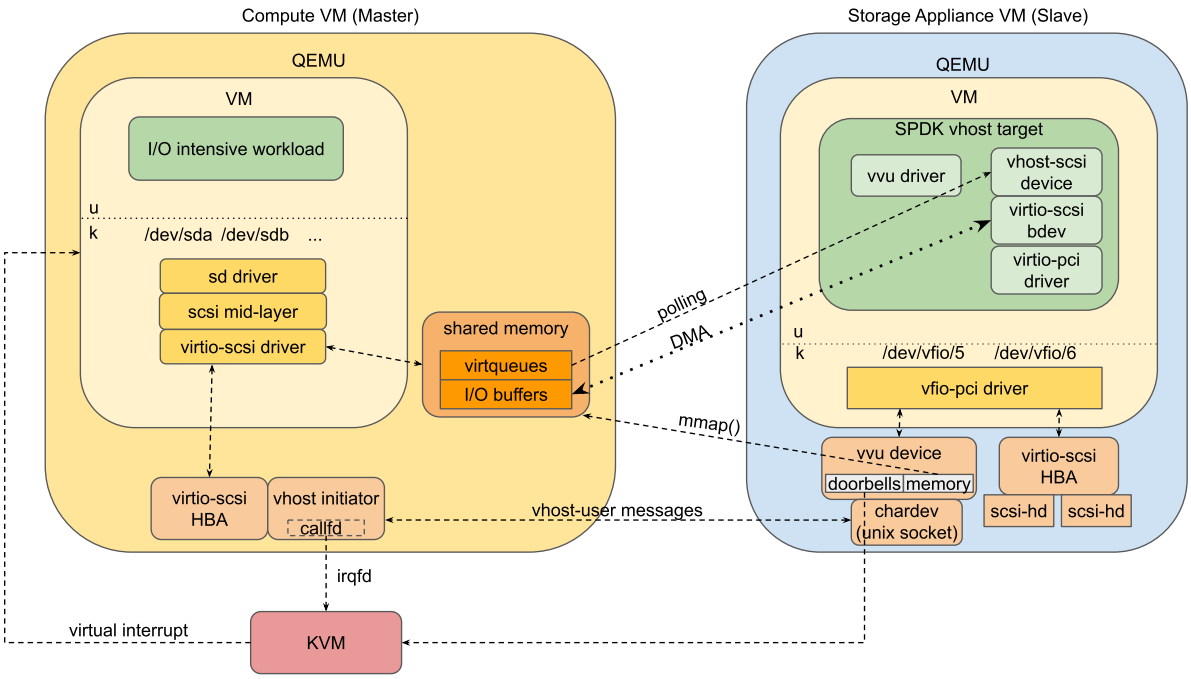
Η προσθήκη υποστήριξης για το virtio-vhost-user transport αφορά αποκλειστικά το μονοπάτι ελέγχου. Επομένως, οι αλλαγές αφορούν τη βιβλιοθήκη *rte_vhost* του SPDK. Ωστόσο, μετά από συνεννόηση με τον Darek Stojaczyk, αποφασίσαμε οι αλλαγές να γίνουν στη βιβλιοθήκη *librte_vhost* του DPDK. Ο λόγος είναι ότι η βιβλιοθήκη *rte_vhost* του SPDK, η οποία υλοποιεί το μονοπάτι ελέγχου, είναι ουσιαστικά ένα αντίγραφο της βιβλιοθήκης *librte_vhost* του DPDK. Η κοινότητα του SPDK έχει αποφασίσει να χρησιμοποιεί απευθείας τη βιβλιοθήκη του DPDK σε μελλοντικές εκδόσεις. Αυτή η λειτουργικότητα υπάρχει ήδη πειραματικά από την έκδοση 19.04 του SPDK.[14]

Για περισσότερες πληροφορίες σχετικά με την τρέχουσα κατάσταση και για επόμενα βήματα, σας παραπέμπουμε στην ενότητα Επίλογος.

[14] `https://spdk.io/release/2019/04/30/19.04_release/`

5.7 Περιγραφή Λειτουργίας του Μηχανισμού SPDK/VVU

Η τοπολογία μοιάζει ως εξής:



Σχήμα 3: *Τοπολογία Μηχανισμού SPDK/VVU*

Η τοπολογία αποτελείται από μια εικονική μηχανή όπου γίνεται υπολογιστική δουλειά (Compute VM ή master VM) και μια εικονική μηχανή που υλοποιεί τη συσκευή αποθήκευσης που παρέχεται στην πρώτη εικονική μηχανή (Storage Appliance VM ή slave VM). Η εικονική μηχανή master είναι εφοδιασμένη με μια συσκευή PCI τύπου vhost-user-scsi, η οποία είναι ένας virtio-scsi προσαρμογέας διαύλου (Host Bus Adapter - HBA) του οποίου το μονοπάτι δεδομένων υλοποιείται εκτός του QEMU. Συγκεκριμένα, το μονοπάτι δεδομένων, το οποίο αποκαλείται “vhost device backend” ή “vhost target”, υλοποιείται από την εφαρμογή vhost του SPDK που τρέχει στο χώρο χρήστη μέσα στην εικονική μηχανή slave. Επιπλέον, η εικονική μηχανή slave είναι εφοδιασμένη με έναν virtio-scsi προσαρμογέα διαύλου και δύο εικονικούς SCSI δίσκους συνδεδεμένους στο δίαυλο SCSI. Αυτοί οι δίσκοι θα παίξουν το ρόλο των τελικών μέσων αποθήκευσης για τα δεδομένα. Σημειώστε ωστόσο ότι η επιλογή αυτή είναι απλά ένα παράδειγμα. Υπάρχουν πολλές άλλες επιλογές για τελικά μέσα αποθήκευσης, με ποιο κατάλληλη για το μηχανισμό SPDK/VVU τη χρήση ενός passthrough NVMe δίσκου.

Η λειτουργία του μηχανισμού SPDK/VVU μπορεί να αναλυθεί σε δύο συνιστώσες: το μονοπάτι ελέγχου και το μονοπάτι δεδομένων. Το μονοπάτι ελέγχου χρησιμοποιείται για τη διαμόρφωση της συσκευής και, κυρίως, για την εγκαθίδρυση του μονοπατιού δεδομένων. Το μονοπάτι δεδομένων χρησιμοποιείται για τη μετοφορά των δεδομένων και των μετα-δεδομένων που σχετίζονται με κάθε αίτημα I/O.

Μονοπάτι Ελέγχου

Το μονοπάτι ελέγχου αφορά στην ανταλλαγή μηνυμάτων μέσω ενός unix domain socket. Η ανταλλαγή μηνυμάτων συμβαίνει κυρίως στη φάση της αρχικοποίησης της συσκευής, δηλαδή όταν ο master συνδέεται στο unix domain socket, το οποίο ελέγχει η συσκευή virtio-vhost-user. Η διαδικασία αρχικοποίησης δουλεύει ως εξής:

υποθέτουμε ότι η εικονική μηχανή slave είναι ενεργή και ότι ο SPDK vhost target τρέχει μέσα στην εικονική μηχανή slave περιμένοντας για νέες συνδέσεις. Σε αυτή τη φάση boot-άρουμε την εικονική μηχανή master. Η εικονική μηχανή master είναι εφοδιασμένη με μια συσκευή vhost-user-scsi. Η συσκευή αυτή είναι στην ουσία ο master που συνδέεται στο unix domain socket της συσκευής virtio-vhost-user. Αμέσως μετά τη σύνδεση, αρχίζει η αποστολή μηνυμάτων από τον master στον slave. Μεταξύ άλλων μηνυμάτων, ο master στέλνει ένα μήνυμα τύπου *VHOST_USER_SET_MEM_TABLE* με το οποίο δίνει πρόσβαση στη μνήμη του στον slave. Η μνήμη της εικονικής μηχανής master είναι ένα αρχείο (ή πολλά αρχεία) στο tmpfs ή στο hugetlbfs, ώστε να είναι μοιραζόμενη. Για κάθε κομμάτι μνήμης, ο master εισάγει τις εξής πληροφορίες στο ανωτέρω μήνυμα:

- έναν περιγραφέα αρχείου (file descriptor) - για *mmap()*
- host εικονική διεύθυνση - αντιστοιχεί στην εικονική διεύθυνση στον εικονικό χώρο διευθύνσεων του QEMU όπου είναι απεικονισμένο το συγκεκριμένο κομμάτι μνήμης του guest
- guest φυσική διεύθυνση - αντιστοιχεί στη φυσική διεύθυνση στο φυσικό χώρο διευθύσεων της guest CPU όπου είναι απεικονισμένο το συγκεκριμένο κομμάτι μνήμης του guest
- offset - offset όπου ο slave πρέπει να κάνει *mmap()* το εν λόγω αρχείο

- size - μέγεθος του κομματιού μνήμης

Η συσκευή virtio-vhost-user λαμβάνει αυτό το μήνυμα και κάνει *mmap()* όλα τα κομμάτια μνήμης της εικονικής μηχανής master. Στη συνέχεια, απεικονίζει αυτά τα κομμάτια μνήμης στον slave guest σαν μέρος του χώρου δευθύνσεων της συσκευής. Έπειτα, ο master στέλνει κάποιες ακόμη πληροφορίες στον slave, ώστε να μπορεί να εντοπίσει τις virtqueues μέσα στη μνήμη του master. Τέλος, στέλνει τους kickfds και callfds για τις virtqueues μέσω μηνυμάτων *VHOST_USER_SET_VRING_KICK* και *VHOST_USER_SET_VRING_CALL* αντίστοιχα. Η συσκευή virtio-vhost-user λαμβάνει τα μηνύματα αυτά. Τους μεν callfds τους σώζει σε εσωτερικές δομές και τους αντιστοιχίζει σε καταχωρητές-κουδούνια, τους οποίους και απεικονίζει στο χώρο διευθύνσεων της συσκευής. Τους δε kickfds επίσης τους σώζει και παρέχει ένα μηχανισμό στον guest driver ώστε να τους αντιστοιχίσει σε MSI-X vectors. Ωστόσο, στο μηχανισμό SPDK/VVU, το SPDK δε χρησιμοποιεί MSI-X vectors διότι εφαρμόζει polling.

Μονοπάτι Δεδομένων

Με την ολοκλήρωση της αρχικοποίησης της συσκευής, έχει δημιουργηθεί το μονοπάτι δεδομένων. Θα μελετήσουμε ένα απλό παράδειγμα ενός αιτήματος *read()* σε ένα αρχείο του δίσκου, ώστε να δείξουμε τη μεταφορά των δεδομένων και των μεταδεδομένων από άκρη σε άκρη.

Υποθέτουμε ότι μια διεργασία που τρέχει στο χώρο χρήστη της εικονικής μηχανής master κάνει *read()* σε ένα ανοικτό αρχείο που είναι αποθηκευμένο στον δίσκο vhost-user-scsi. Υποθέτουμε επίσης ότι η διεργασία χρησιμοποιεί το *O_DIRECT* flag ώστε να παρακάμψει την page cache του πυρήνα. Αυτή η παραδοχή απλά απλοποιεί το αφήγημά μας, καθώς δε χρειάζεται να ασχοληθούμε με την page cache, που δεν είναι και ο σκοπός αυτής της διπλωματικής. Το σύστημα αρχείων στο οποίο είναι αποθηκευμένο το αρχείο, θα μεταφράσει το αίτημα σε ένα σύνολο από Block I/O αιτήματα, τα οποία και προωθεί στο Block Layer του πυρήνα. Το Block Layer μετασχηματίζει τα Block I/O αιτήματα σε στιγμιότυπα της δομής *struct request* και τα παραδίδει στο Request Layer. Το Request Layer εφαρμόζει πολιτικές δρομολόγησης των αιτημάτων και, εν τέλει, παραδίδει τα αιτήματα στον κατάλληλο block device driver. Εν προκειμένω, ο block device driver είναι ο sd driver, που είναι μέρος του υποσυστήματος SCSI του

πυρήνα. Ο sd driver μετασχηματίζει το Block I/O αίτημα σε ένα αίτημα τύπου SCSI και το παραδίδει στο SCSI mid-layer. Το SCSI mid-layer με τη σειρά του το παραδίδει στον low-level driver, ο οποίος εν προκειμένω είναι ο virtio-scsi driver του πυρήνα. Ο virtio-scsi driver μετασχηματίζει το αίτημα σε ένα αίτημα τύπου *struct virtio_scsi_req_cmd*. Στη συνέχεια το τοποθετεί στη request virtqueue, χρησιμοποιώντας όσους descriptors χρειάζεται. Ο vhost target στην εικονική μηχανή slave, ο οποίος παρακολουθεί τις virtqueues στη μνήμη του master μέσω της συσκευής virtio-vhost-user, παρατηρεί τους νέους descriptors και ανασυνθέτει το αίτημα (κάνοντας μεταξύ άλλων και μετάφραση διευθύνσεων). Το αίτημα αυτό προωθείται στο SCSI επίπεδο του SPDK. Το SCSI επίπεδο βρίσκει σε ποιά συσκευή (SCSI LUN) απευθύνεται το αίτημα, το μετασχηματίζει σε ένα αντικείμενο τύπου *spdk_bdev_io* (εσωτερική αναπαράσταση των αιτημάτων Block I/O στο SPDK) και το παραδίδει στον κατάλληλο driver. Εν προκειμένω, το αίτημα παραδίδεται στον virtio-scsi driver του SPDK, ώστε τελικά να παραδοθεί στο τελικό μέσο αποθήκευσης. Το τελικό μέσο αποθήκευσης πραγματοποιεί DMA απευθείας στη μνήμη του master και εξυπηρετεί το αίτημα. Ο virtio-scsi driver του SPDK παρατηρεί την ολοκλήρωση του αιτήματος πραγματοποιώντας polling πάνω στις virtqueues της virtio-scsi συσκευής. Τότε ειδοποιεί τον vhost target καλώντας το κατάλληλο callback και ο vhost target κάνει τις κατάλληλες τροποποιήσεις στη virtqueue. Τέλος, ο vhost target ειδοποιεί τον driver στη πλευρά του master για την ολοκλήρωση του αιτήματος με το να "χτυπήσει" τον κατάλληλο καταχωρητή-κουδούνι της συσκευής virtio-vhost-user. Η συσκευή "βαράει" τον αντίστοιχο callfd και το KVM εισάγει μια διακοπή στην εικονική μηχανή master.

Συμπερασματικά, ο μηχανισμός SPDK/VVU έχει τις εξής δύο σημαντικές ιδιότητες:

1. το μονοπάτι δεδομένων παρακάμπτει σχεδόν εξ ολοκλήρου τον επόπτη (QEMU). Υπάρχουν, ωστόσο, ακόμη δύο σημεία που απαιτούν τη συμμετοχή του επόπτη. Συγκεκριμένα, οι ειδοποιήσεις του slave προς τον master για την ολοκλήρωση των αιτημάτων I/O μέσω των callfds και η παραγωγή εικονικών διακοπών στη πλευρά του master. Και οι δύο περιπτώσεις θα μπορούσαν να παρακαμφθούν αν χρησιμοποιούσαμε έναν poll-mode virtio-scsi driver στη πλευρά του master.

2. η μεταφορά των δεδομένων γίνεται χωρίς αντίγραφα. Τα δεδομένα μεταφέρονται απευθείας από τη μνήμη του master στο τελικό μέσο αποθήκευσης. Αντίγραφα υπάρχουν μόνο για τα μετα-δεδομένα, τα οποία περιγράφουν κάθε αί-

τημα I/O.

6 Υλοποίηση

Για την υλοποίηση του μηχανισμού SPDK/VVU εργαστήκαμε πάνω στα εξής έργα ανοικτού λογισμικού: SPDK, DPDK, QEMU, VIRTIO. Η υλοποίηση χωρίζεται σε δύο κατευθύνσεις: στην ολοκλήρωση των προδιαγραφών και της υλοποίησης της συσκευής virtio-vhost-user από τη μία μεριά, και στη προσθήκη αλλαγών στα SPDK-DPDK για την υποστήριξη του virtio-vhost-user transport από την άλλη μεριά. Στις ακόλουθες ενότητες παρατίθεται αναλυτικά οι αλλαγές που κάναμε στα προαναφερθέντα έργα.

6.1 Αλλαγές στις προδιαγραφές της συσκευής virtio-vhost-user

΄Οσον αφορά τις προδιαγραφες της συσκευής virtio-vhost-user, βασιστήκαμε στην αρχική υλοποίηση του Stefan Hajnoczi[15] και προσθέσαμε κάποιες αλλαγές. Στόχος μας είναι η ενσωμάτωση των προδιαγραφών αυτών στο πρότυπο virtio. Οι αλλαγές που κάναμε είναι οι εξής:

1. διορθώσαμε κάποια ήσσονος σημασίας λάθη

2. προσθέσαμε κάποιες επιπλέον απαιτήσεις στο notification capability.

3. φροντίσαμε ώστε η περιγραφή για το shared memory capability να είναι σε συμφωνία με το *VIRTIO_PCI_CAP_SHARED_MEMORY_CFG* capability που επιχειρεί να προσθέσει ο Alan Gilbert στο virtio. Η συσκευή μας θα βασίζεται σε αυτό το capability.

4. προσθέσαμε μια λίστα με απαιτήσεις τις οποίες πρέπει να ακολουθεί οποιαδήποτε υλοποίηση της συσκευής ή ενός driver για τη συσκευή.

Στη συνέχεια, στείλαμε την αναθεωρημένη έκδοση των προδιαγραφών στη mailing list[16] του virtio. Αυτή τη στιγμή αναμένουμε για σχόλια από την κοινότητα.

[15] https://lists.oasis-open.org/archives/virtio-dev/201801/msg00110.html
[16] https://lists.oasis-open.org/archives/virtio-dev/201906/msg00036.html

6.2 Αλλαγές στην υλοποίηση της συσκευής virtio-vhost-user

Όσον αφορά την υλοποίηση της συσκευής virtio-vhost-user στο QEMU, επίσης βασιστήκαμε στην αρχική υλοποίηση του Stefan Hajnoczi[17]. Ωστόσο, η υλοποίηση αυτή ήταν ημιτελής και απαιτούσε αρκετές αλλαγές. Στις ακόλουθες υποενότητες αναλύονται οι αλλαγές που προσθέσαμε.

Διαχωρισμός του κώδικα σε frontend και backend μέρος

Μια εικονική συσκευή αποτελείται από δύο μέρη: το frontend μέρος και το backend μέρος. Το frontend μέρος αποτελεί την υλοποίηση των πόρων της συσκευής PCI. Το backend μέρος αποτελεί την υλοποίηση της λειτουργικότητας της συσκευής.

Στη περίπτωση της συσκευής virtio-vhost-user, έχουμε εξάγει όλο το κώδικα που αφορά το frontend μέρος στο αρχείο *virtio-vhost-user-pci.c* και έχουμε κρατήσει το κώδικα που υλοποιεί το backend μέρος στο αρχείο *virtio-vhost-user.c*.

Υλοποίηση διακοπών ως απόκριση σε ειδοποιήσεις του master

Προσθέσαμε τη δυνατότητα για παραγωγή διακοπών όποτε ο master "βαράει" έναν kickfd. Αυτή η λειτουργικότητα προβλέπεται από τις προδιαγραφές τις συσκευής, αλλά δεν είχε υλοποιηθεί. Η συσκευή δηλώνει τους kickfds που στέλνει ο master στο main event loop του QEMU. Σε όλους τους kickfds δηλώνει ως handler τη συνάρτηση *event_notifier_set_handler()*. Η συνάρτηση αυτή εισάγει διακοπές στον guest και καλείται από το main event loop όποτε ο master "βαράει" έναν kickfd.

Χρήση του μηχανισμού ioeventfd για τους callfds

Οι callfds είναι συσχετισμένοι με καταχωρητές-κουδούνια (doorbells). Προκειμένου να μειώσουμε το κόστος της εικονικοποίησης των προσβάσεων σε doorbells, χρησιμοποιούμε το μηχανισμό ioeventfd του KVM. Στην ουσία, αυτό που πετυχαίνουμε είναι ότι, σε περίπτωση που ο guest driver γράψει σε ένα doorbell, το KVM "βαράει" τον αντίστοιχο callfd αντί του QEMU. Με αυτό το τρόπο γλυτώνουμε τα context switches

[17] https://lists.nongnu.org/archive/html/qemu-devel/2018-01/msg04806.html

μεταξύ χώρου χρήστη και χώρου πυρήνα. Για τη δήλωση των callfds ως ioeventfds χρησιμοποιούμε τη συνάρτηση *memory_region_add_eventfd()*.

Υλοποίηση των virtio PCI capabilities για τους πρόσθετους πόρους της συσκευής

Οι προδιαγραφές της συσκευής προβλέπουν τη χρήση virtio PCI capabilities για την τυποποίηση των πρόσθετων πόρων της συσκευής. Με τον όρο πρόσθετους πόρους αναφερόμαστε στους καταχωρητές-κουδούνια (doorbells), στους καταχωρητές για ειδοποιήσεις (notifications), και στη μοιραζόμενη μνήμη.

Έχουμε υλοποιήσει τρία πρόσθετα capabilities για τους ισάριθμους πόρους της συσκευής. Τα capabilities αρχικοποιούνται από τη συνάρτηση *virtio_vhost_user_init_bar()*. Κάθε capability συσχετίζεται με έναν πόρο. Οι πόροι για τους doorbells, τα notifications και τη μοιραζόμενη μνήμη τοποθετούνται στον MMIO BAR 2 της συσκευής. Για τους doorbells και τα notifications ορίζουμε *read()*/*write()* handlers. Αυτό γίνεται κατά τη δήλωση των πόρων ως μέρη του χώρου διευθύνσεων της συσκευής με τη συνάρτηση *memory_region_init_io()*. Τέλος, συσχετίζουμε τα virtio PCI capabilities με τους αντίστοιχους πόρους και εισάγουμε τα capabilities στη λίστα με τα capabilities της συσκευής με τη συνάρτηση *virtio_pci_modern_region_map()*.

Υλοποίηση του πεδίου UUID

Οι προδιαγραφές της συσκευής προβλέπουν την ύπαρξη ενός καταχωρητή UUID με σκοπό την ταυτοποίηση της συσκευής από τον οδηγό της συσκευής ανεξαρτήτως της διεύθυνσης PCI της συσκευής. Η τιμή του UUID ορίζεται κατά την αρχικοποίηση της συσκευής και προκύπτει με χρήση της συνάρτησης *qemu_uuid_generate()*.

6.3 Αλλαγές στο DPDK

Οι αλλαγές στο DPDK επικεντρώνονται κυρίως στη βιβλιοθήκη *librte_vhost*, που αποτελεί την υλοποίηση του μονοπατιού ελέγχου του πρωτοκόλλου vhost-user. Σκοπός είναι η υποστήριξη του virtio-vhost-user transport. Οι αλλαγές αναλύονται στις ακόλουθες υποενότητες.

Εισαγωγή της δομής vhost transport operations

Η δημιουργία μιας γενικής διεπαφής που θα υλοποιείται από κάθε transport επιτρέπει να έχουμε πολλαπλά transports. Εν προκειμένω, χρειαζόμαστε μιας υλοποίηση του *AF_UNIX* transport - ο παραδοσιακός μηχανισμός που ορίζεται στο πρωτόκολλο vhost-user - και μια υλοποίηση του virtio-vhost-user transport. Εισάγουμε τη δομή *struct vhost_transport_ops*, η οποία περιγράφει τη διεπαφή που πρέπει να υλοποιεί το κάθε transport.

Εξαγωγή του σχετιζόμενου κώδικα με το AF_UNIX transport

Θέλουμε η υλοποίηση των μηχανισμών του κάθε transport να διατηρείται σε ξεχωριστά αρχεία. Γι' αυτό το σκοπό εξάγουμε όλες τις λειτουργίες που σχετίζονται με το *AF_UNIX* transport από τα αρχεία *socket.c* και *vhost_user.c* και τις τοποθετούμε σε ένα νέο αρχείο που λέγεται *trans_af_unix.c*. Με αυτό το τρόπο, μπορούμε να έχουμε κώδικα που είναι μοιραζόμενος και υλοποιεί λειτουργίες γενικές και για τα δύο transports, ενώ ο ειδικευμένος κώδικας για κάθε transport είναι συμμαζεμένος σε ξεχωριστά αρχεία.

Εισαγωγή του virtio-vhost-user driver και του αντίστοιχου transport

Το virtio-vhost-user transport βασίζεται στην ομώνυμη συσκευή. Γι' αυτό το λόγο χρειαζόμαστε ένα driver για τη συσκευή. Ο driver τοποθετείται στο κατάλογο *drivers/virtio_vhost_user*. Στην ίδια τοποθεσία εισάγεται και η υλοποίηση του virtio-vhost-user transport με το αρχείο *trans_virtio_vhost_user.c*. Για την υποστήριξη του νέου transport έχουμε ενημερώσει τα σχετικά Makefiles.

Εξαγωγή του virtio-vhost-user transport μέσω της διεπαφής της librte_vhost

Θέλουμε ένα μηχανισμό για την επιλογή ανάμεσα στα δύο transports. Για το σκοπό αυτό, εισάγουμε ένα νέο flag, το *RTE_VHOST_USER_VIRTIO_TRANSPORT* flag. Το flag αυτό περνάται μέσω του δεύτερου ορίσματος της συνάρτησης *rte_vhost_driver_register()* και υποδηλώνει ότι επιθυμούμε τη χρησιμοποίηση του virtio-vhost-user transport.

Σε περίπτωση που δε χρησιμοποιείται το flag, εννοείται ότι επιλέγουμε το προκαθορισμένο *AF_UNIX* transport. Η συνάρτηση αυτή χρησιμοποιείται για τη δημιουργία ενός νέου vhost target.

6.4 Αλλαγές στο SPDK

Οι αλλαγές στο SPDK επικεντρώνονται στην αξιοποίηση του virtio-vhost-user transport που παρέχει η βιβλιοθήκη *librte_vhost* του DPDK. Οι αλλαγές αναλύονται στις ακόλουθες υποενότητες.

Ενσωμάτωση του virtio-vhost-user transport στη βιβλιοθήκη libspdk_vhost

Επεκτείνουμε το API του SPDK (JSON RPC calls και configuration files) ώστε να είναι δυνατή η επιλογή μεταξύ των δύο διαθέσιμων transports από τον χρήστη. Συγκεκριμένα, η επιλογή γίνεται με βάση το όνομα του vhost controller. Αν το όνομα είναι μια διεύθυνση PCI, τότε ο χρήστης θέλει να χρησιμοποιήσει το virtio-vhost-user transport και η διεύθυνση PCI αναμένεται να αντιστοιχεί σε μια συσκευή virtio-vhost-user. Σε διαφορετική περίπτωση, ο χρήστης επιλέγει το *AF_UNIX* transport. Ο έλεγχος για το όνομα του vhost controller γίνεται στη συνάρτηση *spdk_vhost_dev_register()*, η οποία χρησιμοποιείται για τη δημιουργία ενός νέου vhost controller.

Υποστήριξη του vfio σε no-IOMMU mode

Στη τοπολογία SPDK/VVU δεν έχουμε κανένα λόγο να χρησιμοποιήσουμε μια εικονική IOMMU για την εικονική μηχανή slave. Γι' αυτό το σκοπό θέλουμε το SPDK να μπορεί να χρησιμοποιεί το vfio και σε no-IOMMU mode. Το no-IOMMU mode διατηρεί την ίδια λειτουργικότητα, δηλαδή απευθείας πρόσβαση στους πόρους μιας PCI συσκευής από το χώρο χρήστη, με μόνη διαφορά ότι δεν χρησιμοποιούμε μεταφράσεις διευθύνσεων για τις λειτουργίες DMA της συσκευής.

Για της υποστήριξη του vfio σε no-IOMMU mode χρειάστηκαν δύο αλλαγές. Η πρώτη αλλαγή αφορά στο setup script (*scripts/setup.sh*). Το script αυτό, μεταξύ άλλων, κάνει bind συσκευές στον vfio driver. Τροποποιήσαμε τη συνθήκη ελέγχου ώστε να χρησιμοποιεί τον vfio driver σε περίπτωση που υπάρχει ο εν λόγω driver, είτε σε no-IOMMU

mode, είτε σε x86 mode. Η δεύτερη αλλαγή αφορά στο vtophys mapping, δηλαδή στον εσωτερικό πίνακα μεταφράσεων διευθύνσεων του SPDK που χρησιμοποιείται για τον καθορισμό των διευθύνσεων DMA. Σε περίπτωση που χρησιμοποιείται ο vfio driver σε no-IOMMU mode, κρατάμε VA-to-PA (virtual address to physical address) μεταφράσεις στο πίνακα αυτό. Σε περίπτωση που χρησιμοποιείται ο vfio driver σε x86 mode, δηλαδή χρησιμοποιείται η IOMMU του συστήματος, τότε κρατάμε μεταφράσεις VA-to-IOVA (virtual address to IO virtual address).

Επέκταση του memory map για υποστήριξη μεταφράσεων μη ευθυγραμμισμένων στα 2MB

Η δομή των memory maps του SPDK, συμπεριλαμβανομένου του vtophys map, είναι δύο επιπέδων με το δέυτερο επίπεδο να έχει ως βαθμό διαμέρισης τα 2MB. Αυτό σημαίνει ότι οι μεταφράσεις που αποθηκεύονται αφορούν τμήματα του εικονικού χώρου διευθύνσεων της διεργασίας μεγέθους και ευθυγράμμισης 2MB. Αυτή η επιλογή είναι θεμιτή στις περιπτώσεις όπου χρησιμοποιούνται υπερσελίδες (hugepages), διότι, εξ ορισμού, οι υπερσελίδες (οι εικονικές διευθύνσεις στις οποίες απεικονίζονται) είναι ευθυγραμμισμένες στα 2MB. Το πρόβλημα προκύπτει στο σενάριο SPDK/VVU, όπου οι διευθύνσεις για DMA αντιστοιχούν στο χώρο διευθύνσεων της συσκευής virtio-vhost-user, οι οποίες δεν αντιστοιχούν σε υπερσελίδες και, επομένως, δεν είναι απαραίτητα ευθυγραμμισμένες στα 2MB.

Προς το παρόν, έχουμε συμφωνήσει με τον Darek να παρακάμψουμε το πρόβλημα αυτό με το να κάνουμε *mmap()* όλους του πόρους όλων των PCI συσκευών σε εικονικές διευθύνσεις που είναι ευθυγραμμισμένες στα 2MB. Ο τελικός στόχος, όμως, είναι η επέκταση της δομής των memory maps, ώστε να μπορούν να κρατούν μεταφράσεις για διευθύνσεις μη ευθυγραμμισμένες στα 2MB.

Δήλωση της συσκευής virtio-vhost-user σαν στόχο για λειτουργίες DMA

Όπως αναφέραμε στην προηγούμενη υποενότητα, στο μηχανισμό SPDK/VVU το τελικό μέσο αποθήκευσης κάνει DMA από πόρους της συσκευής virtio-vhost-user. Αυτό προϋποθέτει ότι υπάρχουν μεταφράσεις VA-to-PA στο vtophys map που αντιστοιχούν στο χώρο διευθύνσεων της συσκευής. Για να γίνει όμως αυτό, για να βρεί δηλαδή το

SPDK τις φυσικές διευθύνσεις που αντιστοιχούν στον mmapped χώρο διευθύνσεων της συσκευής, πρέπει να έχουμε δηλώσει τη συσκευή στη λίστα *g_vtophys_pci_devices* του SPDK. Το λύσαμε αυτό στη συνάρτηση *spdk_vhost_dev_register()* με το να δηλώνουμε κάθε συσκευή virtio-vhost-user στην εν λόγω λίστα.

7 Αξιολόγηση

Σε αυτή την ενότητα θα δούμε ποιά είναι τα πλεονεκτήματα της μεθόδου SPDK/VVU.

7.1 Κόστος Εικονικοποίησης

Το κόστος εικονικοποίησης αφορά στο κόστος που έχει η εξυπηρέτηση κάθε αιτήματος I/O. Η μέθοδος SPDK/VVU έχει σημαντικά μειωμένο κόστος σε σχέση με άλλες υπάρχουσες προσεγγίσεις. Αναλυτικά, η συμπεριφορά της μεθόδου ως προς τις διάφορες πηγές κόστους εικονικοποίησης είναι η εξής:

- **ειδοποιήσεις του guest οδηγού συσκευής**

  Το SPDK βασίζεται αποκλειστικά σε polling για τη γνωστοποίηση νέων αιτημάτων I/O. Γι' αυτό καταστέλλει τις ειδοποιήσεις από τον guest οδηγό συσκευής. Αυτό γίνεται με χρήση του *VRING_USER_F_NO_NOTIFY* flag, όπως αυτό ορίζεται στο πρότυπο virtio. Ο vhost target ενημερώνεται για νέα αιτήματα I/O παρακολουθώντας τις virtqueues στη μνήμη του master μέσω της συσκευής virtio-vhost-user. Εδώ να σημειώσουμε ότι υπάρχει ένα πρόσθετο κόστος για τις προσβάσεις στον πόρο μοιραζόμενης μνήμης της συσκευής virtio-vhost-user. Αυτό το κόστος προκύπτει και εξαρτάται από της τεχνική εικονικοποίησης της μνήμης. Στους σύγχρονους επεξεργαστές, η εικονικοποίηση της μνήμης είναι υποβοηθούμενη από το υλικό (βλέπε [83]).

- **διακοπές από τη συσκευή**

  Ο vhost target πρέπει να στέλνει ειδοποίησεις στον guest οδηγό συσκευής στην εικονική μηχανή master για της ολοκλήρωση των αιτημάτων I/O. Ο μηχανισμός SPDK/VVU εφαρμόζει δύο βελτιστοποιήσεις στο ζήτημα αυτό. Η πρώτη βελτιστοποίηση είναι ότι πραγματοποιεί συγχώνευση των διακοπών. Δηλαδή, αντί να ειδοποιεί τον οδηγό συσκευής για την ολοκλήρωση κάθε αιτήματος, τον ειδοποιεί για την ολοκλήρωση πολλών αιτημάτων. Η δεύτερη βελτιστοποίηση είναι ότι το κόστος εικονικοποίησης των διακοπών μειώνεται με τη χρήση των μηχανισμών ioeventfd/irqfd του KVM. Ο μηχανισμός υλοποίησης των διακοπών δουλεύει ως εξής: ο vhost target "βαράει" ένα doorbell της συσκευής

virtio-vhost-user για να στείλει μια ειδοποίηση. Το KVM προσομοιώνει αυτή την ενέργεια "βαρώντας" τον αντίστοιχο callfd και πραγματοποιεί *VMENTRY*. Στη μεριά του slave, το KVM θα παρατηρήσει το event στον callfd, διότι είναι δηλωμένος σαν irqfd. Ως απόκριση σε αυτό το event, θα στείλει ένα interrupt στην εικονική μηχανή master.

- **εξυπηρέτηση των αιτημάτων I/O**

  Ο μηχανισμός SPDK/VVU υιοθετεί όλα τα πλεονεκτήματα του SPDK ως ένα framework για αποδοτικό I/O με συσκευές αποθήκευσης. Αυτά τα πλεονεκτήματα είναι κυρίως η αποδοτικότητα (efficiency) και η κλιμακωσιμότητα (scalability). Αποδοτικότητα εδώ σημαίνει ότι το SPDK χρειάζεται λιγότερους πυρήνες να εκτελούν I/O ώστε να πετύχει το ίδιο IOPS με τον πυρήνα του Linux. Με άλλα λόγια, μπορούμε με λιγότερους πυρήνες για I/O να καλύψουμε τις ανάγκες για αποθηκευτική λειτουργικότητα περισσότερων εικονικών μηχανών. Αυτό είναι ιδιαίτερα σημαντικό καθώς επιτρέπει να έχουμε περισσότερες εικονικές μηχανές ανά φυσικό μηχάνημα, διότι έχουμε περισσότερους πυρήνες διαθέσιμους για υπολογιστικά φορτία. Κλιμακωσιμότητα σημαίνει ότι μπορούμε αυξάνοντας, είτε τον αριθμό των πυρήνων, είτε τον αριθμό των δίσκων, να αυξήσουμε το throughput. Και τα δύο ανωτέρω πλεονεκτήματα ισχύουν στη περίπτωση που παρακάμπτουμε πλήρως τον πυρήνα στη μεριά του slave. Δηλαδή στη περίπτωση που χρησιμοποιούμε ως τελικό μέσο αποθήκευσης έναν passthrough δίσκο.

  Ένα επίσης σημαντικό πλεονέκτημα του μηχανισμού SPDK/VVU που προκύπτει από το SPDK είναι ότι δουλεύει χωρίς αντίγραφα στο μονοπάτι δεδομένων. Δηλαδή, το τελικό μέσο αποθήκευσης κάνει DMA απευθείας στη μνήμη της εικονικής μηχανής master.

7.2 Ασφάλεια

Από πλευράς ασφαλείας, είναι προφανές ότι το να "τρέχουμε" το λογισμικό εικονικοποίησης (SPDK) μέσα σε μια εικονική μηχανή είναι σαφώς πιο ασφαλές από το να το "τρέχουμε" στο χώρο χρήστη στον host.

7.3 Μεταφορά Ελέγχου στο Χρήστη

Στην εισαγωγική ενότητα αναφερθήκαμε σε αυτό το χαρακτηριστικό με της έννοια "ευελιξία". Αυτή είναι ίσως η σημαντικότερη ιδιότητα του μηχανισμού SPDK/VVU σε σχέση με άλλες προσεγγίσεις. Συνιστά μάλιστα καινοτομία καθώς δεν προσφέρεται τέτοιου είδους υπηρεσία από κάποιο δημόσιο περιβάλλον νέφους. Πιο αναλυτικά, ο μηχανισμός SPDK/VVU επιτρέπει στους χρήστες σε ένα περιβάλλον νέφους να έχουν τον πλήρη έλεγχο των συσκευών αποθήκευσης που τους δίνει ο πάροχος. Ο εκάστοτε χρήστης παραλαμβάνει έναν φυσικό δίσκο από τον πάροχο και ένα σύνολο από εικονικές μηχανές όπου θα τρέξει τις εφαρμογές του. Ο χρήστης μπορεί μετέπειτα να χειριστεί τον δίσκο όπως νομίζει. Μπορεί να τον μοιράζει ανάμεσα στις εικονικές μηχανές του, μπορεί να τον μετακινεί από τη μία εικονική μηχανή στην άλλη χωρίς να χρειάζεται επανεκκινήσεις, και ούτω καθεξής. Από τη μεριά του παρόχου, ο μηχανισμός αυτός είναι απολύτως ασφαλής, καθώς ο χρήστης πράττει όλες τις ενέργειες αυτές μέσα σε μια εικονική μηχανή.

Εκτός από την ευελιξία όσον αφορά το χειρισμό των δίσκων, ο χρήστης έχει επίσης την ευελιξία όσον αφορά το μονοπάτι δεδομένων. Ο μηχανισμός SPDK/VVU παρέχει στο χρήστη πλήρη έλεγχο στο μονοπάτι δεδομένων. Αυτό σημαίνει ότι μπορεί ο χρήστης να κάνει ρυθμίσεις με βάση της εφαρμογή που θέλει να τρέξει, τόσο στη μεριά του guest πυρήνα στην εικονική μηχανή master, όσο και στη μεριά του SPDK vhost target στην εικονική μηχανή slave, ώστε να έχει τη μέγιστη δυνατή απόδοση.

8 Επίλογος

8.1 Αποτίμηση

Ο σχεδιασμός και η υλοποίηση του μηχανισμού SPDK/VVU περιελάμβανε ενασχόληση με πολλά έργα ανοικτού κώδικα και αλληλεπίδραση με τις αντίστοιχες κοινότητες. Ο απώτερος στόχος μας εξαρχής ήταν, όχι μόνο η υλοποίηση του μηχανισμού SPDK/VVU, αλλά και η ενσωμάτωσή του στα αντίστοιχα έργα ανοικτού κώδικα. Αν και στο διάστημα συγγραφής της παρούσης διπλωματικής ο δεύτερος στόχος δεν έχει ολοκληρωθεί, έχουμε σημειώσει σημαντική πρόοδο προς αυτή την κατεύθυνση. Έχουμε καταφέρει να ενσωματώσουμε κάποιες αλλαγές στο SPDK[18] και έχουμε καταλήξει σε συμφωνία με τη κοινότητα σε κάποιες άλλες[19], οι οποίες θα ενσωματωθούν κι αυτές αφότου περάσουμε το virtio-vhost-user transport στο DPDK. Έχουμε επίσης στείλει ένα εκτενές πακέτο αλλαγών στο DPDK[20] για την ενσωμάτωση του virtio-vhost-user transport στη βιβλιοθήκη *librte_vhost*. Αναμένουμε σχόλια από τη κοινότητα πάνω σε αυτό. Σχετικά με τη συσκευή virtio-vhost-user, έχουμε στείλει ένα πακέτο αλλαγών στη λίστα virtio-dev[21] για τις προδιαγραφές της συσκευής και αναμένουμε σχόλια. Τέλος, έχουμε στείλει ένα mail στη λίστα του QEMU[22] στο οποίο παραθέτουμε το βελτιωμένο κώδικα της συσκευής και ζητάμε για σχόλια/παρατηρήσεις.

Αναφορικά με τα επόμενα βήματα, πρώτος στόχος είναι η ενσωμάτωση του virtio-vhost-user transport στο DPDK. Στόχος είναι η ολοκλήρωση αυτού στην έκδοση 19.11. Παράλληλα, θέλουμε να προχωρήσει η διαδικασία για την έγκριση των προδιαγραφών της συσκευής virtio-vhost-user. Η συζήτηση αναμένεται να ενταθεί μετά την ολοκλήρωση ενός άλλου προγενέστερου πακέτου αλλαγών που εκκρεμεί. Αμέσως μετά την έγκριση των προδιαγραφών, θα εντείνουμε τη διαδικασία για την ενσωμάτωση της υλοποίησης της συσκευής στο QEMU. Τέλος, αφότου έχουν ολοκληρωθεί τα ανωτέρω, θα ζητήσουμε από τη κοινότητα του SPDK να περάσει τις προσυμφωνηθέντες αλλαγές για τη χρήση του virtio-vhost-user transport στο SPDK.

[18] https://review.gerrithub.io/q/owner:+Dragazis+status:merged
[19] https://review.gerrithub.io/q/status:+open+owner:+Dragazis+repo:+spdk/spdk
[20] http://mails.dpdk.org/archives/dev/2019-June/135116.html
[21] https://lists.oasis-open.org/archives/virtio-dev/201906/msg00036.html
[22] https://lists.gnu.org/archive/html/qemu-devel/2019-04/msg02910.html

8.2 Μελλοντικές Επεκτάσεις

Υπάρχουν διάφορες προτάσεις για μελλοντική δουλειά πάνω στο μηχανισμό SPDK/VVU. Για παράδειγμα, θα μπορούσαμε να εμπλουτίσουμε τη λίστα με τα CI tests του SPDK με tests για το virtio-vhost-user transport. Επίσης, θα μπορούσαμε να ενσωματώσουμε το μηχανισμό SPDK/VVU στα Katacontainers[86]. Να έχουμε δηλαδή μια εικονική μηχανή ελεγχόμενη από το χρήστη που να προσφέρει υψηλής απόδοσης υπηρεσίες αποθήκευσης σε containers. Μια άλλη ενδιαφέρουσα επέκταση όσον αφορά το SPDK θα ήταν η υλοποίηση ενός συστήματος αρχείων πάνω από το Block Layer. Αυτό θα ήταν ιδιαίτερα χρήσιμο ώστε να μπορούν εφαρμογές να αλληλεπιδρούν απευθείας με συσκευές αποθήκευσης μέσω του SPDK. Για παράδειγμα, θα μπορούσαμε με αυτό το τρόπο να αντικαταστήσουμε στην τοπολογία SPDK/VVU τον πυρήνα στην εικονική μηχανή master με SPDK, έχοντας εν τέλει ένα μονοπάτι δεδομένων απ' άκρη σε άκρη αποκλειστικά στο χώρο χρήστη. Επιπρόσθετα, θα μπορούσαμε να τροποποιήσουμε τη διεπαφή με το χρήστη για την επιλογή transport. Υπενθυμίζουμε ότι στην τρέχουσα υλοποίηση το επιλεχθέν transport καθορίζεται με βάση το όνομα του vhost controller. Μια πιο κατάλληλη προσέγγιση θα ήταν η επέκταση της διεπαφής με την είσοδο μιας νέας παραμέτρου *–trtype* αποκλειστικά γι' αυτό το σκοπό. Τέλος, θα μπορούσαμε να επεκτείνουμε τη δομή των memory maps στο SPDK. Όπως έχουμε προαναφέρει στην ενότητα της υλοποίησης, η κλίμακα (granularity) των μεταφράσεων (ευθυγράμμιση στα 2MB) είναι ιδιαίτερα περιοριστική. Αυτό αφορά τόσο το μηχανισμό SPDK/VVU όσο και άλλες περιπτώσεις χρήσης του SPDK. Η προφανής λύση είναι η προσθήκη ενός επιπλέον επιπέδου κλίμακας 4KB στη δομή των memory maps. Ωστόσο, μια τέτοια αλλαγή προϋποθέτει συζήτηση με την κοινότητα.

1 Introduction

1.1 Purpose

The problem we are engaging with in this diploma dissertation is optimizing storage virtualization in the cloud. Nowadays, there is an ever increasing amount of applications running in the cloud in containerized environments. Many of these applications are stateful, which means that they need some kind of persistent storage to keep their data. More specifically, the high-performance cloud computing applications have raised great demands on the I/O performance of modern datacenters. Therefore, it turns out that the performance of the I/O datapath is becoming a significant factor in the cloud. Currently, there are various storage virtualization approaches. However, it seems that little effort has been put on optimizing those approaches in terms of performance, especially for the new high-performance storage devices like NVMe SSDs.

Nonetheless, optimizing storage virtualization is not just about improving the I/O performance. Another aspect of storage virtualization is security. The emulated storage devices are implemented in software. Therefore, they are prone to bugs that may be adding security holes in the system. Security is more important in cloud environments where multiple users have access to virtual machines running on the same physical machine.

Another problem with the existing storage virtualization solutions is that they are entirely configured by the cloud provider. In other words, the user has no control over his storage devices. Based on the type of the workload that he wants to run, the user orders beforehand from the cloud provider the types of emulated or physical disks that

he wants to offer to his VMs. This means that the end user has no control over the I/O datapath and he cannot rearrange his emulated hardware on the fly.

This is a feature that we will be referring to from now on as "flexibility". It would be a breakthrough if we could transfer the control of the emulated storage devices from the cloud provider to the end user. This means that the user would gain control of his storage. The user would have the flexibility to bring his own customized storage to his apps. In other words, the user would have full control of what his applications would see as storage devices and how the I/O processing would have been done by his storage software. This is what we call here "User Defined Storage". We strongly believe that this would unlock new capabilities in terms of storage management in the cloud.

In this diploma thesis, we are going to introduce a new storage virtualization solution that combines high performance, security and flexibility.

1.2 Motive

The technological evolution in the field of computer storage is revealing new potential as far as I/O performance is concerned. We have moved from the low spinning disk drives to the SATA solid state drives, then to the NVMe solid state drives (thus getting rid of the SATA controller latency by attaching the device directly to a PCIe endpoint) and recently to the NVMe solid state 3D crosspoint devices [1] (eg. Intel Optane SSDs). The new solid state media offer lower latency, higher throughput, higher parallelism against the old rotational hard disks. Unfortunately, the current software (kernel storage stack) is not making the full out of these new storage media, because it has been designed to work well regardless of the underlying storage media. Nonetheless, we have reached to a point where the software latency has become comparable to the hardware latency due to the evolution of hardware. So, it turns out that hardware evolution has created the need for restructuring the corresponding storage software. Reducing the overall I/O latency even further involves re-designing the software to meet the demands of the modern hardware.

What is more, the I/O virtualization overhead in Virtual Machines is critical in cloud services. In the cloud, users run their applications on top of emulated hardware. And nowadays, there is an ever increasing amount of high-performance cloud computing

applications (eg. machine learning), thus turning the I/O virtualization overhead into an important issue. However, judging from the existing storage virtualization solutions, it seems that the storage performance in virtualized environments has not been adequately improved. There is enough room for improvements in this field.

Finally, we argue that there is currently limited flexibility when it comes to IaaS in public clouds. The users cannot adjust the storage devices for their VMs. This is currently the cloud provider's responsibility. The reason is that there is no secure way for supporting this. Here, secure means that the end users should have access only to Virtual Machines.

1.3 Existing Solutions and SPDK/VVU

There are plenty of storage virtualization solutions and many ways to categorize them. Based on the different criteria, we have the following categories:

1. **Virtualization Techniques**

   - **full device emulation (trap and emulate)**:
     the device is emulated by the hypervisor (eg. QEMU) on host user space
   - **paravirtualization (VIRTIO [2])**:
     the device is still emulated by the hypervisor. However, the I/O performance is better, because the guest drivers have been optimized to produce less VMEXITs per I/O request, thus reducing the virtualization overhead.
   - **direct device assignment (passthrough) with vfio [3]**:
     the device is a physical device attached on the host PCI bus. The device is assigned to a virtual machine. The device's PCI resources (PCI Configuration Space, memory address space, port address space, interrupt vectors) are exposed directly to the guest system. This technique achieves near-native performance.
   - **mediated passthrough (mdev) [4]**:
     the device is exported as multiple virtual devices by the host kernel. The host kernel exposes as many physical units of I/O operations (eg. HW

I/O queues, MSI vectors) as possible directly to guests. This method is a combination of full device emulation and vfio-based passthrough. Its purpose is to combine the near-native performance of direct passthrough with the device sharing feature of full device emulation.

2. **Backend Device Emulation (I/O Datapath)**

   - **QEMU user space device emulation**:
     the I/O requests are emulated by the hypervisor and they are implemented as read/writes to a disk image, which is a file on the local host filesystem. So, the guest storage device is backed by a file on the host filesystem.
   - **kernel device emulation (vhost [5])**:
     the I/O datapath is offloaded from the hypervisor (QEMU). The device is emulated by the kernel vhost subsystem. Thus, we have an in-kernel I/O datapath, which means that the I/O datapath stays entirely inside the host kernel. There is no need for QEMU intervention, and thus there are no context switches from host kernel space to host user space. Vhost works with shared memory, that is it has full access to the whole guest physical memory.
   - **user space device emulation (vhost-user [6])**:
     the I/O datapath is offloaded from the hypervisor (QEMU) and implemented in a separate process on host user space. This mechanism is a variation of the in-kernel vhost implementation. Like vhost, vhost-user works with shared memory. Overall, this technique can perform better that the kernel storage stack if we replace the kernel with efficient user space drivers (this is what SPDK does)

3. **Storage Protocol**

   - **virtio-blk**:
     dedicated paravirtualized storage protocol. It bypasses the guest kernel SCSI subsystem.
   - **SCSI**:
     storage protocol used by many types of storage devices (eg. virtio-scsi devices)

- **NVMe**:

  performs better for non-volatile media by taking advantage of their inherent properties (low latency, high parallelism, high throughput)

In the following chapters, we are going to study a storage virtualization solution based on vhost-user protocol, SPDK and the virtio-vhost-user device. We call this new storage virtualization solution **"SPDK/VVU"**. The basic idea is to keep the user space device emulation, like in case of vhost-user, but perform the device emulation inside a dedicated Storage Appliance VM instead of host user space. This requires extending the vhost-user mechanism so that the device emulation software - in our case the SPDK - has still access to the VM's memory. This extension of the vhost-user machanism is implemented via the virtio-vhost-user device. We will also need to add some changes in the SPDK code-base to support this new communication mechanism.

What we get with SPDK/VVU is great I/O performance due to vhost-user and SPDK, better security because the device emulation is performed inside a containerized environment (that is a virtual machine), and flexibility in the sense that the Storage Appliance VM belongs to the end user, and thus the end user has full control of the dataplath. It is important to say that despite the fact that the I/O datapath goes through a Virtual Machine, the I/O performance is not reduced, because the aforementioned technique bypasses almost completely the hypervisor from the I/O datapath.

1.4 Structure of the diploma thesis

The rest of this diploma thesis is organized as follows:

- **Chapter 2**: this chapter resumes the background knowledge that is necessary to follow up with this diploma thesis

- **Chapter 3**: here we give an introduction to the SPDK framework. We give a brief description of its purpose, its architecture and its key components. The purpose of this chapter is to give the reader an overall understanding about the purpose of the SPDK framework and how it works.

- **Chapter 4**: in this chapter we introduce the vhost protocol. We emphasize on the user space implementation of vhost (also called "vhost-user"), because we are not interested in the kernel space implementation in this diploma thesis.

- **Chapter 5**: in this chapter we introduce the design of SPDK/VVU, our storage virtualization solution that relies on SPDK and the virtio-vhost-user device. We are going to explain how the virtio-vhost-user device extends the vhost-user mechanism, how the SPDK vhost code is organized and what we have to do in order to support this new setup.

- **Chapter 6**: this chapter describes the internals of the actual implementation with references to the code.

- **Chapter 7**: here we attempt an evaluation of SPDK/VVU as a cloud-oriented storage virtualization solution. We first explain the source of the virtualization overhead. Then, based on our previous conclusions, we make a comparison among the pre-existing storage virtualization solutions. Note that the comparative evaluation is constrained on qualitative criteria. In other words, we are deliberately not resorting on performance measurements in this thesis.

- **Chapter 8**: in this chapter we are assessing our current state and describing our next steps. We are also outlining some proposals for future improvements and enhancements on SPDK/VVU.

2 Background

2.1 Port I/O and Memory Mapped I/O

Port I/O (PIO) and Memory Mapped I/O (MMIO)[7] are two methods of performing I/O between the CPU and peripheral devices. They are collectively referred to as "Programmed I/O", meaning that these methods rely on CPU instructions in order to move the data from main memory to device memory, and thus occupying the CPU. The other alternative is Direct Memory Access (DMA), where devices perform I/O from/to physical memory without the CPU intervention.

From a software perspective, PIO is being performed by special-purpose instructions that operate on a separate port address space, which is distinct from the processor's memory address space. PIO is not supported by all architectures. While some CPU manufacturers implement a single address space in their chips, others have decided that peripheral devices are different from memory and, therefore, deserve a separate address space. Some processors (most notably the x86 family) have separate read and write electrical lines for I/O ports and special CPU instructions (eg. IN/OUT in x86) to access ports. Other processors, like ARM, do not have this functionality, hence exclusively relying on memory mapped I/O for device manipulation.([8] chapter 9)

On the other side, MMIO is being performed with the same instructions used for accessing the main memory. During the bootup process, the BIOS maps the PCI device memory regions to the processor's memory address space. Whenever a memory access is attempted by the CPU, the Northbridge redirects the memory access either to the main memory or to a specific device based on the instruction's source/target address.

2.2 PCI, PCI device resources

PCI ([9], [8] chapter 12) is a local computer bus for attaching hardware devices to a computer. PCI is standardized. It is a complete set of specifications defining how different parts of a computer should interact. The PCI architecture was designed as a replacement for the ISA standard, with three main goals: to get better performance when transferring data between the computer and its peripherals, to be as platform independent as possible, and to simplify adding/detecting/configuring/removing peripherals to the system (this utility is referred to as "Plug-and-Play" and is inherent to the PCI bus as opposed to older buses like ISA).

Each PCI peripheral is identified by a bus number, a device number, and a function number. The PCI specification permits a single system to host up to 256 buses, but because 256 buses are not sufficient for many large systems, Linux now supports PCI domains. Each PCI domain can host up to 256 buses. Each bus hosts up to 32 devices, and each device can be a multifunction board with a maximum of eight functions. Therefore, each function can be identified at hardware level by a 16-bit address, or key.

PCI devices have their own address spaces. Specifically, PCI devices have three address spaces: PCI memory address space, PCI I/O address space, and PCI Configuration space. The hardware circuitry of each peripheral board answers queries referring to these address spaces. The first two address spaces are shared by all the devices on the same PCI bus (i.e., when you access a memory location, all the devices on that PCI bus see the bus cycle at the same time). The configuration space, on the other hand, exploits geographical addressing. Configuration queries address only one slot at a time, so they never collide.

The PCI Specification defines that each PCI device has a PCI Configuration Space Header. The PCI Configuration Space Header is a set of device registers that allow the system to identify and control the device. The structure of the Configuration Space Header is standardized. The most important registers in the PCI Configuration Space Header are the Vendor ID, Device ID, Base Address Registers (BARs) and Capability Pointer.

The Vendor and Device ID registers identify the device, and hence they are used by the kernel to probe the corresponding device driver. The BARs are the interface for

accessing the device's memory regions and I/O ports. In specific, the BARs contain the physical addresses where the device's memory regions and I/O ports have been mapped in the processor's memory and port address spaces respectively by the BIOS. From the BAR's msb we can tell whether this BAR refers to PIO or MMIO. The BARs can also give us the size of each region. In specific, the device driver learns the size of each region by writing all ones to the BAR bits and reading back the register value. The PCI Configuration Space Header contains 6 BARs. However, there is an optional 7th BAR that can be used to point to the device's ROM. A PCI device may be equipped with a ROM which can contain driver code or configuration information. BIOS runs each device's ROM code during the POST [10] phase. Last but not least, the capability pointer points to the first capability in the device capability list. PCI capabilities allow more parts of the configuration space to be standardized. Each capability has one byte that describes the capability type, and one byte to point to the next capability. There are various types of capabilities, either generic ones (eg. MSI capability) or vendor-specific (eg. virtio capabilities [2]) ones. The full list of capabilities can be found here. The capabilities are often referring to configuration structures located in the device's memory address space or I/O address space. The capability pointers contain the physical addresses for those structures.

2.3 PCI Express

PCI Express (PCIe) [11] is a computer bus specification, designed to replace the older PCI, PCI-X and AGP bus standards. In fact, PCI express was designed to replace PCI while making sure it is software compatible with PCI. This means that older PCI software systems will still be able to detect and configure PCI express cards, although without the PCI express features. Apart from the software compatibility and the same usage model with respect to *read*/*write* I/O memory transactions, PCI express has made a number of improvements over PCI. PCI express uses serial bus technology whereas PCI is based on parallel bus. This reduces the number of I/O lines connecting the PCI express devices, thereby reducing the board cost and the board complexity. This also allows to increase the transmission frequency, thereby providing better throughput. PCI express also allows to scale bandwidth by providing support for multiple links. Links can be scaled by a factor of 2, 4, 12, 16 or 32. Links are made up of lanes. A x1

connection, the smallest PCIe connection, has one lane made up of four wires which implement a full duplex connection. A x1 connection can carry 1 bit per cycle in both directions. A x2 connection can carry 2 bits per cycle and so forth. Subsequently, x16 connections are capable of handling I/O intensive devices like graphics cards. PCI express has been designed only to be software compatible with PCI, so you can't have a PCI express card inserted into a PCI slot or a PCI card inserted into a PCI express slot.

The components in a PCIe topology are [12]:

- **Root Complex**: this is the host controller presented to the SoC. It connects the processor and memory to the PCIe devices. The root complex is integrated into the chipset. It provides slots, which are called **Root Ports**, using which other PCIe devices can be connected to the system.
- **PCIe Endpoint**: this term refers to each PCIe card connected to the PCIe transport. Usually, it is either a controller (SATA, SCSI, USB) that has a device connected to it or a device directly attached to the PCIe transport (eg. NVMe disk, graphics card).
- **PCI express address space**: the address space for the PCI express bus. It can be either 32 bit or 64 bit depending on the root complex. This address space is visible only to the PCI express components like the root complex, the endpoints, the switches and the bridges.
- **Bridge**: component used to connect a PCI or a PCI-X device to a PCI express root complex
- **Switch**: component used to connect multiple PCI express devices to the root complex. Usually, a switch is being used if there are not enough slots present in the board.

The topology goes as follows:

The root complex connects the CPU with the rest of the PCI express devices. It is part of the chipset (also called "Host Bridge" or "North Bridge"). In modern CPUs, where the host bridge is integrated inside the SoC, the root complex is part of the SoC as well. The CPU uses the root complex to communicate with any of the PCI express

devices and the root complex can interrupt the CPU for any of the events generated by the root complex itself or any of the PCI express devices. The root complex can also directly access the memory without CPU intervention. Root complex offers a set of "root ports" where other PCI express devices can be connected.

PCI express uses point-to-point topology, which means a single serial link to connect any pair of devices. Each link is full-duplex and can be made of multiple lanes, thus offering greater bandwidth. Each lane is a data stream that allows for a bidirectional bit transfer per cycle.

Inside the root complex, there is a host bridge that connects the CPU to the multiple root ports. These root ports are nothing but "virtual PCI-to-PCI bridges". These bridges are connected to the root bus spawned by the host bridge which is "Bus 0". Each virtual PCI-to-PCI bridge spawns a new PCI bus using which other PCI express devices can be connected.

Similarly, in the case of a switch, the point-to-point topology is achieved by using multiple PCI-to-PCI bridges. There is one virtual PCI-to-PCI bridge for the "upstream port" of the switch. The rest are used for the "downstream ports" of the switch.

An abstract overview of a PCI Express topology would look like this:

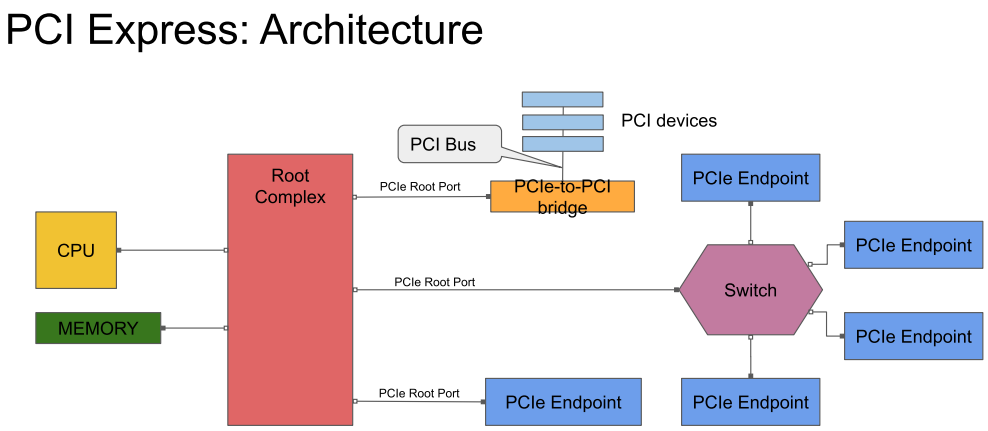


Figure 2.1: *PCI Express Topology*

Each bus is assigned a number by the BIOS during the bus scan and enumeration process. These bus numbers are used by bridges or switches in order to root the transac-

tions. So, each bridge or switch will have information about three bus numbers. One is the "primary bus number" which tells the bus to which this bridge/switch is connected. The "secondary bus number" is the lowest bus number which can be reached through this bridge. The "subordinate bus number" is the highest bus number that can be reached through this bridge.

PCI express also does not use dedicated IRQ lines, unlike PCI which has four dedicated IRQ lines. It uses "inband signaling" even for interrupts. These are called "Message Signaled Interrupts" or "MSI" in short. However, PCIe also supports a Legacy interrupt mechanism [13].

PCIe offers two important features that concern this diploma thesis:

1. the device transactions are tagged with a Requester ID. This means that the root complex knows from which device the request came from. This is possible in PCIe due to the point-to-point connection against the bus topology in PCI. This feature enables creating distinct IOVA domains for each PCIe device, and thus achieving device isolation. Device isolation in terms of DMA transactions is necessary for secure device passthrough in virtual machines [14].

2. Address translation Services (ATS). ATS enables the PCIe Endpoints to request the DMA address translations from the IOMMU and cache the translations, thus alleviating IOMMU pressure and improving the hardware performance in the I/O virtualization environment. In short, it is a cache for IOMMU translations [15].

2.4 QEMU/KVM

QEMU [16] stands for "Quick Emulator". It is a full system emulator meaning that it can emulate a full computer system, including CPU, memory and peripheral devices. However, it can also be used as a CPU emulator.

QEMU can be used to run machine code compiled for any architecture to a different architecture. For example, we can run MIPS code on x86 host CPU. This is achieved with dynamic binary translation (also called Just-in-Time compilation) performed by QEMU's Tiny Code Generator (TCG) [17].

In case we want to run x86 code on x86 architecture, then it would be an overkill to use the TCG. In such case and given that the CPU has virtualization features (eg. Intel VT-x[78], VT-d[55]), we can use the KVM kernel module to run x86 code natively on the host CPU.

KVM [18] [19] stands for "Kernel Virtual Machine". It is a hypervisor that takes advantage of the hardware virtualization extensions present in modern Intel and AMD CPUs for safely executing guest code directly on the host CPU, thus achieving near-native performance. KVM emulates some hardware components like the MMU (shadow mmu) and the interrupt controllers (8259 PIC, APIC, IOAPIC, LAPIC). It also handles the guest processor state (Virtual Machine Control Structure or VMCS in x86) and injects virtual interrupts to the guest in case of no hardware assistance. However, KVM is not a full system emulator in the sense that it can not emulate peripheral devices. This is why it is commonly used alongside QEMU. KVM uses the CPU's virtualization extensions in order to **safely** run **guest** code **natively** on the host CPU and relies on QEMU to do the device emulation.

From an API standpoint, KVM is a kernel module. The KVM API consists of a set of ioctls [20] [21] that are issued to control various aspects of a virtual machine. The ioctls are issued on file descriptors. The primary KVM interface is the character device file */dev/kvm/*. There are also some ioctl types that return other file descriptors. In general, the KVM API consists of a set of ioctls that involve spawning a new VM, registering its memory, registering PIO and MMIO PCI regions, registering its virtual CPUs, creating interrupt controllers (or irqchips in KVM terminology), injecting virtual interrupts to the guest, starting executing guest code, etc.

In hardware-assisted virtualization, which we are considering here, the privileged CPU instructions (like *IN/OUT/LOAD/STORE*) and privileged CPU operations (like writing to CR3 register) are emulated by the hypervisor (KVM) [22]. This means that the CPU runs guest code natively but traps whenever a privileged instruction is executed. At the same time, running host code (including privilege instructions) should nevel trap. This differentiation is implemented in hardware with the privilege levels. In general, the convention is that privilege level 0 corresponds to host code and privilege level -1 corresponds to guest code. In x86 architecture with VT-x support, the ring 0 privilege level is called **"root mode"** and the -1 privilege level is called **"non-root mode"**.

In fact, x86 does not only distinguish between root mode and non-root mode, but also between root user-space/kernel-space and non-root user-space/kernel-space. Otherwise, running guest OS's system call handlers would be impossible (system call emulation is another kind of virtualization design). x86 architecture offers two instructions for switching between root mode and non-root mode. These are ***VMXON*** and ***VMXOFF***. The *VMXON* command switches the CPU state into non-root mode, while the *VMXOFF* command switches the CPU state into root mode. The corresponding operations of switching between root node and non-root mode are called ***VMEXIT*** and ***VMENTRY*** respectively.

Given all the above information, lets see how QEMU/KVM works under the hood [23]:

Prior to booting up the VM, QEMU has to map either the BIOS code or directly the guest kernel code (removing guest BIOS completely) into the guest memory address space. This is accomplished by first mapping the code into QEMU's process address space and then using the *KVM_SET_USER_MEMORY_REGION* KVM ioctl in order to register this memory as a guest physical memory slot. QEMU accommodates each guest physical memory slot with a guest physical address, which is where the KVM will map this memory in the guest CPU memory address space. QEMU then creates a set of vCPUs with the *KVM_CREATE_VCPU* ioctl. Each vCPU corresponds to a separate thread on the host. Before we can run code, we need to set up the initial states of the vCPU register sets. This is done via the *KVM_GET_SREGS* and *KVM_SET_SREGS* ioctls. QEMU then instructs KVM to start running guest code natively with the ***KVM_RUN*** ioctl. In response to this ioctl, the KVM runs the *VMXON* CPU instruction (this insruction is x86-specific, but a similar one exists for other architectures with HW virtualization features). This is a virtualization-specific instruction that tells the CPU to switch to non-root mode and start running guest code. When the guest OS performs a privileged operation, the CPU will exit to the VMM code (KVM handler) on host kernel space. This switch between root mode and non-root mode is generally called a **"world switch"**, because it requires changing the whole CPU state. KVM takes over and finds the exit reason from the VM's control structure (VMCS) registers. The VMCS is where the guest CPU state is saved during the world switch. If the KVM can service the request itself, it will do so, and give control back to the guest. This is called a **"lightweight VMEXIT"**. For requests that the KVM code can't serve, like any device

emulation, it will defer to QEMU. This implies exiting to host user space from the host Linux kernel, and hence this is called a **"heavyweight VMEXIT"**. QEMU finds the exit reason from a KVM's data structure and performs the device emulation. It then switches back to KVM with *KVM_RUN* ioctl and the KVM in turn switches back to guest code with the *VMXON* CPU instruction.

To sum up, the basic flow of a guest CPU in a simplified form, goes like this:

```
1  open("/dev/kvm")
2  ioctl(KVM_CREATE_VM)
3  ioctl(KVM_CREATE_VCPU)
4  for (;;) {
5       ioctl(KVM_RUN)
6       switch (exit_reason) {
7       case KVM_EXIT_IO:  /* ... */
8       case KVM_EXIT_HLT: /* ... */
9       }
10 }
```

2.5 Event File Descriptor

eventfd [24] is a notification mechanism offered by the Linux kernel that can be used by user space applications and by the kernel to notify user space applications of events.

It works as follows:

the kernel allocates a 64-bit counter in kernel space memory for each created eventfd. Every user space application that has access to the event file descriptor can read or write a value to this counter via *read()*/*write()* system calls. The side effects of *read()* and *write()* depend on whether the eventfd counter currently has a nonzero value and whether the *EFD_SEMAPHORE* flag was specified when creating the event file descriptor.

Applications can use an eventfd instead of a pipe in all cases where a pipe is used simply to signal events. The kernel overhead of an eventfd is much lower than that of a pipe, and only one file descriptor is required (versus the two required for a pipe).

A key point about an eventfd is that it can be monitored just like any other file descriptor using *select()*, *poll()* or *epoll()*.

2.6 ioeventfd/irqfd

ioeventfd and irqfd are two KVM mechanisms that can be used in order to produce lightweight *VMEXITs*.

The ioeventfd mechanism binds a guest I/O port to a host eventfd. Whenever a *VMEXIT* happens due to a guest *read*/*write* on that port, KVM kicks the eventfd and immediately resumes executing guest code with the *VMXON* command.

The irqfd mechanism binds a host eventfd to a guest irqchip pin. KVM polls on this eventfd. Whenever the eventfd is kicked, KVM injects a virtual interrupt to the guest.

There are two ioctls in the KVM API for these two mechanisms. These are "*KVM_IOEVENTFD*"[20] and "*KVM_IRQFD*"[20]. The eventfds are allocated by the user space process (QEMU) and passed to KVM through a proper data structure. QEMU hooks up those eventfds to specific guest I/O ports and guest IRQs respectively.

These mechanisms are used both by the vhost[5] and vhost-user devices in order to reduce the software latency in the I/O datapath. They are also used in cases where we want to offload the device emulation to QEMU IOTHREADS[25] instead of the main QEMU event loop thread (eg. virtio-blk/virtio-scsi dataplane).

2.7 DMA

DMA[26] stands for "Direct Memory Access". It is the action where a device accesses the system memory or another device's MMIO space without the intervention of the CPU. This means that we do not waste CPU cycles for data transfer among devices and physical memory. This is a great benefit in comparison with the conventional method of "CPU I/O" or "Programmed I/O", in which the CPU is doing the data transfer from the system memory to the device memory and vice versa. DMA is the preferred method for data transfer in case of I/O intensive devices like Gigabit Ethernet cards, PCIe NVMe SSDs, GPUs, etc.

In the past, devices could not perform DMA on their own. There was a separate device called "DMA Controller" or "DMA engine" for this purpose. At present, most modern devices enclose a dedicated DMA controller inside the PCI card.

The memory from which a device is programmed to perform DMA has to follow some limitations[27][28]:

- the memory has to be **pinned**. This implies two things. The first is that it must never get swapped out by the kernel memory management subsystem. The second is that it must never get transferred to another physical frame, that is change the VA-to-PA translation. This could be done for example by the kernel memory management subsystem during memory compaction due to external memory fragmentation, or in case of memory hotplug, where the load has to be rebalanced.

- the memory has to be **assigned to a physical frame** by the kernel before the DMA operation takes place. This makes sense for user space memory buffers which get allocated with the *mmap()* system call. The kernel follows the demand paging strategy for user space memory. This means that the memory gets allocated after the first page fault, which is triggered when this memory is accessed for the first time. This has to be done before instructing the device to perform DMA from this user space buffer. Here, we assume that this is a separate limitation from pinning, though it could be regarded as one.

- the memory has to be **physically contiguous**. A physical device understands physical addresses. It cannot perform DMA from a memory buffer that is scattered into multiple discontiguous physical 4KB frames. To be more precise, a single I/O operation from the device perspective has to correspond to a physically contiguous memory segment. However, some modern devices support doing multiple I/Os from physically scattered memory buffers as part of a single DMA operation.

- the memory has to be **aligned**. This is a rather device-specific limitation. Devices have certain limitations regarding the buffer alignment. For example, the NVMe specification[51] requires all physical memory to be describable by what

is called a "PRP list". PRP lists have certain limitations regarding the address alignment.

The last two limitations hold in cases where there is no IOMMU in the system. Otherwise, the IOMMU satisfies these two limitations. (See [14] for the third limitation)

A memory region that satisfies the above limitations is often called **"DMA-able"** memory.

Except for DMA operations from system memory, it is also possible for DMA operations to be routed to the MMIO PCI resources of another device on the same PCI bus. This is called **"peer-to-peer DMA"**. Generally, peer-to-peer DMA is a root of problems as far as device isolation is concerned. This happens because the DMA operations may not reach the Northbridge, where the IOMMU is integrated, thus being transparent to the chipset.

2.8 VFIO

Vfio[3] is a kernel device driver that enables creating user space device drivers. It does so by exposing all the PCI device resources (PCI Configuration Space, PCI device's memory address space, PCI device's port address space, interrupts) to user space. So, vfio enables I/O from user space, both Programmed I/O and DMA, in a secure, IOMMU protected environment.

Vfio has been designed as a replacement for the pre-existing uio[29] framework. Uio has serious limitations like no IOMMU protection, limited interrupt support, and requires root privileges to access things like PCI Configuration Space. Vfio attempts to solve all these limitations. However, vfio also supports bypassing the IOMMU or working with systems without IOMMU, thus supporting unrestricted DMA.

Interface for VFIO

Vfio exposes each device's resources through character device files in */dev/vfio/*. The device resources (PCI Configuration Space, PCI MMIO BARs, PCI PIO BARs) correspond to offsets inside the device file. A user space process can interact with these

resources by reading, writing or mmapping the corresponding segments in the device file. Other device properties are discovered via ioctls. Vfio notifies user space about device interrupts through eventfds, which are bound to device interrupts. Eventfds are also configured via an ioctl. The vfio-pci driver supports all the PCI interrupt mechanisms, that is legacy INTx interrupts, MSI interrupts and MSI-X interrupts. Note that the interrupts are not configured by direct manipulation of the interrupt related registers in PCI Configuration Space (eg. Interrupt Pin) or MMIO space (eg. MSI-X vector table). Interrupts are configured via ioctls[32].[30][31]

Enabling DMA from user space buffers

Before examining how vfio satisfies the DMA limitations, it is important to understand the difference between CPU and DMA addresses[33]. The CPU uses physical addresses in order to manage the physical memory and the device resources. On the other hand, I/O devices use a different kind of addresses: "bus addresses". If a device has registers at an MMIO address, or if it performs DMA to read or write system memory, the addresses used by the device are bus addresses. In some systems, bus addresses are identical to CPU physical addresses, but in general they are not. IOMMUs and host bridges can produce arbitrary mappings between physical and bus addresses. From a device's point of view, DMA uses the bus address space, but it may be restricted to a subset of that space. For example, even if a system supports 64-bit addresses for main memory and PCI BARs, it may use an IOMMU so devices only need to use 32-bit DMA addresses. In some simple systems, the device can do DMA directly to the physical addresses that the driver gives. But in many others, there is IOMMU hardware sitting between the devices and physical memory that translates DMA addresses (bus addresses) to physical addresses.

The vfio has to satisfy the DMA memory limitations for the user space DMA buffers. In case of x86 architecture with an IOMMU (this is called Type1 IOMMU), vfio offers an ioctl type called *VFIO_IOMMU_MAP_DMA*, which is used by user space drivers to register the user space memory that will be used for DMA. With this ioctl, the user space driver passes to the kernel a contiguous range of virtual addresses corresponding to a user space buffer allocated from the process' heap (eg. with *malloc()* system call). It also passes an IOVA address (I/O virtual address), which will be used by the kernel

to program the IOMMU. The vfio driver cannot change the suggested IOVA address. It will just fail in case it is not available. The vfio serves this ioctl by basically doing two things:

- pin the registered user space memory. This means that this memory will never get swapped out or moved to another physical address. This is done here: [1]
- program the IOMMU using the IOVAs passed from the user space driver. The kernel IOMMU driver will insert the appropriate entries in the device IOVA domain. This implies that the user space driver can use IOVAs instead of physical addresses for the DMA operations. In case the DMA buffer is scattered in physical memory, vfio will arrange the IOMMU entries in such a way that the device will be seeing this memory as contiguous. This means that the registered memory, although it might be physically scattered, it will be mapped to a contiguous IOVA segment. This is done here: [2]

  The number of IOVA entries needed depends on the physical segments making up the user space buffer. The user space buffer is possibly sparse in physical memory. So, for each physical memory segment, a new IOMMU entry, mapping the physical address to an IOVA address, has to be inserted.

VFIO in no-IOMMU mode

Vfio currently supports running on systems without an IOMMU. This is called "no-IOMMU mode". This mode of operation can be chosen by using a kernel parameter when we insert the vfio module into the kernel. The no-IOMMU mode has a major difference with the Type1 IOMMU (x86 IOMMU). There is no IOMMU to do the address translation (or we just don't use it). This means that the user space driver has to pass physical addresses instead of IOVAs to the device. The vfio driver in no-IOMMU mode does not support the *VFIO_IOMMU_MAP_DMA* ioctl. This means that we lose some benefits given by vfio like page pinning and mapping scattered PAs to contiguous IOVAs. This implies that the user space driver cannot use regular heap

[1] https://elixir.bootlin.com/linux/latest/source/drivers/vfio/vfio_iommu_type1.c#L1046

[2] https://elixir.bootlin.com/linux/latest/source/drivers/vfio/vfio_iommu_type1.c#L1055

memory as DMA buffer, because it does not follow the DMA memory limitations. It turns out that the only solution (at least for now) is using hugepages.

The hugepages satisfy the DMA memory limitations as follows:

- hugepages by definition are pinned in physical memory. From the moment they are allocated in physical memory, they never get swapped out or moved to another location in memory. This requires that we are not talking about transparent hugepages, which have recently been added to the Linux kernel and enable swapping out hugepages.
- older kernel versions used to prefault the hugepage memory during allocation with *mmap()* system call. This comes in contrast to regular anonymous mappings (mmap with *MAP_ANONYMOUS* flag), which are assigned to 4KB pages in the page cache after the first page fault (this is called demand paging). However, this has changed in recent kernels, which follow the demand paging strategy for the hugepages as well. So, in order to use hugepage backed memory buffers, the hugepages have to be "touched" once.
- hugepages are physically contiguous. For example, a 2MB hugepage will reside in a contiguous 2MB physical memory range. In other words, a 2MB hugepage is made of 512 consecutive physical 4KB memory frames.
- the address alignment limitation is device-specific and hugepages know nothing about this. It is the responsibility of the driver to place the buffer in aligned addresses inside the hugepage backed memory.

For more information about the hugepages, read the following section.

2.9 Hugepages

Description and Characteristics

Modern CPU architectures support multiple page sizes. Hugepages are contiguous chunks of physical memory that are bigger than normal pages (like 4KB pages in x86).

The size of hugepages depends on the architecture. For example, x86 CPUs support 2MB and 1GB hugepages.

Hugepages have three key features[27]:

- they are **physically contiguous**. This means that a 2MB hugepage will correspond to a contiguous 2MB physical memory segment. This feature allows for the use of a single page table entry for the whole 2MB memory region. Therefore, one TLB cache line corresponds to 2MB physical addresses, thus reducing the number of cache misses.

- they are **non-swappable**. This means that upon getting allocated on physical memory after the first page fault (due to demand paging), the kernel memory management subsystem will never swap them out.

- the **VA-to-PA translation is fixed**. This means that the kernel will never move a hugepage to a different location in physical memory. On the contrary, this is something common for regular 4KB pages. Given that a hugepage's VA-to-PA translation cannot change due to a swap-out/swap-in operation because of the above feature, the kernel moves physical frames around in two scenarios:

    1. as a means of reducing external memory fragmentation. External memory fragmentation is a state of physical memory where there is free memory but it is scattered in distinct segments rather that being contiguous. This means that if a process wants to allocate a 1GB hugepage and there is enough free memory but it is scattered, the hugepage allocation will fail. It is caused naturally by the fact that multiple processes on a system allocate and deallocate memory of various sizes. It is solved by a procedure called "memory compaction". Memory compaction involves moving occupied physical memory frames to a different location in memory. This implies that their physical address will change and this requires updating the page tables of the relevant processes.

    2. as a way to uniformly distribute the physical frames in case additional physical memory is hotplugged in the system.

Advantages and Purpose

The advantages of hugepages against normal 4KB pages are the following[34][35]:

- the contiguity of hugepages allows for greater performance due to less translations. Current platforms use multi-level page tables in which we can have a single page table entry for a whole 2MB hugepage. This means that there is need for only one TLB entry for a hugepage instead of 512 TLB entries for 2MB of regural 4KB pages. This implies less cache misses, and thus greater performance in terms of address translations.

- the TLB miss will run faster, because it requires walking three levels in the process multi-level page table[36] instead of four, which holds for ordinary 4KB pages (assuming 4-level page tables). Thus we achieve lower TLB miss overhead.

- second and third features listed above are known together as "page pinning". There is currently no other way in Linux to do pinning from user space without the assistance of the kernel. There is a relevant concept called "page locking" but this is different from pinning. "Page locking" is basically the non-swapping property and there is a system call for that called "mlock"[37]. But page locking doesn't guarantee fixed VA-to-PA translations. There is currently no other way except for hugepages to do page pinning and consequently to do DMA from user space memory without the help of the kernel (eg. vfio). In short, hugepages are the only way of allocating DMA-able memory from user space, and thus creating user space drivers without the kernel intervention.

There is a controversy as to whether the non-swapping feature is useful. Hugepages are mostly used for the first advantage. That is greater performance due to less address translations. There is a claim that non-swapping property is not useful. Thus, a new feature has been developed which is called "transparent hugepages"[38]. This feature is quite new and allows for the hugepages to get swapped out. It works by dynamically gathering adjacent normal pages into a single hugepage and splitting a hugepage into hundreds of normal pages when swapping it out. Transparent hugepages where created at first for use by KVM in order to decrease the memory virtualization overhead[19].

Reasons for adoption by SPDK and user space drivers in general

SPDK relies either on uio or vfio in order to access the PCI device resources directly from user space. SPDK manages mostly storage devices, which perform DMA operations for data transfers. Enabling DMA operations from user space memory requires that this memory is DMA-able or, in other words, pinned. That means that it will never get swapped out or moved to another physical location in memory. So, SPDK needs to somehow allocate pinned memory from user space. As mentioned above, there is a POSIX system call that satisfies the first limitation and is called "mlock". But it does not satisfy the second one. The only available solution (at least for now) is hugepages. At this point, it is important to point out that using hugepages to achieve page pinning is only necessary in case of uio. On the contrary, in case of vfio, the vfio driver itself performs the page pinning. The user space process just declares a portion of its virtual memory as DMA-able with the *VFIO_IOMMU_MAP_DMA* ioctl and the vfio driver performs the necessary actions to make this memory DMA-able[39].

Except for user space drivers, hugepages are also widely used in Virtual Machines in order to minimize the memory virtualization overhead[40].

Interfaces for using hugepages in Linux

From an API standoint, Linux offers plenty of explicit interfaces for hugepages to user space. The kernel does not use hugepages transparently. The most commonly used interfaces for using hugepages are the following[41]:

1. shared memory

   Hugepages are requested by using the *shmget()* system call. In specific, hugepages are used by specifying the *SHM_HUGETLB* flag and ensuring the *size* is hugepage-aligned. A limitation of this interface is that only the default hugepage size can be used.

2. hugetlbfs

   hugetlbfs is a filesystem in which all files are backed by hugepages. It is similar to tmpfs in the sense that it is a RAM-based filesystem, but instead of using

4KB pages, it uses hugepages. Another difference between tmpfs and hugetlbfs is that hugepages are not dynamically allocated in physical memory, in contrast to regural pages. This means that the user, except for mounting the hugetlbfs on the root filesystem, also has to manually pre-allocate the number of hugepages he is going to use into the hugepage pool. This is done through the procfs (*/proc/sys/vm/nr_hugepages*).

3. anomymous *mmap()*

   As of kernel 2.6.32, support is available that allows anonymous mappings to be created backed by huge pages with *mmap()* by specifying the flags *MAP_ANONYMOUS|MAP_HUGETLB*.

2.10 VIRTIO

VIRTIO[2] is a paravirtualized driver specification. It provides a common framework for I/O virtualization that can be used by any type of device. VIRTIO defines how drivers and emulated devices should be implemented in order to get greater performance than full device emulation. The purpose of VIRTIO is that the guest should continue to see a PCI device, but the driver and the device emulation are optimized in a way that minimizes the operational overhead of the device. In other words, VIRTIO aims at improving the host-guest interaction. This basically involves minimizing the number of VMEXITs triggered by PIO or MMIO compared to a fully emulated device.

Reading from the Virtio specification:

> The purpose of virtio and the virtio specification is that virtual environments and guests should have a straightforward, efficient, standard and extensible mechanism for virtual devices, rather than boutique per-environment or per-OS mechanisms.

- Straightforward:

  Virtio devices use normal bus mechanisms of interrupts and DMA which should be familiar to any device driver author. There is no exotic page-flipping or COW mechanism: it's just a normal device.

- Efficient:

  Virtio devices consist of rings of descriptors for both input and output, which are neatly laid out to avoid cache effects from both driver and device writing to the same cache lines.

- Standard:

  Virtio makes no assumptions about the environment in which it operates, beyond supporting the bus to which device is attached. In this specification, virtio devices are implemented over MMIO, Channel I/O and PCI bus transports, earlier drafts have been implemented on other buses not included here.

- Extensible:

  Virtio devices contain feature bits which are acknowledged by the guest operating system during device setup. This allows forwards and backwards compatibility: the device offers all the features it knows about, and the driver acknowledges those it understands and wishes to use.

The virtio spec defines that the virtio drivers follow a split-driver model[42]. There is a frontend driver in the guest kernel and there is a backend driver in the hypervisor (eg. QEMU). Data transfer is done through virtqueues. Virtqueues are queues that allow for efficient host-guest communication between the frontend and the backend drivers. The virtqueues are being allocated by the virtio frontend driver in physically contiguous guest memory. The virtio frontend driver informs the virtio backend driver about the location of the virtqueues in guest memory via a transport-specific mechanism (eg. via a virtio PCI capability in case of the PCI transport). Each virtqueue consists of three parts: Descriptor Table, Available Ring, Used Ring. The data exchange between the frontend and the backend driver goes as follows: the frontend driver inserts data in the virtqueues in the form of scatter-gather lists. This essentially involves filling in new descriptors with the guest physical addresses of the I/O buffers and pointing an equal amount of available ring entries to the corresponding descriptor indexes. The backend driver tracks down the new descriptors via the available ring and eventually handles the actual data located in guest memory. Whenever finishing with some data processing, the backend driver notifies the frontend driver about the amount of work that has

been done via the used ring. In specific, the backend driver writes the corresponding descriptor indexes into the used ring. As far as the host-guest notification mechanism is concerned, the frontend driver notifies the backend driver for new data by kicking a doorbell register. On the other side, the backend driver notifies the frontend driver that the request has been completed by injecting a virtual interrupt into the guest.

Paravirtualized drivers are aware that the hardware is emulated. This allows for the drivers to be designed in a way that takes into consideration the performance differences between emulated and physical devices. These differences are focused on three basic operations. The first is device memory access. In case of emulated devices, accessing device memory has a notable performance penalty because it triggers a VMEXIT and it has to be emulated by the hypervisor. This is the reason why virtio defines that the virtqueues must be allocated in guest physical memory rather that in device memory. In general, the fully emulated devices cause a larger number of VMEXITs compared to their paravirtualized counterparts. Secondly, in case of emulated devices, DMA is faster than in case of physical devices, because DMA is emulated in software. So, allocating the virtqueues in guest memory instead of device memory makes no difference in the performance from a device perspective. Thirdly, the host-guest notifications (guest notifications via doorbells and device interrupts) impose greater overhead in case of emulated devices in comparison to physical devices. Therefore, paravirtualized devices/drivers are designed in the direction of minimizing/batching the host-guest notifications.[43][44][45]

The VIRTIO Specification has gone through two revisions. The first implementation was written as a draft spec by Rusty Russel[3]. Then, is was standardized by the OASIS committee into an official specification. The first official version was 1.0. Recently, at the end of 2018, version 1.1[2] has been launched. This version incorporates some changes focused on performance.

2.11 SCSI

Small Computer System Interface (SCSI)[46] is a set of standards for physically connecting and transferring data between computers and peripheral devices. SCSI is both

[3] https://www.ozlabs.org/~rusty/virtio-spec/virtio-paper.pdf

a command set specification and a parallel peripheral bus hardware specification for physically attaching peripheral devices to a computer system. It is a common misconception that SCSI is a protocol for the control of hard disks. SCSI can be used for other device types as well. For example, tape drives, optical drives, printers, etc.

Many modern device controllers use the SCSI command set as a protocol to communicate with their devices through many different types of physical connections. In SCSI language, a bus capable of carrying SCSI commands is called a "transport", and a controller connecting to such a bus is called a "host bus adapter" (HBA). Evolving from the initial SCSI bus specification, the SCSI protocol has been standardized to work over various interconnects/transports. These specifications are collectively called the "SCSI Transport Protocols". There are specifications for SCSI over Parallel Bus (Parallel SCSI), SCSI over Serial Bus (SAS), SCSI over TCP/IP (iSCSI), SCSI over FibreChannel, SCSI over PCIe, SCSI over ATA, etc.

There are two types of devices on a SCSI bus[47]:

- SCSI initiators: SCSI devices that originate SCSI commands transmitted through an initiator port or a target/initiator port.
- SCSI targets: SCSI devices containing logical units that service commands received through a target port or a target/initiator port.

The SCSI Architecture Model [48] adopts a client-server communication model. SCSI initiators act like clients and SCSI targets act like servers. Initiators send Command Descriptor Blocks (CDBs) to the Targets over the underlying interconnect. The targets execute the received CDBs and return the appropriate response. Targets can be subdivided into several Logical Units (LUNs).

From a software perspective, the Linux SCSI Subsystem is responsible for handling the SCSI devices in the system. The SCSI Subsystem lies underneath the Block Layer in the kernel storage stack, thus handling bio requests passed down by the Block Layer. The SCSI subsystem uses a three layer design, with upper, mid, and low-level layers. Every operation involving the SCSI subsystem (such as reading a sector from a disk) uses one driver at each of the 3 levels[49]: one upper layer driver, one lower layer driver, and the SCSI midlayer.

The SCSI upper layer provides the interface between user space and the kernel, in the form of block and character device nodes for I/O and *ioctl()*. The SCSI lower layer contains drivers for specific hardware devices.

In between is the SCSI mid-layer, analogous to a network routing layer such as the IPv4 stack. The SCSI mid-layer routes a packet based data protocol between the upper layer's */dev* nodes and the corresponding devices in the lower layer. It manages command queues, provides error handling and power management functions, and responds to *ioctl()* requests. It is also responsible for bus scanning, device enumeration and tag allocation.

Except for NVMe devices, all other kind of storage devices are treated as SCSI devices by the Linux kernel. This means that the I/O requests pass through the SCSI Subsystem. The SCSI low-level driver is responsible to do the translation from the SCSI protocol to another storage protocol. For example, libata transforms SCSI commands into ATA commands[50]. This is called "SCSI Emulation". The reason that the kernel treats all devices like SCSI devices is that the Linux SCSI Subsystem is well-designed and very efficient. So, it seems like a good idea to use it for other protocols like SATA, USB, etc.

2.12 NVMe

NVM Express (NVMe) is a storage protocol[51] designed to exploit the inherent parallelism of modern solid state media.

Modern solid state media are inherently highly parallel and low latency devices. The SATA(AHCI) and SCSI interfaces cannot meet the demands of modern storage devices. The NVMe protocol solves the latency problem by directly attaching the SSDs to PCIe endpoints and thus bringing the device closer to the CPU. This results in lower I/O latency, because we get rid of the intermediate controller's overhead (eg. SATA, SCSI, etc.) in the CPU↔device communication. NVMe also requires less MMIO operations per I/O request in contrast to older specifications like AHCI, thus saving CPU cycles. In addition, NVMe results in lower OS software latency, because the I/O requests targeted to NVMe devices bypass the SCSI subsystem in the kernel storage stack. What is more, the NVMe protocol exports the high-parallelism of the modern storage

media. Each NVMe device can support up to 65535 I/O queues with 65535 pending I/Os each. The kernel takes advantage of the device parallelism with the multiqueue block layer (knows as blk-mq[52]). The multiqueue block layer maintains multiple lockless software queues - one per core - and multiple dispatch queues that match the device's HW queues.

The NVMe I/O datapath goes through a set of I/O queues. In I/O operations, commands are placed by the host software into the Submission Queue (SQ), and completion information received from SSD hardware is then placed into an associated Completion Queue (CQ) by the controller. A SQ and a CQ form a "queue pair" (QP). I/O queue pairs are allocated in host memory by the NVMe driver. This is used for most NVMe controllers. Some NVMe controllers, which can support Controller Memory Buffer, may put I/O queue pairs at controllers' PCI BAR space.[53]

Last but not least, NVMe can work over various transports. It can work over local connections (NVM over PCIe or NVMe for short) or over remote connections (NVM over Fabrics or NVMf for short). Remote connections include fabrics like Ethernet, FibreChannel, etc.

2.13 IOMMU

IOMMU is a hardware component that works as a layer of indirection between the I/O devices and the physical memory. The I/O devices understand only virtual addresses that are called IOVAs and perform DMA from these addresses. The IOMMU sits between devices and physical memory and makes the translation of I/O virtual addresses to physical addresses in the CPU physical address space. Thus, whenever a device performs a DMA operation on an IOVA, the IOMMU transparently translates the IOVA to a physical address, thus redirecting the memory access to the right memory address. In modern systems, the IOMMU hardware component is part of the PCIe Root Complex (which is part of the host bridge)[54]. Intel incorporates this functionality in the modern chipsets along with other virtualization features which are collectively referred to as "Intel Virtualization Technology for Direct I/O (Intel VT-d)"[55]. Intel uses the term "DMA Remapping (DMAR)" for the IOMMU.

The IOMMU works in the same way as the MMU in the CPU. It can be seen as an MMU

for I/O. The MMU exposes a virtual address space per process in order to control the memory accesses made by each process. Similarly, the IOMMU exposes an I/O virtual address space per I/O device in order to control the DMA accesses of each I/O device.

The IOMMU is useful for two reasons. Firstly, it protects the memory against arbitrary DMA operations made by a malicious I/O device (eg. an external device connected through a Thunderbolt port) or a malicious driver or even a buggy driver. This is called the **"translation property"**. Note though that this works in page granularity. Restricting access to a part of a page is possible with another feature called "bounce buffers"[56]. Secondly, it can expose a different I/O virtual address space to each I/O device. This is a property that has been made possible with the launching of the PCIe standard. In a PCI Express transport, the requests are tagged. This implies that the root complex can tell which device triggered each request. Consequently, the IOMMU, which is part of the root complex, can tell which device makes each DMA operation, and hence keep a different IOMMU table (called IOVA domain) per I/O device. This is called the **"isolation property"**. This last feature of modern IOMMUs is extremely useful in case of passed-through devices. Without this feature, passing through different devices to different VMs would not be possible, because each VM requires different GPA-to-HPA mappings[14].

2.14 Direct Device Assignment (Passthrough)

Direct Device Assignment [57] is the act of passing control of an entire physical device on the host system to a single Virtual Machine. So, instead of emulating the device on the hypervisor, we expose a physical device directly to the guest. Exposing a physical PCI device to a guest essentially involves exposing the PCI device resources, that is the PCI Configuration Space, the PCI memory address space, the PCI port address space and the device interrupts (legacy or MSI/MSI-X). Note that, under certain circumstances, this can be done in such a way that the guest can access these resources directly, thus taking the hypervisor completely out of the picture. Direct device assignment is usually useful for network devices, storage devices and GPUs. The advantage of this technique is the near-native performance in terms of latency and throughput. The performance is definitely better than any other virtualization technique (QEMU

emulation, paravirtualization, vhost, vhost-user). The disadvantage is that we have to assign the entire device to a single VM, thus losing the ability for device sharing. However, this is getting addressed in modern devices with SR-IOV[58]. SR-IOV is about exposing a single physical device as multiple virtual devices to multiple VMs. This seems the same as device emulation. The key difference with device emulation is that the virtual devices are implemented by the device itself (the hardware) instead of the hypervisor (the software).

There are certain requirements for the implementation of Direct Device Assignment. Firstly, QEMU must have access to the device resources (PCI configuration space, memory regions, ports). This is necessary because QEMU is responsible for creating the guest system architecture (CPU, chipset, PCIe topology) and also QEMU may need to modify some of the device information (eg. capability bits, etc.). Secondly, QEMU must be notified for device interrupts. Depending on the hardware limitations, there may be no hardware virtualization extensions for routing a device's interrupt directly to a guest. In this case, QEMU/KVM has to perform the interrupt injection. Thirdly, in case the device is a DMA-capable device, it must be able to perform DMA from the guest memory in a secure way.

All the above requirements are actually the same as with the case of user space drivers. So, Direct Device Assignment is implemented with the aid of vfio kernel driver. vfio provides access to a device in a secure and programmable IOMMU context. QEMU has access to the device resources through the corresponding vfio device file under */dev/vfio/*. Also, vfio uses eventfds to notify QEMU about device interrupts. In specific, whenever the kernel receives an interrupt caused by the passed-through device, it kicks the eventfd that corresponds to that interrupt. Last but not least, DMA operations from the device to the guest memory is enabled with the aid of the host IOMMU. The IOMMU is necessary for two reasons:

1. First, it performs the GPA-to-HPA translations, which are necessary for doing DMA operations directly to guest memory.

2. Second, it offers security in the DMA operations in the sense that the device can access only a portion of the guest memory.

QEMU is responsible for setting up the GPA-to-HPA mappings in the IOVA domain

of the passed-through device. This is quite tricky and there are various techniques for this[59][60]. Although declaring the whole guest memory as DMA-able to the vfio driver is sufficient, it is not a suitable solution, because vfio will pin all guest memory. Note that setting up GPA-to-HPA translations involves both translations for the guest physical memory and translations for the memory address spaces of the guest PCI devices (thus enabling peer-to-peer DMAs as well).

2.15 File Sharing via Unix Sockets

The Linux kernel offers a mechanism for sharing an open file between unrelated processes on a local system. Unrelated here means with no parent-child relationship. This mechanism uses the ancillary data of a message sent over a Unix domain socket. This procedure is often referred to as "file descriptor passing", but this is inaccurate because we do not pass a file descriptor. What happens is that the kernel creates a new file descriptor referring to the same *struct file* instance. This mechanism resembles to the way the *dup()* system call works.[61]

Ancillary data can be sent with system calls *sendmsg()* and *recvmsg()*.

3 Introduction to Storage Performance Development Kit

3.1 What is SPDK (Brief Description)

SPDK [62] is a user space driver framework for storage. It is a set of libraries, drivers and target applications for a variety of storage protocols like vhost, NVMe-oF and iSCSI. It supports a variety of storage backends like malloc RAM-disks, local NVMe disks, remote NVMe disks, virtio-scsi devices, virtio-blk devices, Linux AIO block devices, etc. It runs completely in user space and bypasses the kernel storage stack. It has been designed for maximum scalability and efficiency, especially in case of next generation NVMe SSDs.

The motive for the creation of SPDK was an earlier Intel's driver framework called **DPDK** [63]. DPDK is a user space driver framework for networking that achieves great performance by combining the Intel CPUs and NICs. So, the Intel engineers thought about creating an equivalent framework for storage that will make best use of the Intel NVMe SSDs. Therefore, SPDK has been created. At first, it was called "DPDK for Storage". SPDK and DPDK are not independent projects. SPDK incorporates DPDK as a submodule and it uses DPDK's core libraries to do some low-level operations like memory management (hugepages), PCI bus scanning, vfio device handling, message passing, implementing lockless queues (I/O channels), etc.

3.2 Purpose of SPDK, target audience, use cases

In the last few decades there has been a great evolution in the storage media technology. Solid-state storage media have come to replace the slow spinning rotational hard disks. Solid-state media offer significant advantages over the conventional hard disks like lower latency, greater throughput, higher parallelism, lower power consumption and bigger rack density. In figure 3.1, we present an empirical estimation about the average hardware latencies (hardware latency = bus latency + device latency) of the main storage technologies compared to the software latencies. In practice, a more accurate estimation would require taking into consideration many factors, like queue depth, I/O pattern, etc.

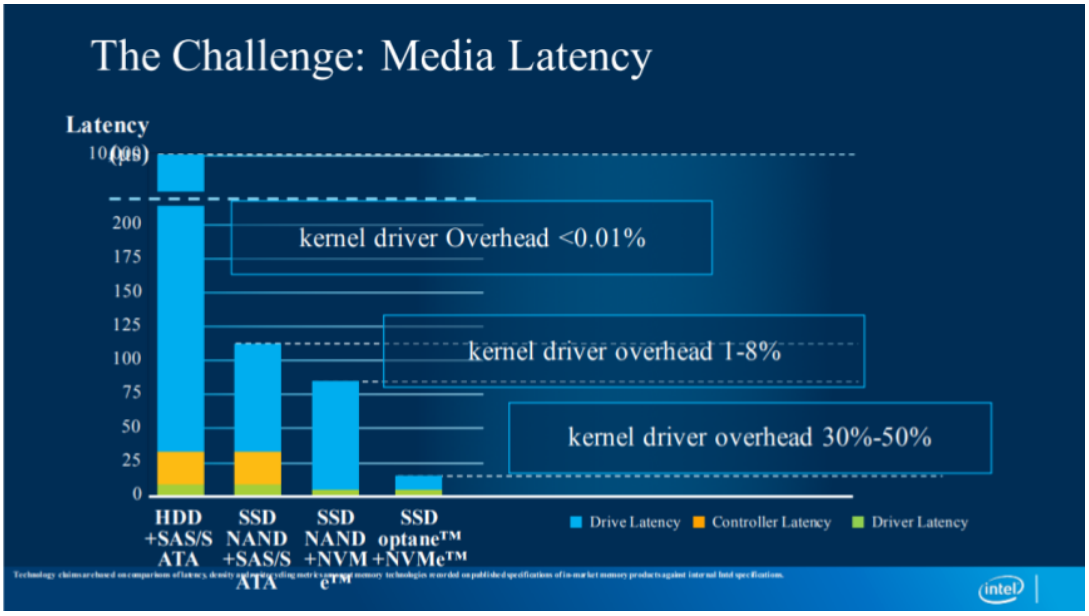


Figure 3.1: *HW I/O latency*

Moving into new storage media has unlocked a huge range of possibilities about what people can do with their storage. Drives are getting incredibly fast. This shift from the slow spinning disks to the incredibly fast solid state media comes to the challenge. In the era of hard disks, it didn't really matter if you had a small portion of software latency from the software storage stack, because the media itself was so slow. But, as soon as we entered the solid state era, the software latency became a bigger portion of the overall latency. The software latency for I/O operations comes mainly from the kernel storage stack. The kernel storage stack has been designed with a top-down approach in order to be able to sufficiently handle a great variety of storage media. This means that is has

been designed is such a way that it works decently with any storage media underneath. However, this comes to an expense of not being optimized for certain storage media, like the NVMe SSDs. To conclude, nowadays it makes sense to focus our efforts on reducing the software I/O latency in order to improve the overall I/O latency.

SPDK is a driver framework for storage. Unlike the kernel storage stack, it has been designed in a bottom-up approach. This means that it has been optimized to make the best out of the modern solid-state media. As a rule of thumb, it has been measured to offer 10 times greater I/O performance than the kernel storage stack. SPDK also works as a reference storage architecture, as it is the first user space storage framework. Except for SPDK applications, someone could possibly use some SPDK libraries in other projects. This is feasible due to SPDK's BSD license.

However, it is important to emphasize that SPDK is not targeted to average I/O workloads. For those tasks, the kernel is doing pretty well. SPDK refers to users that want to run **I/O intensive workloads**. In addition, SPDK is a great solution for datacenters, because it gives to the datacenter architects the tools they need to increase the overall scalability, flexibility and efficiency of their storage architecture. Getting into more detail, efficiency is about the number of cores we need to keep up with a specific storage workload. SPDK needs less cores to reach the same amount of IOPS than the Linux kernel. Scalability is about increasing the number of storage devices handled by the software storage stack. With SPDK this is possible by simply using more cores for the I/O processing. And also this can be seen from another perspective. SPDK offers greater scalability as it allows scaling out the number of locally connected physical disks per machine while achieving proportional IOPS scalability.

Last but not least, SPDK takes advantage of the high parallelism of modern hardware. Modern CPUs have many cores. Modern I/O devices have many I/O queues. SPDK makes use of this inherent parallelism by assigning I/O queues to cores. Each core works on a specific I/O queue. The idea behind this approach is to eliminate the need for locks, and thus avoid the inherent constraints of the locking schemes. Threads can communicate with message passing to resolve synchronization issues.

3.3 Architecture

SPDK has a modularized architecture. It is made of libraries, device drivers and applications. The following figure depicts the layout of the various SPDK components that make up the SPDK framework.

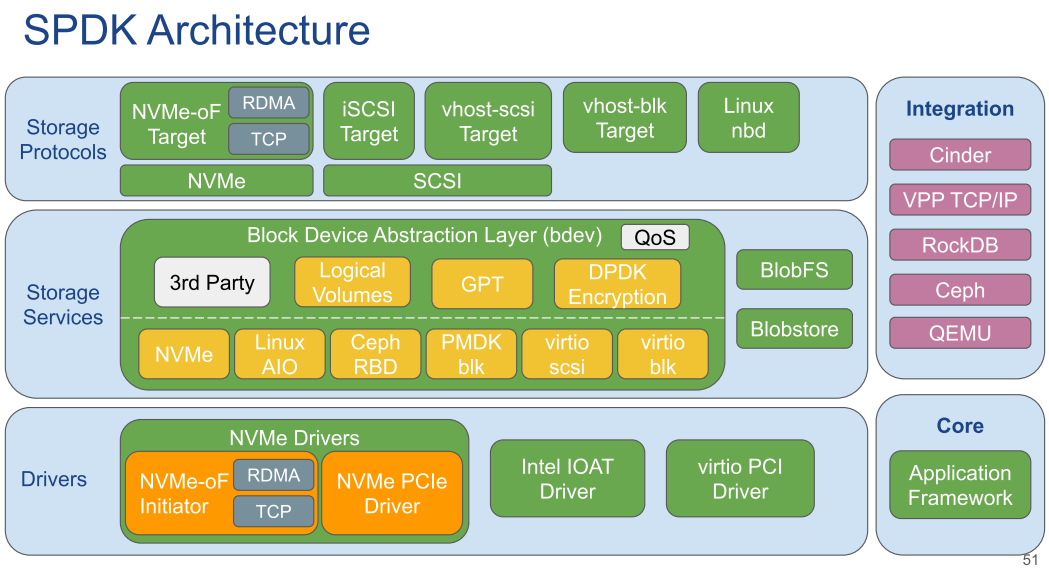


Figure 3.2: *SPDK Architecture*

The SPDK's components can be grouped into three distinct layers. Starting from the bottom layer and building up, these layers are the following [64]:

1. **Hardware Drivers**

   This layer contains the device drivers for the physical devices. At the moment, it contains an NVMe driver, an I/OAT driver and a virtio-pci driver. The NVMe driver contains an abstracted transport API so that it can support implementations for various NVMe transports. At the moment, the NVMe driver supports PCIe transport, RDMA transport and TCP transport.

2. **Block Device Layer**

   This layer is equivalent to the kernel block layer. It can be split into two parts:

   (a) the Block Device Abstraction Layer (BDAL). This layer creates a generic interface for all block devices implemented by the underlying bdev modules. This is the API exported to the storage protocols lying on top. Each

new bdev module must implement this API. This generic interface could be parallelized with the *request* function that has to be implemented by all block device drivers in the kernel.

(b) the bdev modules. A bdev module is the equivalent of a kernel block device driver. This means that it exports block devices (or bdevs for short) from storage backends, just like kernel block device drivers export block device files. For example, the NVMe bdev module creates bdevs from local NVMe SSDs or remote NVMe SSDs. The actual hardware is driven by the underlying NVMe driver. The bdev modules implement a generic interface defined by the BDAL, so that the upper layers can interact with the bdev modules through the BDAL. SPDK also supports creating virtual bdevs (vbdevs) on top of other bdevs. This allows implementing higher level operations like Logical Volumes (lvols), GPT, encryption, compression, etc.

3. **Storage Protocols**

   This is the upper layer. It contains a set of libraries that implement the following storage protocols: vhost-user target (blk,scsi,nvme), iSCSI target, NVMe-oF target. This layer interacts with the underlying bdev modules through the BDAL API.

All the above are just libraries. SPDK also contains a number of applications. These applications make use of the above SPDK libraries that implement the storage protocols. Therefore, the code of the applications is quite trivial. This separation between the storage protocol libraries and the applications that make use of these libraries has been made deliberately so that a third party could make use of these libraries in other projects. SPDK is equipped with four applications:

- iSCSI target
- NVMe-oF target
- vhost target
- spdk target (a unified application combining the above three)

SPDK relies on DPDK EAL library for some low-level management operations. Given that DPDK already has some core libraries that implement these operations, SPDK integrates DPDK as a submodule and makes use of DPDK libraries.

3.4 How it works (Key features, primary concepts)

SPDK achieves greater performance (better efficiency and scalability) than the kernel storage stack. This has been achieved by combining three key techniques [65]:

1. SPDK is a **user space** framework[66]. This means that it runs completely in user space and it bypasses the kernel storage stack. In other words, it handles the storage devices directly from user space, thus removing the kernel overhead from the critical path. In detail, what happens is that the kernel exposes all the PCI resources of each PCI storage device to user space. The PCI resources of each PCI card is the PCI Configuration Space, the memory and port address spaces and the interrupts. Currently, the kernel offers two mechanisms (kernel modules) for this purpose. These are the uio driver and its ancestor, the vfio driver. Bypassing the kernel is useful for two reasons:

   (a) we avoid context switches between kernel space and user space

   (b) we are not relying on the general-purpose kernel storage stack that adds unnecessary overhead

2. SPDK is **lockless**, at least in the datapath. SPDK uses message passing among threads for synchronization instead of locks. It is not obvious that this is a better synchronization technique in terms of scalability. However, it turns out that locks have some intrinsic limitations that make them non-scalable like lock contention, cache coherence in NUMA systems, etc. In specific, SPDK works as follows: each SPDK application is multithreaded. Each shared data structure is assigned to a single thread. This means that this thread will always be responsible for the manipulation of this data structure. This mainly concerns HW I/O queues (NVMe queue pairs, virtqueues, etc.). Whenever another thread needs to do some operation on this data structure, it sends an event to the other thread via a lockless ring buffer. Each event declares what job needs to be done. An

event is basically a function pointer and pointers to the function arguments. This approach scales better than locks, because the data remain in the cache close to the thread that manipulates the data. It is important to note at this point that SPDK does not use message passing to move data, and hence to speed up the data processing. SPDK uses message passing to eliminate the resource contention caused by conventional locking schemes.[67]

3. SPDK uses **poll-mode drivers**. SPDK polls on devices for I/O completions instead of waiting for interrupts. It turns out that in case of modern low-latency NVMe devices, interrupts add a significant overhead, thus turning polling into a better option. The explicit reasons for the high overhead of interrupts compared to polling are analyzed below. First of all, polling is more efficient than interrupts in case of intensive I/O workloads. Handling an interrupt is very expensive compared to the modern media overhead, because it requires the intervention of the operating system. In specific, it requires two context switches (swap out the running process, run the interrupt handler, swap in the previous process) and the execution of the interrupt handler, which is an additional software overhead. Secondly, the execution of the interrupt handler affects the CPU cache, thereby causing additional cache misses for potentially performance-critical data. Thirdly, polling on an NVMe device is fast, because only host memory needs to be read (no MMIO) in order to check a queue pair for a bit flip, and technologies such as Intel's DDIO will ensure that the host memory being checked is present in the CPU cache after an update by the device. It is important to mention at this point that polling does not mean that SPDK threads are blocked waiting for I/O completions. SPDK uses asynchronous I/O, which means that SPDK multiplexes other work with periodically checking for I/O completions. Also, note that, due to the low-latency properties of the modern storage media, the Linux kernel also supports a variation of polling which is called "hybrid polling". Its purpose is to avoid the overhead of waking up an idle core waiting for I/O completion (synchronous I/O).[68]

Except for the aforementioned techniques, SPDK also uses some more techniques to further optimize the performance. In case of NVMe devices, SPDK performs far fewer MMIO writes than most NVMe drivers. When the CPU performs an MMIO write, a

request is generated by the CPU that's placed into a hardware queue to later be sent over the PCIe bus to the device. If the CPU generates too many MMIO operations and overflows that queue, the CPU will stall and wait for a slot at the end of the queue to open up. What is more, SPDK, and especially the low-level drivers, have been optimized so that they make best use of the system's cache hierarchy. This involves mainly minimizing the collisions, avoiding stalling due to chained data-dependent loads and prefetching when possible.[65]

3.5 Application Framework

SPDK is made of many components. The Application Framework [69][70] is the glue that combines all these components. The associated code lives under *lib/event/*. The application framework builds on the environment abstraction framework.

SPDK adopts a multithreading model of execution. During the app initialization, SPDK creates one thread for each core mask and pins all these threads to the underlying cores. The cores that will be used by the SPDK application are provided by the user via a command-line core mask parameter. The thread launching and pinning is done with function *spdk_reactors_start()* in *lib/event/reactor.c*. These threads are essentially busy loops (or schedulers) and they are called **"reactors"**. So, SPDK maintains one reactor per core. Each reactor just awaits for and runs any incoming work. The code for each reactor is function *_spdk_reactor_run()*. The incoming work can either be an incoming **event** in the reactor's event ring or a **poller**. An event is just a function pointer along with a bunch of pointers to the function's parameters. The data structure for each event is *struct spdk_event*. Usually, events are being sent by other threads to ask for some work to be done on a specific data structure. This is how the message passing mechanism has been implemented in SPDK. As a reminder, SPDK achieves data synchronization by pinning certain data structures to certain threads so that these data structures can be processed only by their corresponding threads. A poller is also a function pointer and the function arguments. The difference between a poller and an event is that the reactor runs an event only once, while a poller is being run periodically. Pollers are usually simple tasks that check hardware for async events. For example, a poller could be checking the completion queue of an NVMe queue pair or

the used ring of a virtqueue. SPDK abstracts hardware I/O queues into a thread-bound (reactor-bound) data structure that is called **"I/O Channel"**. The data structure for an I/O channel is *struct spdk_io_channel*. I/O channels are being processed by pollers running on the same thread. I/O channels do not need locks, because they cannot be processed by different threads. Each I/O channel is being processed only by pollers and events running on the same thread. In general, SPDK's motivation is to parallelize I/O and eliminate resource contention by assigning each HW I/O queue to a separate thread.

Schematically, SPDK's application framework looks like this:

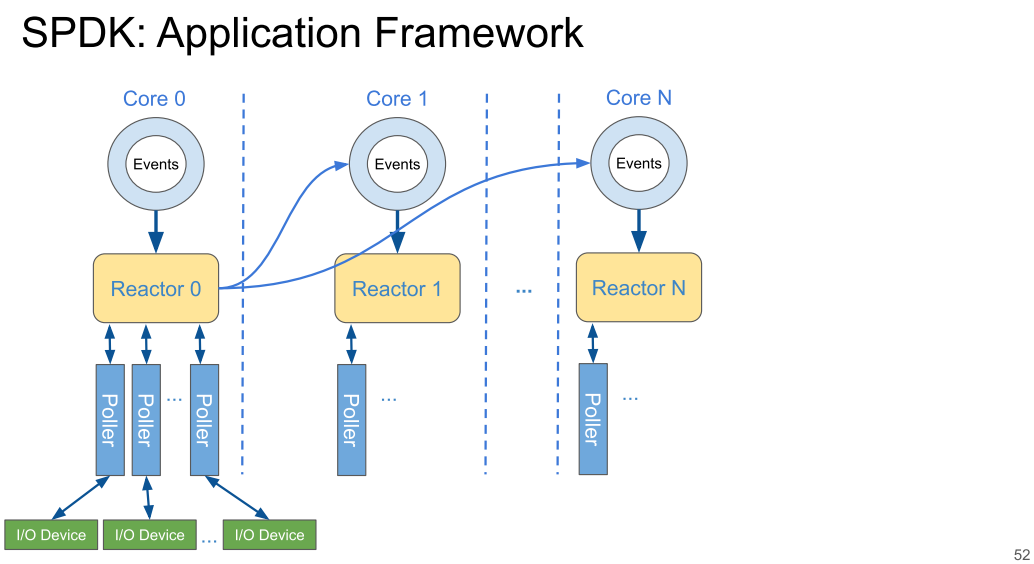


Figure 3.3: *SPDK Application Framework*

4 Vhost

4.1 What is vhost (Brief Description)

Vhost is a protocol specification that allows offloading the emulation of a virtio device from the hypervisor. This mechanism is device agnostic, meaning that it can be used to implement any paravirtualized device type that complies to the virtio specification. Currently, it is being used for the emulation of network and storage devices, where offloading the I/O data path from the hypervisor can make a huge difference in the I/O performance. The underlying vhost communication mechanisms are completely transparent to the guest device driver, so there is no need for guest kernel modifications.

In a nutshell, vhost is a very efficient mechanism because it works with shared memory. The emulated device has full access to the whole guest memory, hence there is no need for data copies in the data path.

Nonetheless, vhost does not emulate a complete virtio PCI adapter. Instead, it restricts itself to virtqueue operations only. It can also handle the device's virtio configuration structure. QEMU is still used to implement the PCI resources (PCI Configuration Space, memory address space, port address space) and to perform tasks like, for example, virtio feature negotiation and live migration. This means that a vhost driver is not a self-contained virtio device implementation. It depends on QEMU to implement the transport-specific guest-visible part and handle the device's control plane. In other words, vhost is a mechanism that enables offloading a virtio device's data plane.

There are two implementations of the vhost mechanism. The first approach is running the vhost device backend (that is the virtio backend driver and the device emulation code) inside the host kernel. This is the initial vhost implementation and is part of the Linux kernel. The second approach is running the vhost device backend as a separate process on host user space alongside QEMU. The former approach is called "vhost"[5] and the latter approach is called "vhost-user"[6]. Essentially, "vhost-user" is nothing more than just an extension of the "vhost" mechanism on user space. We will get into more details about these two approaches in the following sections.

4.2 Purpose of vhost

Vhost is a mechanism that can be used to offload the I/O datapath of an emulated virtio device from the hypervisor. This is useful for a number of reasons[71]:

1. the hypervisor's I/O data path may not scale well. For example, in QEMU, the I/O datapath goes through just one host thread (the IOThread) [25]. This implies that this is a bottleneck in scaling up the IOPS, even if we supply the guest kernel with more vCPUs. Making changes in QEMU architecture to fix that is quite complicated. So, vhost is a great solution in case the hypervisor is the bottleneck.

2. taking the hypervisor (QEMU) out of the picture may result to improved I/O performance. This happens both in case of the kernel vhost and user space vhost. Regarding the kernel vhost, vhost inserts virtio emulation code into the kernel. This allows device emulation code to directly call into kernel subsystems instead of performing system calls from user space. For example, in case of a virtio-scsi device, the kernel vhost-scsi driver can submit I/O requests directly to the host kernel Block Layer. This approach is much faster than doing this with *read()/write()* system calls from QEMU user space. Regarding the user space vhost, we can improve the datapath with a customized user space implementation. There are some open-source projects out there, like SPDK and DPDK, which implement custom high-performance datapaths. Vhost-user may actually be a better choice than vhost for implementing a customized high-performance I/O datapath for two reasons. Firstly, vhost is part of the kernel

and therefore it is hard to incorporate changes/enhancements. Secondly, there are cases where we can have better performance if we completely bypass the host kernel. This is what DPDK and SPDK do. They use user space drivers that "talk" directly to the physical devices without the intervention of the kernel.

4.3 Differences between kernel-space and user-space vhost

As we have mentioned above, there are two implementations of the vhost mechanism. There is a kernel implementation, called "vhost", and a user space implementation, called "vhost-user". Historically, vhost appeared first, and vhost-user was just a copy of vhost in user space with some necessary adjustments in the communication mechanisms. There is a protocol specification for the vhost-user mechanism that is part of the QEMU Documentation. The main differences between vhost and vhost-user are the following:

- vhost is a series of kernel drivers living in *drivers/vhost/* in the kernel source code. They can be loaded as modules as well. Each vhost driver exposes a character device file, which serves as the interface for configuring the device instance. The device instance is configured through a series of ioctl types. These implement the vhost protocol messages. On the contrary, vhost-user uses a unix domain socket as a communication mechanism instead of ioctls. The device initialization and configuration is done through a set of vhost-user protocol messages, sent over the unix domain socket. These messages complement the vhost ioctl interface in the Linux kernel. We will get into more details about the vhost protocol messages in the next section.

- the set of vhost-user protocol messages diverges from the set of kernel vhost protocol messages in order to support more devices. This is reasonable because the kernel, in general, is more cumbersome in incorporating changes. At the moment, kernel vhost drivers support only virtio-net and virtio-scsi device backends.

Unfortunately, in the web, the terms "vhost" and "vhost-user" are often used interchangeably. For example, in SPDK Documentation, it goes without saying that "vhost"

means “vhost-user”, because SPDK runs on user space.

In this thesis, we are using the user space vhost implementation, so we are not going to talk more about the kernel vhost.

4.4 How vhost-user works

Vhost-user allows offloading the device emulation from the hypervisor into separate processes on host user space. From now on, these processes will be referred to as “vhost-user device backends”. In other words, vhost-user is a protocol for devices accessible via inter-process communication. The protocol defines two sides of the communication, **master** and **slave**. Master is the application that uses the emulated device, that is the hypervisor (QEMU). Slave is the process that performs the device emulation, that is the vhost-user device backend. From a different perspective, master is the application that shares its virtqueues and slave is the consumer of the virtqueues. Note that the terms **initiator** and **target** are also used.

The overall vhost-user mechanism can be split into two parts: the **control plane** and the **data plane**. The control plane is the mechanism for establishing virtqueue sharing between the device driver and the vhost-user device backend. The data plane is the mechanism for accessing those virtqueues from the vhost-user device backend. The data plane includes both the I/O control path, that is the movement of the metadata, and the I/O data path, that is the actual data movement. The metadata are usually structures holding pointers to the actual data in guest memory. Obviously, a well designed vhost device backend would have a complicated, secure control plane and, at the same time, a fast and efficient data plane.

The control plane is the mechanism for establishing the virtqueue sharing, so that the vhost-user device backend can gain access to the guest driver’s virtqueues and serve any I/O requests made by the guest virtio device driver. The vhost-user protocol defines that this setup is done through a series of **vhost-user protocol messages**. In general, all initialization and management information is exchanged using vhost-user messages. The vhost-user messages are primarily sent from master to slave through a unix domain socket on the host. However, there are some message types that are slave-initiated. There are also some cases where replies are necessary for certain master-

initiated messages. Master and slave can be either a client or a server in the socket communication. However, slave is commonly used as a server. This means that the slave creates the vhost-user socket and waits for guests to connect. The vhost-user device initialization via vhost-user messages is triggered whenever there is a new connection to a vhost-user socket. The message exchange goes as follows:

- a new client connects to the unix socket. The server creates a new socket with the *accept()* system call and triggers the message exchange.
- The connection starts with the feature negotiation. Master sends message *VHOST_USER_GET_FEATURES* to get the virtio features that the slave supports.
- The slave sends a reply with the virtio features that it supports.
- The master compares the slave's features with the features that he (that is the guest driver) implements. The master chooses the common features and enables those features in the slave with message *VHOST_USER_SET_FEATURES*.
- If the slave supports the *VHOST_USER_F_PROTOCOL_FEATURES* virtio feature, this means that the slave supports some vhost-user-specific protocol extensions. Any protocol extensions are gated by protocol feature bits, thus allowing full backwards compatibility on both master and slave. So, in case the slave supports some protocol extensions, the master sends message *VHOST_USER_GET_PROTOCOL_FEATURES* to get to know those vhost-user-specific features.
- The master checks which vhost-user-specific features are supported by the guest driver. So, he finds the common features and enables them by informing the vhost-user device backend accordingly with message *VHOST_USER_SET_PROTOCOL_FEATURES*.
- When the feature negotiation is over, the master shares the whole VM's file-backed memory, so that the slave can access the guest driver's virtqueues allocated in guest memory. The guest memory may be fragmented into multiple physically discontiguous regions on the host. The vhost-user specification puts a limit on their number - currently 8. The master sends the necessary information about all its regions with a message of type *VHOST_USER_SET_MEM_TABLE*.

The master passes file descriptors to the open files that correspond to the VM's file-backed memory through the ancillary data of the message, so that the slave can map the guest's memory. The master also sends the following information per region:

  - host virtual address - the virtual address in QEMU's host virtual address space where this region has been mapped
  - guest physical address - the address in which the guest kernel has mapped this region in the guest processor's physical memory address space
  - offset - offset where the file should be mapped
  - size - the size of the region

- The master may need to process the vhost device's virtio configuration structure, assuming that the device has a device-specific virtio configuration structure. This is done with messages *VHOST_USER_GET_CONFIG* and *VHOST_USER_SET_CONFIG*. The existence of a device-specific virtio configuration structure, just like other standardized parts of the device's PCI Configuration Space, is exposed to the guest driver via a virtio PCI capability in the device's PCI Configuration Space.

- After granting access to guest memory, the master sends information about the guest driver's virtqueues. This is necessary so that the vhost device backend driver can find the virtqueues and the I/O buffers inside guest memory. For each virtqueue, the master sends:

  - its size with *VHOST_USER_SET_VRING_NUM*
  - the base offset in the avail ring with *VHOST_USER_SET_VRING_BASE*
  - the host virtual addresses of the avail, used and descriptor rings making up each virtqueue with *VHOST_USER_SET_VRING_ADDR*.

  For each virtqueue, the slave subtracts the HVA, where the guest memory is mapped in the virtual address space of QEMU process, from the HVA, where the virtqueue is mapped in the virtual address space of QEMU process, thus getting an offset. The slave then adds this offset to the host virtual address where he has

previously mapped the guest memory in his virtual address space. This is how the slave tracks the virtqueues in guest memory.

- Finally, the master sends a kickfd and a callfd for each virtqueue with messages *VHOST_USER_SET_VRING_KICK* and *VHOST_USER_SET_VRING_-CALL* respectively. Kickfds and callfds are event file descriptors used for I/O submission notifications and I/O completion interrupts per virtqueue respectively. These file descriptors are created by the master and sent to slave with the same mechanism that is used to pass file descriptors for the guest memory regions. These eventfds are usually hooked up as KVM ioeventfds and irqfds, thus triggering lightweight VMEXITs and minimizing the virtualization overhead.

The data plane consists of the mechanisms for accessing the virtqueues and sending/receiving notifications. It is important to note that the data plane can be further split into the I/O control path and the I/O data path. The I/O control path is the processing of the metadata that describe each I/O request. Essentially, the metadata are pointers to the actual data in guest memory. The I/O data path is the processing of the actual data. The metadata are handled differently than the data. When the vhost device initialization phase is finished, the slave has access to the whole guest memory, and hence to the driver's virtqueues inside guest memory. The vhost device is ready to handle I/O requests generated by the guest driver. The guest driver sends I/O requests by allocating I/O buffers in guest memory, filling them with data and putting the guest physical addresses of these buffers into virtqueues. The guest driver then kicks the virtqueue to notify the device backend about the new I/O requests. According to the virtio specification, kicking the virtqueue means writing to an I/O port correlated to this virtqueue. The control returns to QEMU, which kicks the corresponding kickfd related to this virtqueue, thus notifying the vhost device backend. However, note that the kickfd could have been kicked by KVM directly if it had been hooked up as an ioeventfd, but this is just an optimization. The vhost device backend polls on the kickfds, notices the kick of the eventfd, and thus learns about the existence of new descriptors in the virtqueue. So, it reads the descriptors. The desciptors contain GPAs pointing to the actual data. The vhost device backend translates the GPAs to HVAs by subtracting the GPA where the guest physical memory is mapped in the guest physical memory address space and adding the remaining offset to the virtual address where it

has mapped the guest memory in its virtual address space. The vhost device backend may also split the requests into I/O vectors in case they are fragmented into separate vhost memory regions. Then, the vhost device backend serves the I/O requests. Once I/O completes, the vhost device backend fills the response buffer with proper data and notifies the master by kicking the corresponding callfd. Similarly to the kickfd, either QEMU or KVM notices the kick and notifies the guest driver by injecting a virtual interrupt. We deliberately did not get into details about the form of the I/O requests and the I/O control and data path beyond the vhost device backend, because the vhost-user protocol is **device-agnostic**. The number of virtqueues, the layout of the descriptors in the virtqueues and the I/O data path depends on the type of virtio device. The above abstract analysis of the vhost-user data plane applies to any type of virtio device.

Various optimizations can be applied to the data plane in order to make it more efficient. For example, using poll-mode drivers both in the master guest kernel and in the slave process would completely eliminate the hypervisor's intervention in the I/O control path, thus eliminating VMEXITs.

Last but not least, the vhost-user protocol supports some optional features. For example, the protocol specification includes features like pre-copy/post-copy live migration, multi-queue support and IOMMU support.[72]

4.5 More about inter-process communication via shared memory

It has been mentioned in the previous section that the vhost-user protocol grants the vhost device backend process with full access to the guest physical memory. The backend tracks the driver's virtqueues inside guest memory via some vhost-user messages. A reasonable question rises at this point: "Why should we expose the whole guest memory to the backend process and not just constraint the backend process with access to the driver's virtqueues and I/O buffers?". Let's get into more details about this approach.

The vhost backend process must have access to the driver's virtqueues and the I/O buffers containing the actual data. In case of block devices, the data may be stored

in kernel memory (page cache) or in user space memory (Direct I/O). This means that the I/O buffers are not stored in fixed positions. Consequently, giving access to the whole guest memory solves this problem without any further modifications in the guest device driver.

The other approach, that is restricting access to just a portion of guest memory, would work as follows:

the guest device driver should allocate the virtqueues inside this memory. The I/O buffers should be allocated inside this memory as well. In case of virtio-scsi driver, whenever a new I/O request would arrive from the SCSI mid-layer, the driver should check if the I/O buffer (the actual data) related to this request is located inside the portion of shared memory. If not, the driver should copy the data inside the shared memory so that the vhost backend process can access them.

Comparing these two approaches, the advantage of exposing the whole guest memory, instead of just a portion of it, is that there is no need for data copies, thus achieving a fast zero-copy datapath. On the other hand, what we lose by exposing the whole guest memory is that the backend process has unrestricted access to all guest memory, and this is certainly a security hole.

5 Design of SPDK/VVU

In this chapter, we are going to analyze how the SPDK/VVU mechanism works, what are the necessary changes that we have to make in order to implement this, and how we ended up to this design after the interaction with the open source communities.

5.1 General Description

SPDK is a storage framework with a lot of libraries and applications that utilize these libraries. Among other things, SPDK can create vhost targets that provide storage services to local VMs. The SPDK vhost library is capable of exposing virtualized block devices to QEMU instances or other arbitrary processes. It uses the vhost-user protocol to communicate with the VM.

In this chapter, we are going to introduce an alternative storage virtualization solution that relies on SPDK vhost. This new storage virtualization solution is called **"SPDK/VVU"**. It is based on the SPDK vhost application and the virtio-vhost-user device, hence its name. It could be considered as an extension of the traditional vhost-user communication mechanism, which is defined in the vhost-user protocol specification. In a nutshell, SPDK/VVU works as follows: instead of running the SPDK vhost target on host user space, we are shipping the SPDK vhost target inside a dedicated Storage Appliance VM, running locally alongside the Compute VM. This way we can have guests offering storage services to other guests.

Isolating the SPDK vhost target inside a Storage Appliance VM requires that we extend

the vhost-user protocol communication mechanisms, so that the vhost target can still have access to the Compute VM's memory. The vhost-user protocol is split into three parts: the control plane, the data plane and the notification mechanism. Actually, the notification mechanism could be considered as part of the data plane, but we are going to see it separately here. In this chapter, we are going to look through a proposal for extending the traditional vhost-user communication mechanism, as described in the specification, from the host all the way up to guest user space in the Storage Appliance VM. We are going to introduce a new virtio device for this purpose that is called "virtio-vhost-user". Therefore, the new vhost-user transport is called "virtio-vhost-user transport". The traditional vhost-user transport is called "AF_UNIX transport", since it is based solely on socket communication. We are going to explain how the virtio-vhost-user device extends the vhost-user communication mechanisms. Last but not least, we will have to extend the SPDK and DPDK codebase (DPDK has been integrated as a submodule in SPDK), so that the SPDK vhost target can work over this new transport. We also need to extend the API, so that this new functionality is exported to the end user.

Our goal in this thesis is not just to implement this new storage virtualization mechanism, but to push it upstream into the corresponding projects. Our goal is to become members of the SPDK, DPDK, QEMU and VIRTIO open source projects, interact with the communities and push changes into these projects, so that SPDK/VVU is available to everyone. In this direction, we first approached Stefan Hajnoczi [1](Software Engineer at RedHat's virtualization team). Stefan owns the original idea for the virtio-vhost-user device and the setup of VMs offering storage services to other VMs. Stefan had actually presented an RFC implementation for the device and for its spec. However, the project had been stalled due to lack of spare time. We used this RFC implementation and started working on the SPDK codebase in order to add support for the virtio-vhost-user transport (we will show more information on this in section List of Changes). After some time and after a successful demo of the SPDK/VVU setup in the SPDK Community Meeting, Stefan authorized us [2] to pick up his work and finish the device. Currently, we are awaiting for the device spec to be approved by the virtio committee. For more information about our current state and our next steps, refer to

[1] https://lists.01.org/pipermail/spdk/2018-September/002488.html
[2] https://lists.01.org/pipermail/spdk/2018-December/002822.html

the Conclusion chapter.

5.2 The virtio-vhost-user device in a nutshell

The virtio-vhost-user device is a QEMU character device that will be used to implement SPDK/VVU. It is a paravirtualized device that complies with the Virtio specification. Its purpose is to extend the vhost-user transport, so that it will be able to offload the vhost device backend process from host user space into a dedicated Storage Appliance VM running locally alongside the Compute VMs. In other words, the virtio-vhost-user device allows running vhost devices inside VMs. We will see how this is achieved in the following sections.

The virtio-vhost-user device is equipped with a pair of RX/TX virtqueues for host-guest communication. It complies with the modern virtio interface (in other words, not compliant with legacy drivers). It supports MSI-X interrupts. Currently, the virtio-vhost-user device has been implemented over PCI transport only.

Unlike the rest of the virtio PCI devices, it is equipped with some additional device resources. These resources are standardized with three virtio PCI capabilities. In specific, the additional device resources are “doorbells”, “notifications” and “shared memory”. The doorbells resource is a set of MMIO addresses that are hooked up with eventfds. This means that whenever the guest driver kicks a doorbell, the QEMU device code emulates this operation by sending an event to the appropriate eventfd. The guest driver can find the location and the number of the available doorbells in the device’s memory space via a doorbell configuration structure pointed out by the doorbell capability. The notifications resource is the opposite to doorbells. MSI-X vectors are being hooked up to eventfds. Whenever an eventfd is kicked on the host, the QEMU device code triggers a guest interrupt with the appropriate MSI-X vector. The device offers a notification capability, so that the guest driver can hook up MSI-X vectors to certain eventfds. Finally, the shared memory resource is a portion of the device’s memory address space that exposes a part of host memory to the guest. This memory is usually the file-backed memory of another VM. The guest driver can find out the address and length of the shared memory resource from the shared memory virtio PCI capability.

5.3 Extending the vhost-user control plane

In this section we are going to examine the vhost-user control plane and how it is altered in case of SPDK/VVU.

In short, the vhost-user control plane consists of the exchange of vhost-user protocol messages for setting up the vhost-user target. According to the vhost-user specification, the inter-process communication between the vhost-user master and slave for the exchange of the vhost-user messages is done through a unix domain socket.

In SPDK/VVU, the QEMU vhost-user driver has to be able to exchange vhost-user protocol messages with the SPDK vhost-user target that is running inside the Storage Appliance VM. This is fulfilled via the virtio-vhost-user device. The device is associated with a chardev socket that will correspond to the host vhost-user unix domain socket. It will also have a pair of virtqueues (RX and TX). The virtio-vhost-user device will be working as a proxy for the most of the vhost-user protocol messages. This means that it will be capturing any vhost-user messages arriving through the unix socket and will be forwarding those messages to the guest. Subsequently, the driver for this device will be receiving the messages from the RX virtqueue and passing them to the message handler of the SPDK vhost target. It is obvious that a similar procedure is followed for the slave-initiated vhost-user messages. It is important to mention that the virtio-vhost-user device works as a proxy for the most of the vhost-user messages, but not for all of them. There are some message types that require some work to be done by the virtio-vhost-user backend driver on the host side.

5.4 Extending the vhost-user data plane

In this section we are going to examine the vhost-user data plane and how it is altered in case of SPDK/VVU.

In short, the vhost-user data plane consists of the chain of events that are being done during the submission of a block I/O request from the guest. It involves transferring the metadata (eg. SCSI CDB) to the vhost target and further down to the actual storage backend. It also involves the DMA operation that is performed by the storage backend in order to serve the I/O request described in the metadata. In other words, the vhost-

user data plane consists of the I/O control path, the I/O data path and the notifications.

The metadata for each block I/O request are inserted in virtio descriptors from the master guest device driver. The actual data related to this metadata are saved in I/O buffers. The I/O buffers could be located either in the page cache or in user space memory in case of Direct I/O. Both the virtqueues and the I/O buffers are allocated by the Compute VM's virtio frontend driver in the Compute VM's physical memory. The SPDK vhost target and the underlying SPDK storage backend have to be able to have access to those resources inside the Compute VM's memory. In case of the vhost-user protocol, this works by exposing the whole VM's memory to the vhost target. In case of SPDK/VVU, we need a mechanism so that the vhost target, running in slave guest user space, can still have access to the Compute VM's memory. This is done as follows:

during the initialization phase, the master sends a *VHOST_USER_SET_MEM_TABLE* vhost-user message, which contains file descriptors for the file-backed memory of the master VM. The virtio-vhost-user device intercepts this message and it maps all the vhost memory regions in the virtual address space of the slave QEMU process. Then, the virtio-vhost-user device exposes the mapped vhost memory regions to the guest as a PCI MMIO region. This means that the master VM's memory is exported to the slave guest as device memory. Thus, the SPDK vhost target has access to the vhost memory regions through the PCI memory address space of the virtio-vhost-user device.

The SPDK vhost target polls on the vhost-user virtqueues for any new descriptors sent by the master guest driver. This means that the SPDK vhost target constantly accesses the PCI MMIO region that exposes those virtqueues to the slave guest. It is important to mention at this point that these MMIO accesses do not trigger VMEXITs. The aforementioned PCI MMIO region of the virtio-vhost-user device is backed by host memory, and specifically by the file-backed memory of the master VM. This means that QEMU registers the mapped memory of the master VM as a memory slot to the KVM module. As a result, any accesses from the slave guest to the MMIO region result in a valid page table entry in the Extended Page Table, and thus no VMEXITs occur. To conclude, the I/O control path and data path do not involve the intervention of the hypervisor at any point, except for the slave-to-master notifications, which we are going to examine in the following section.

5.5 Extending the vhost-user notification mechanism

In this section we are going to examine the vhost-user notification mechanism and how it is altered in case of SPDK/VVU.

The vhost-user notification mechanism is the means for the master guest virtio frontend driver of the virtio block device to notify the SPDK vhost target for new descriptors, and conversely, the means for the vhost-user target to notify the guest driver for the completion of the submitted I/O requests. The vhost-user protocol defines that event file descriptors are used for these notifications. The event file descriptors are called kickfds and callfds respectively. They are usually accompanied with the ioeventfd/irqfd KVM features to trigger lightweight VMEXITs.

According to the vhost-user protocol, the kickfds and callfds are created by the master and passed to the slave via vhost-user messages. However, in case of SPDK/VVU, it is obvious that these eventfds cannot be passed to the SPDK vhost target running in slave guest user space. So, the solution adopted in SPDK/VVU is the following:

the virtio-vhost-user device intercepts these vhost-user messages and saves the kickfds and callfds. Concerning the kickfds, the device supports hooking up kickfds to MSI-X vectors. It exposes a virtio PCI capability to the guest so that the guest driver can set up the mapping between callfds and interrupt vectors. However, notice that in case of SPDK/VVU, this feature is unused, because SPDK relies exclusively on polling and therefore disables the notifications from the frontend driver. Concerning the callfds, the device exposes one doorbell per callfd to the guest. The doorbells are MMIO addresses in the BAR 2 MMIO region. The device also registers the doorbells as ioeventfds. This means that every access to a doorbell causes a trap→VMEXIT and the KVM kicks the corresponding callfd.

Note that the doorbells and the shared memory regions are mapped to the same PCI MMIO region of the virtio-vhost-user device. In QEMU, it is possible to create an MMIO region that is split into multiple subregions. These subregions may be emulated in different ways. For example, in QEMU it is possible to create a PCI BAR that is composed of a RAM region and an MMIO region. This means that accesses to different addresses inside that PCI BAR are emulated in a different way. If a guest MMIO address is backed by host RAM, then an access to this address results in a valid PTE in the

extended page table, and therefore, there is no VMEXIT. If a guest MMIO address is emulated by QEMU, then an access to this address causes a trap→VMEXIT, the KVM returns the guest physical address to QEMU to emulate, and QEMU emulates the access using the associated callbacks.

5.6 Changes in SPDK and DPDK

According to the previous chapters, the virtio-vhost-user device extends the vhost-user communication mechanism, thus introducing a new vhost-user transport. We call it the "virtio-vhost-user" transport as opposed to the "AF_UNIX" transport, which is the traditional vhost-user transport defined by the vhost-user protocol specification. In order to add support for this new transport, we have to make changes in SPDK and DPDK. The reason why we are engaging with DPDK is because SPDK incorporates DPDK as a submodule and uses DPDK's *librte_vhost*, which is the control plane implementation for the vhost-user protocol. In the following section, we are outlining the architeture of the vhost-related code in SPDK, so that it becomes clear what we have to change.

5.6.1 Architecture of SPDK's vhost code

SPDK contains a library called "vhost" that implements a vhost-user target. The architecture of the target can be split into three parts:

- the control plane implementation
- the generic data plane implementation
- the protocol-specific data plane implementations

The control plane implementation is the code related to vhost-user message handling, socket I/O, fd handling and setting up of the translations for the virtqueues and the I/O buffers. SPDK maintains its own control plane implementation in *lib/vhost/rte_vhost/*. This library is just a copy of DPDK's *librte_vhost* with some storage-specific changes. Recall that SPDK incorporates DPDK as a git submodule. Recently, a new feature has

been added in SPDK that allows using DPDK's *librte_vhost* directly. The long-term plan is that the internal *rte_vhost* copy will be only maintained as a legacy feature.

The generic data plane implementation is the code for creating a new vhost device, destoying a vhost device, triggering the construction of the protocol-specific data plane and registering the vhost memory regions to the SPDK's user space page table (called "vtophys memory map"). The code lives in *lib/vhost/vhost.c*. It exposes its API with a *vhost_device_ops* structure, which is passed to the data plane. This is necessary for the data plane to be able to trigger the data plane creation when the vhost-user initialization phase is completed and, respectively, destroy the data plane, in case it is necessary. The generic data plane code interacts with the data plane through the public API of the latter (*rte_vhost.h*). This interaction is necessary for operations like triggering the vhost-user message exchange, when a new vhost device has been requested, or translating a guest virtual address to host virtual address.

The protocol-specific data plane implementation is the code that implements the data plane for each of the supported storage protocols. The code for the three supported protocols (virtio-scsi, virtio-blk, nvme) lives in *lib/vhost/vhost_scsi[blk,nvme].c*. Each protocol implements the *spdk_vhost_dev_backend* structure, which is used by the generic data plane to invoke the protocol-specific operations. The main field of this structure is function *start_session()*, which creates the data plane for a vhost device. It is invoked whenever a VM connects to the associated vhost-user unix socket and right after the completion of the vhost-user message exchange. In short, what this function does is to allocate I/O request buffers (called "tasks" in SPDK terminology and they are equivalent to the bio requests in the Linux kernel) for each of the device's I/O queues, and create a poller for each of the device's I/O queues. Each poller polls on the corresponding I/O queue in master guest memory. Whenever a poller tracks a new submitted I/O request, it parses the request, it translates it according to the storage protocol, and eventually delivers it to the SPDK Block Layer.

SPDK exposes all the aforementioned storage protocol implementations through the vhost application located in *app/vhost/*.

5.6.2 List of Changes

Based on the above explanation, it is clear that we have to make changes in DPDK's *librte_vhost*, in order to insert the virtio-vhost-user transport. In other words, we have to change the control plane. In specific, we have to do the following changes:

1. add a user space driver for the virtio-vhost-user devices
2. implement the virtio-vhost-user transport as part of *librte_vhost*.
3. export this new transport to the end user through the SPDK's application-wide APIs.

When we first engaged with SPDK, there was no option for using DPDK's *librte_vhost* directly. So, our first submitted patchset was attempting to add support for the virtio-vhost-user transport in SPDK's internal *rte_vhost* copy. Darek Stojaczyk (Software Engineer at Intel and core maintainer of SPDK) replied [3] by saying that the community's goal is to switch to DPDK's *librte_vhost* and maintain the internal *rte_vhost* copy as a legacy feature. Therefore, he rejected our patchset and kindly asked us to implement the virtio-vhost-user transport in DPDK's *librte_vhost* instead. After some discussion on the mailing list, we reached to an end-to-end plan [4]. This plan involved many steps, but the most important ones where the following:

1. support using DPDK's *librte_vhost* directly
2. implement the virtio-vhost-user transport in DPDK's *librte_vhost*

The first one has been implemented by Darek and released with the 19.04 release version of SPDK as an experimental feature[5]. The second one is our responsibility. For more on our current state on this and our next steps, refer to the Conclusion chapter.

5.7 Operation End-to-End

Ultimately, the end-to-end setup looks like this:

[3] https://lists.01.org/pipermail/spdk/2019-January/003045.html
[4] https://lists.01.org/pipermail/spdk/2019-March/003163.html
[5] https://spdk.io/release/2019/04/30/19.04_release/

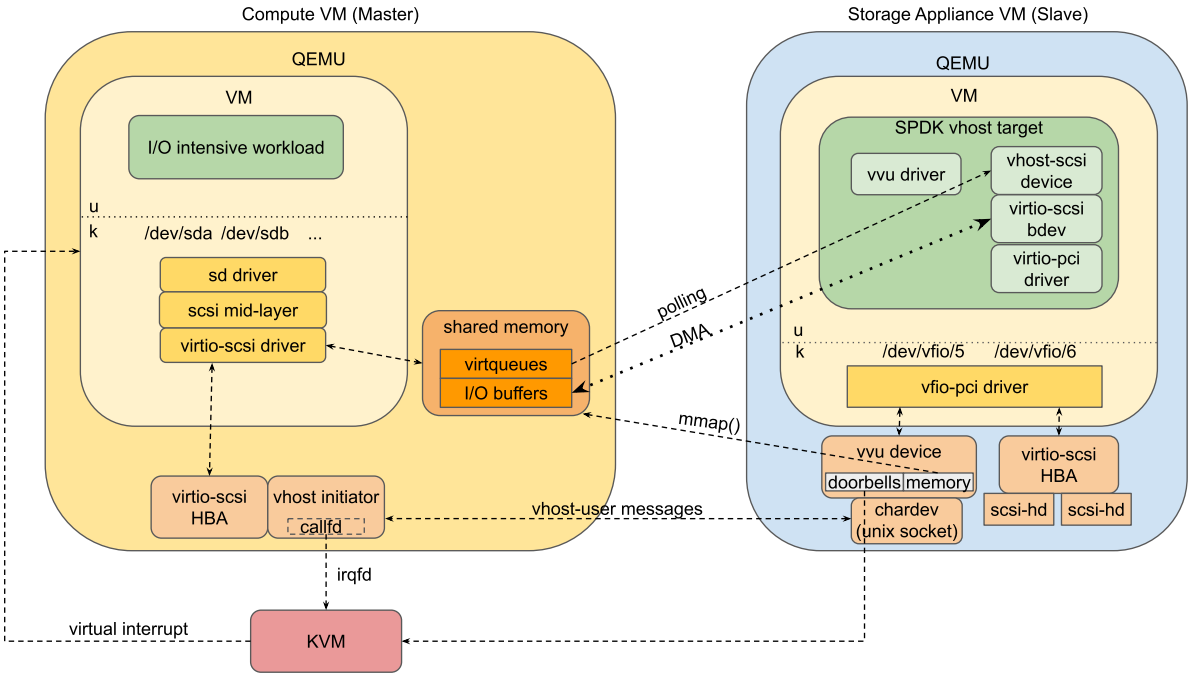


Figure 5.1: *SPDK/VVU Topology*

The setup consists of a Compute VM (master VM) and a Storage Appliance VM (Slave VM). The two VMs communicate via the virtio-vhost-user device. The master VM is equipped with a vhost-user-scsi PCI device, which is a virtio-scsi HBA whose backend part (the data plane) is being emulated outside QEMU. In specific, the guest-visible PCI-specific part (PCI Configuration Space, memory address space, port address space, interrupt mechanism) is being emulated by the master QEMU process, while the backend part is being emulated by a separate process. The backend part, also called the "vhost device backend" or "vhost target", is being emulated by the SPDK vhost app running in slave guest user space. Last but not least, the slave VM is equipped with a virtio-scsi HBA with a couple of SCSI disks attached on its SCSI transport. These disks serve as the storage backends for the emulated vhost SCSI disks exposed to the master VM via the vhost-user protocol. Note that the fact that we are using here a virtio-scsi HBA as a storage backend is just an example. There are many other options for this setup, like for example passing through an entire physical disk from the host. Ultimately, what we have is a VM (Storage Appliance VM) that offers storage services to another VM (Compute VM) via shared memory.

Getting into the internals of this storage virtualization mechanism, it would be easier

to be studied if we splitted the device operation into two parts: the control plane and the data plane. The control plane involves all the management operations that are necessary to setup the inter-process communication, configure the backend part and ultimately enable the beginning of the I/O processing. The data plane involves the whole data and metadata movement from the master guest memory to the underlying storage backend in the slave VM side.

5.7.1 Control Plane

Concerning the control plane, it is obvious that most management operations are being done during the intialization phase. The initialization phase is triggered by the connection of the master or the slave to the vhost-user unix domain socket. This depends on whether the master is the client or the server in the socket communication. In practice, this can be configured with the *server* option of the master's and slave's QEMU chardevs. However, notice that the virtio-vhost-user transport, in contrast to the AF_UNIX transport, does not support working as a client in the socket communication, so this configuration should be avoided in this case. The reason for this is that when the slave VM would boot up and the virtio-vhost-user device would connect to the master's unix socket, the vhost-user backend in the master side would immediately start sending vhost-user messages to the unix socket, while the virtio-vhost-user device wouldn't have been initialized yet (not bound to a driver yet). In the following explanation we assume that the slave works as a server in the socket communication.

In detail, the initialization phase goes as follows:

Suppose that the slave VM is up and running and the SPDK vhost target has been setup inside the slave VM and awaits for new connections. At this point, we boot up the master VM equipped with a vhost-user-scsi PCI device. The memory of the master VM is file-backed so that it is sharable and the slave VM can gain access to it. The master VM's memory could be backed either by regural 4KB memory frames (eg. a file on host tmpfs) or by hugepages for better performance (eg. a file on host hugetlbfs). In case we want to use a physical passthrough device as storage backend, then the same setup for the master VM's memory is sufficient. In such case, QEMU will declare the whole guest memory (guest physical memory and MMIO regions of all devices) as DMA-able memory to the host vfio-pci driver, which in turn will pin

the guest memory and program the host IOMMU with GPA→HPA translations.

When the master VM is booting up, the master QEMU's chardev socket connects to the vhost-user unix socket created by the virtio-vhost-user device in the slave side. Master QEMU also instantiates a vhost-user-scsi-pci device. This triggers the instantiation of a vhost-user-scsi device:

hw/virtio/vhost-user-scsi-pci.c:

```
static void vhost_user_scsi_pci_instance_init(Object *obj)
{
    VHostUserSCSIPCI *dev = VHOST_USER_SCSI_PCI(obj);

    virtio_instance_init_common(obj, &dev->vdev, sizeof(dev->vdev),
                                TYPE_VHOST_USER_SCSI);
    object_property_add_alias(obj, "bootindex", OBJECT(&dev->vdev),
                              "bootindex", &error_abort);
}
```

Then QEMU realizes the vhost-user-scsi instance by calling the *.realize()* method of this type. The realize function triggers the vhost-user message exchange by calling *vhost_dev_init()*:

hw/scsi/vhost-user-scsi.c:

```
static void vhost_user_scsi_realize(DeviceState *dev, Error **errp)
{
    VirtIOSCSICommon *vs = VIRTIO_SCSI_COMMON(dev);
    VHostUserSCSI *s = VHOST_USER_SCSI(dev);
    VHostSCSICommon *vsc = VHOST_SCSI_COMMON(s);
    Error *err = NULL;
    int ret;

    if (!vs->conf.chardev.chr) {
        error_setg(errp, "vhost-user-scsi: missing chardev");
        return;
    }

```

```
14      virtio_scsi_common_realize(dev, vhost_dummy_handle_output,
15                                 vhost_dummy_handle_output,
16                                 vhost_dummy_handle_output, &err);
17      if (err != NULL) {
18          error_propagate(errp, err);
19          return;
20      }
21
22      if (!vhost_user_init(&s->vhost_user, &vs->conf.chardev, errp)) {
23          return;
24      }
25
26      vsc->dev.nvqs = 2 + vs->conf.num_queues;
27      vsc->dev.vqs = g_new(struct vhost_virtqueue, vsc->dev.nvqs);
28      vsc->dev.vq_index = 0;
29      vsc->dev.backend_features = 0;
30
31      ret = vhost_dev_init(&vsc->dev, &s->vhost_user,
32                           VHOST_BACKEND_TYPE_USER, 0);
33      if (ret < 0) {
34          error_setg(errp, "vhost-user-scsi: vhost initialization failed:
35  %s",
36                     strerror(-ret));
37          vhost_user_cleanup(&s->vhost_user);
38          return;
39      }
40
41      /* Channel and lun both are 0 for bootable vhost-user-scsi disk */
42      vsc->channel = 0;
43      vsc->lun = 0;
44      vsc->target = vs->conf.boot_tpgt;
45  }
```

When the vhost-user connection is established, the vhost-user feature negotiation takes place. The master QEMU vhost-user driver starts sending vhost-user protocol messages to the vhost-user slave. These messages are intercepted by the virtio-vhost-

user device and tunnelled through virtqueues all the way up to the SPDK vhost target.

At first, the master negotiates the virtio features and the vhost-user protocol-specific features. After the negotiation, the master shares the Compute VM's file-backed memory, so that the vhost target can access it directly. The memory can be fragmented into multiple physically-discontiguous regions and the vhost-user specification puts a limit on their number - currently 8. The driver sends a single message for all the regions - the *VHOST_USER_SET_MEM_TABLE* message - giving the following data for each region:

- file descriptor - this is a file descriptor related to the Compute VM's file-backed memory
- host virtual address - the virtual address where the master guest physical memory is mapped in QEMU's virtual address space - used for memory translations in vhost-user messages (e.g. translating vring addresses)
- guest physical address - the master guest physical address where this memory region is mapped in master guest physical memory address space - used for address translations in vrings (for QEMU this is a physical address inside the guest)
- offset - positive offset for the mmap
- size - size of the vhost memory region

The virtio-vhost-user device captures this last message and maps all the vhost memory regions into the virtual address space of the slave QEMU process. It then exposes the mapped vhost memory regions to the slave guest as a RAM-backed PCI MMIO region (BAR 2). Later on, the master sends messages with the necessary information so that the vhost target can track the virtqueues and the I/O buffers inside the previously mapped vhost memory regions. Specifically, for each virtqueue, it sends the size of the virtqueue, the host virtual addresses for the three virtqueue structures (avail ring, used ring, descriptor table), and the starting index in those structures. Last but not least, the master sends the callfd and kickfd for each virtqueue with messages *VHOST_USET_SET_VRING_CALL* and *VHOST_USER_SET_VRING_KICK* respectively. These vhost-user messages are also intercepted by the virtio-vhost-user device.

The virtio-vhost-user device saves the kickfds (kickfds are not used in SPDK/VVU) and callfds, and maps the callfds to doorbells. The doorbells are exposed to the slave guest as the first 4KB in the BAR 2 PCI MMIO region of the virtio-vhost-user device.

5.7.2 Data Plane

When the device initialization is finished, the SPDK vhost target has access to the virtqueues in master guest memory and is ready for I/O processing. We will showcase an example with a read I/O request. Through this example, we are going to demonstrate the vhost-user data plane, that is the I/O control plane and the I/O data plane. Notice though that the kernel offers a wide range of interfaces, except for *read()*/*write()* system calls, and tweaks for making I/O faster.

Suppose that a process, running in master guest user space, issues a *read()* system call on a file saved on a vhost-user-scsi disk. Suppose also that this process uses direct I/O[73] (*open()* with *O_DIRECT* flag) in order to bypass the master guest kernel page cache. This is just an assumption in order to overlook the complexity added by the kernel page cache management subsystem. Using *O_DIRECT* poses certain limitations as far as the user space data buffer is concerned[74]. But we will overlook these limitations as well, as this is not the purpose of this thesis. Bypassing the page cache reduces the I/O latency in case of solid state media, so it makes sense in our case. The kernel system call handler for the *read()* system call is part of the VFS subsystem[75]. The VFS subsystem will translate the read request, which looks like "read those bytes from this open file and return the data into this user space buffer" into a bunch of Block I/O requests (or simply bio requests implemeted by *struct bio* in <*linux/bio.h*>), which are something like "read those sectors from this partition in this block device and return the data in the user space buffer". These bio requests are formed by the VFS layer and passed down to the Block Layer[76]. The user space I/O buffer is represented inside the bio structure as an I/O vector, since the user space buffer may correspond to multiple discontiguous physical memory segments.

The Block Layer is the kernel component that is responsible for passing the bio requests from the VFS layer to the appropriate block device driver. The Block Layer processes the bio requests in two phases. Inherently, the Block Layer is split into two parts: the bio layer and the request layer. The bio layer provides an upstream interface

to filesystems, allowing them to access a multitude of storage backends in a uniform manner. The VFS passes the bio requests to the bio layer. The bio layer incorporates some higher-level services (device mapper, LVM, etc.). In our case, we assume that the bio layer doesn't modify the bio requests. It just forwards them to the request layer.

The request layer[77] performs the scheduling of the I/O requests. It essentially handles the request queues of the underlying block devices. It determines the order of the requests in the queue and the moment that each request is dispatched to the underlying block device driver. Each request queue is filled with requests from the bio layer. In specific, the bio layer creates a request instance (*struct request* defined in <*linux/blkdev.h*>) for each bio request and passes it to the appropriate request queue. Note that each request can be composed of more than one bios, because individual requests can operate on multiple consecutive disk blocks. The request queues of all block devices are being handled by the I/O scheduler. The request layer performs I/O scheduling in order to maximize the I/O performance in case of non-random access hard disk drives. Maximizing the I/O performance practically means minimizing disk seeks when possible. The I/O scheduler has to perform three tasks [79]:

1. coalesce multiple physically contiguous (that means requests concerning contiguous disk sectors) bio requests into a single request. This is referred to as **"merging"**.

2. reorder the requests in such an order that minimizes the seek time, while not delaying important requests unduly. This is referred to as **"sorting"**. Providing an optimal solution to this problem is the source of all the complexity. There are various algorithms for this task. Some of the available I/O schedulers (also called elevators) in Linux are: CFQ, NOOP, Deadline, etc.

3. make these requests available to the underlying driver, so it can pluck them off the queue when it is ready and to provide a mechanism for notification when those requests are complete.

In addition, the I/O scheduler can operate on the requests in order to apply policies for fairness and bandwidth limits.

So, the request sublayer of the Block Layer performs some fancy optimizations with the bio requests. Note that here we assumed that the blk-mq feature is not enabled in

the master guest kernel. Eventually, a bio will be handed to the *request* handler[8] of the block device driver of the target block device. In our case, the underlying device is a SCSI disk, so the block device driver that handles this disk is the sd driver. The sd driver is an upper level driver in the Linux SCSI Subsystem. The sd driver creates a SCSI request from the bio request. It basically uses the last bytes of the bio structure to create a SCSI CDB structure for this bio request. The CDB is the metadata that describe a SCSI command. It contains an opcode (eg. READ, WRITE, INQUIRY), a tag (unique identifier for this request), the target LUN, a logical block number (the first logical block accessed by this command), the transfer length (the number of contiguous logical blocks of data that shall be read and trasferred to the data-in buffer) and some other information. The physical address of the user space I/O buffer can be found in the bio request structure. The sd driver passes the SCSI request to the SCSI mid-layer. The mid-layer inserts the SCSI request in the command queue and eventually passes the request futher down to the low-level SCSI driver. In our case, the low-level SCSI driver is the virtio-scsi driver. The virtio-scsi driver transforms each SCSI request into a virtio-scsi request. A virtio-scsi request has the following format:

```
1  struct virtio_scsi_req_cmd {
2          // Device-readable part
3          u8 lun[8];
4          le64 id;
5          u8 task_attr;
6          u8 prio;
7          u8 crn;
8          u8 cdb[cdb_size];
9          // The next three fields are only present if
           ↪  VIRTIO_SCSI_F_T10_PI
10         // is negotiated.
11         le32 pi_bytesout;
12         le32 pi_bytesin;
13         u8 pi_out[pi_bytesout];
14         u8 dataout[];
15
16         // Device-writable part
17         le32 sense_len;
```

```
        le32 residual;
        le16 status_qualifier;
        u8 status;
        u8 response;
        u8 sense[sense_size];
        // The next field is only present if VIRTIO_SCSI_F_T10_PI
        // is negotiated
        u8 pi_in[pi_bytesin];
        u8 datain[];
};
```

A virtqueue generally consists of an array of descriptors. Each descriptor has the following format:

```
struct virtq_desc {
        /* Address (guest-physical). */
        le64 addr;
        /* Length. */
        le32 len;
/* This marks a buffer as continuing via the next field. */
#define VIRTQ_DESC_F_NEXT   1
/* This marks a buffer as device write-only (otherwise device
read-only). */
#define VIRTQ_DESC_F_WRITE     2
        /* The flags as indicated above. */
        le16 flags;
        /* Next field if flags & NEXT */
        le16 next;
};
```

Essentially, a virtio descriptor is a pointer to an I/O request. So, each I/O needs to be converted into a chain of such descriptors. In our case, an I/O request is represented by *struct virtio_scsi_req_cmd*. A single descriptor can be either readable or writable, so each I/O request consists of at least two (request + response) descriptors. It is up to the driver to partition the I/O request into a virtio descriptor chain. In our case,

the virtio-scsi driver transforms each received SCSI request into a virtio-scsi request and then it maps each virtio-scsi request into a descriptor chain in one of the device's request virtqueues (a virtio-scsi device may have multiple request queues). According to the virtio spec, a descriptor chain is a set of descriptors chained together via the *next* field. The virtio-scsi driver transforms the I/O requests into descriptor chains as follows:

each request must have at least two descriptors (request and response). The first descriptor points to the SCSI command request, which is defined as follows:

```
/* SCSI command request, followed by data-out */
struct virtio_scsi_cmd_req {
        __u8 lun[8];            /* Logical Unit Number */
        __virtio64 tag;         /* Command identifier */
        __u8 task_attr;         /* Task attribute */
        __u8 prio;              /* SAM command priority field */
        __u8 crn;
        __u8 cdb[VIRTIO_SCSI_CDB_SIZE];
} __attribute__((packed));
```

The second descriptor points to the SCSI command response, which is defined as follows:

```
/* Response, followed by sense data and data-in */
struct virtio_scsi_cmd_resp {
        __virtio32 sense_len;           /* Sense data length */
        __virtio32 resid;               /* Residual bytes in data buffer
        ↪  */
        __virtio16 status_qualifier;    /* Status qualifier */
        __u8 status;            /* Command completion status */
        __u8 response;          /* Response values */
        __u8 sense[VIRTIO_SCSI_SENSE_SIZE];
} __attribute__((packed));
```

The existence of any more descriptors depends on the command type. In case of a SCSI READ command, any more descriptors point to the data_out buffers.

Upon finishing with the descriptors, the virtio-scsi driver adds the head index of the new descriptor chain to the available ring and increments the avail ring's head pointer by 1. The virtio-scsi driver doesn't kick the virtqueue in order to notify the backend driver, because the SPDK vhost target has suppressed the guest notifications with the *VIRTIO_F_NO_INTERRUPT* feature flag. Thus, there are no VMEXITs in the datapath due to device notifications.

On the slave side, the SPDK vhost-scsi device pollers poll on the master guest virtqueues for any new I/O requests. According to the virtio spec, a virtio-scsi device has at least 3 virtqueues: a control queue, an event queue and one or more request queues. The control queue and the event queue are used for management operations. The request queues are used for transferring the SCSI requests. SPDK uses two pollers for all of these virtqueues. Specifically, one poller for the control and event queues and one poller for the request queues. The former is function *vdev_mgmt_worker()* and the latter is function *vdev_worker()*. Both of them are defined in *lib/vhost/vhost_scsi.c*. These pollers are being registered to the main reactor thread after the vhost-user feature negotiation. Specifically, this is done by function *spdk_vhost_scsi_start_cb()*.

As a reminder, the virtqueues are allocated in master guest memory, which is exposed to the slave guest as a portion of an MMIO BAR of the virtio-vhost-user device. This implies that virtqueue polling is actually memory mapped I/O on a memory region of the virto-vhost-user device. It is important to note at this point that these MMIO accesses do not cause VMEXITs on the slave side, because this portion of the MMIO BAR is RAM-backed. As a result, whenever the slave guest performs a memory access to this MMIO RAM-backed BAR, then the HW MMU finds a valid PTE in the Extended Page Table ([78] Chapter 28) and performs the translation from guest virtual address to host physical address. In other words, this PTE maps a portion of the MMIO BAR 2 of the virtio-vhost-user device to the host memory frames that make up the master VM's file-backed memory.

When the SPDK vhost-scsi device poller notices a new descriptor chain in the request virtqueue, it needs to translate and transform it back into the original request form. This is done by function *task_data_setup()*. In detail, for each descriptor, the device performs a lookup in the vhost-user memory region table and goes through a gpa_-to_vva translation (master guest physical address to slave guest vhost virtual address).

This is done as follows:

the metadata, that is the descriptor addresses and the addresses contained in the request and response structures, are master GPAs. For each master GPA, the vhost target subtracts the master GPA where the master guest physical memory is mapped in master guest physical memory address space. The remaining offset is then added in the slave guest virtual address where the vhost target has mapped the shared memory region of the virtio-vhost-user device.

SPDK enforces the request and response data to be contained within a single memory region. I/O buffers do not have such limitations. In case an I/O buffer is scattered into multiple discontiguous vhost memory regions, SPDK automatically performs additional iovec splitting and gpa_to_vva translations. After forming the request structs, SPDK forwards such I/O to the SCSI layer.

The SPDK SCSI layer does the SCSI emulation, that is basically mapping the target LUN to the corresponding bdev. The SCSI emulation layer is also responsible to reply to incoming INQUIRY commands (recall that the SPDK vhost-scsi target can work over various non-SCSI storage backends). The SCSI layer then passes the SCSI request to the SPDK bdev layer. The SPDK bdev layer will handoff the SCSI request to the bdev module that handles the storage backend that the request's target LUN is associated with. In this case, the bdev module is the poll-mode virtio-scsi driver that handles the virtio-scsi storage backend.

The I/O request contains slave guest virtual addresses. The DMA-capable storage backend that will serve the I/O request understands slave guest physical addresses or slave IOVAs if a vIOMMU is present. In either case, an address translation is necessary. For this purpose, SPDK maintains its own user space "page tables" or "memory maps". So, slave GVA $\rightarrow$ slave GPA (or slave IOVA) translation is done using the SPDK vtophys memory map. Specifically, in case of the virtio-vhost-user device, SPDK uses the sysfs to find the slave guest physical address where the device's memory is mapped in slave guest physical memory address space.

The bdev module sends the I/O request to the storage backend and it polls for the completion. The storage backend could be any device, passed-through or emulated, that SPDK has a driver for. The storage backend will perform DMA to the I/O buffers in master guest memory. In case the storage backend is emulated by QEMU, then this is

peer-to-peer DMA. In case the storage backend is a physical passed-through device, then the device performs DMA directly from the master guest memory. Note that in this case, if there is an IOMMU in the system and the passthrough has been implemented with the vfio driver, then the DMA operation is possible if the slave QEMU has setup the host IOMMU with GPA-to-HPA mappings to the MMIO BAR 2 of the virtio-vhost-user device. Eventually, the storage backend will finish handling the I/O request. The SPDK bdev module will notice that by polling for the I/O completion. Once the I/O request is completed, the bdev module will notify the SPDK vhost-scsi device. The vhost-scsi device will fill the virtio_scsi response buffer with proper data, will add the head index of the descriptor chain to the used ring and then will interrupt the master guest virtio-scsi driver by kicking the corresponding doorbell. The doorbell is an MMIO address in the MMIO BAR 2 of the virtio-vhost-user device. The doorbells of the virtio-vhost-user device have been hooked up as ioeventfds by QEMU. So, the doorbell kick will cause a VMEXIT, KVM will kick the callfd that corresponds to the guest MMIO address and then will resume executing guest code.

The callfd kick will be captured by KVM in the master side, since the callfds have been bound to IRQs with the KVM irqfd mechanism. Therefore, the KVM will inject a virtual interrupt in the master VM. The interrupt is delivered to the virtio-scsi driver. As a result, the virtio-scsi driver in the master guest kernel gets notified about the completion of the I/O request from the vhost device backend.

In conclusion, it is important to emphasize on two properties of this whole I/O mechanism:

1. the virtio-vhost-user communication mechanism bypasses almost completely the hypervisor in the datapath. There are two key-points though that involve the hypervisor (KVM):

   (a) the I/O completion notifications via the callfds in the slave VM.

   (b) the virtual interrupts for the I/O completions in the master VM.

   Both of these could be bypassed if a poll-mode virtio-scsi driver were used in the master guest kernel/user space instead of the default kernel virtio-scsi driver.

2. the DMA operation from the storage backend is completely zero-copy. This means that there is no data copying in the I/O datapath. The data are being

trasferred directly from the master guest memory to the device memory via the device DMA engine. What is actually being copied is the metadata for each I/O request. In case of a vhost-scsi device, the metadata is the CDB for each SCSI request. Zero-copy DMA applies to most of the SPDK bdevs, but not for all of them. For example, it is true for local NVMe bdevs and virtio-scsi bdevs, but not for NVMe over Fabrics. Moreover, the master guest may give misaligned I/O buffers in which case copying is necessary.

6 Implementation of SPDK/VVU

6.1 Brief Overview

Implementing SPDK/VVU involved working on multiple projects simultaneously and interacting with the corresponding communities. In brief, our work involved working on the Virtio specification, QEMU, SPDK and DPDK. Here is an outline of our work:

1. Virtio Spec: update Stefan's RFC implementation and submit a patchset in virtio-dev mailing list. Until present, we have submitted multiple versions that incorporate the feedback from the community. At the moment, we are waiting for reviews on the fourth version of the patchset.

2. QEMU: update Stefan's RFC implementation of the QEMU device code to comply with the revised spec. The goal is to get the device merged into upstream QEMU. The review process will start soon after the device spec gets approved.

3. DPDK:

   (a) librte_vhost: add support for the virtio-vhost-user transport

   (b) introduce the virtio-vhost-user device driver

   (c) update the testpmd and vhost-scsi applications so that they support choosing among the available vhost-user transports

4. SPDK:

(a) vhost: integrate the virtio-vhost-user transport (auto-detect transport based on vhost controller's name)

(b) add support for vfio no-IOMMU mode

(c) support registering non-2MB aligned virtual addresses in the vtophys map

(d) register the virtio-vhost-user device as a DMA-capable device

In the following sections, we are analyzing the implementation details.

6.2 Changes in the virtio-vhost-user device specification

Prior to this thesis, Stefan Hajnoczi had already introduced an RFC implementation[1] for the virtio-vhost-user device specification. We used this as a reference and incorporated some changes. These changes are the following:

1. rebased the spec on top of the latest master branch. This basically involved moving the device's spec out of the main tex file and into a separate file.
2. fixed some minor bugs in the spec.
3. added some device/driver requirements for the notification capability. The configuration structure of the notification capability has certain requirements, similar to those of the MSI-X capability.
4. updated the shared memory capability so that it gets synced with the *VIRTIO_PCI_CAP_SHARED_MEMORY_CFG* capability. Alan Gilbert has already submitted a spec for shared memory regions over multiple transports (PCI, MMIO). This is expected to be merged soon.
5. added conformance targets for the virtio-vhost-user device. This is just a short list of all the device/driver requirements that a device/driver implementation shall comply with.

We then submitted the revised version to the virtio-dev mailing list for review. Until present, Stefan has reviewed the series. We have incorporated his comments along with

[1] `https://lists.oasis-open.org/archives/virtio-dev/201801/msg00110.html`

some other changes in the patchset and we have re-submitted subsequent versions of the patchset. At the moment, we are on the fourth version of the patchset and we are awaiting for review comments on this. The community is expected to start reviewing more actively right after Alan's patchset has been approved. The reason is that my patchset depends on his.

6.3 Changes in the virtio-vhost-user device code

6.3.1 Architecture of the virtio-vhost-user PCI device

The virtio-vhost-user device code is located in *hw/virtio/*. There is a PCI-specific part in *virtio-vhost-user-pci.c* that implements all the PCI-related operations (capabilities, additional resources, interrupts, etc.) and the backend part in *virtio-vhost-user.c* that implements the core device functionality (socket I/O handling, vhost-user message parsing, virtqueue operations). There is also a header file (*include/hw/virtio/virtio-vhost-user.h*) with all the necessary structure definitions.

6.3.2 Improvements in the QEMU device code

As we have already mentioned, Stefan has proposed an RFC implementation [2] for the virtio-vhost-user device. However, this is by no means complete, because it is just a Proof of Concept. So, given the revised device spec, we worked on Stefan's RFC device implementation. We added some changes and improvements. Our primary concern was to get the code aligned with the device specification. In specific, our changes are the following:

Separate PCI transport-specific code from the backend code

virtio devices are split into two layers:

- transport (how the virtio device connects to the guest), which could be PCI, or MMIO, or S390 device channels

[2] `https://lists.nongnu.org/archive/html/qemu-devel/2018-01/msg04806.html`

- backend (block, net, etc)

The two layers are connected via a virtio bus, which is 1-1 (ie. connects exactly one backend to one transport)

Here, *virtio-vhost-user-device* is the backend, while *virtio-vhost-user-pci* is the backend along with the transport. The declarations for these two device types comply with the QEMU Object Model (QOM) and are the following:

```
static const TypeInfo virtio_vhost_user_info = {
    .name = TYPE_VIRTIO_VHOST_USER,
    .parent = TYPE_VIRTIO_DEVICE,
    .instance_size = sizeof(VirtIOVhostUser),
    .class_init = virtio_vhost_user_class_init,
};

static const VirtioPCIDeviceTypeInfo virtio_vhost_user_pci_info = {
    .base_name     = TYPE_VIRTIO_VHOST_USER_PCI,
    .generic_name  = "virtio-vhost-user-pci",
    .instance_size = sizeof(VirtIOVhostUserPCI),
    .instance_init = virtio_vhost_user_pci_initfn,
    .class_size    = sizeof(VirtioVhostUserPCIClass),
    .class_init    = virtio_vhost_user_pci_class_init,
};
```

Almost always the end user doesn't need to care about the split between backends and transports, because QEMU provides convenient wrappers like virtio-vhost-user-pci, which are a PCI transport plus a backend already connected to each other and wrapped up in a handy single device package.

Concerning the virtio-vhost-user device, we have extracted the PCI transport-specific code into *virtio-vhost-user-pci.c* and we have kept the backend code in *virtio-vhost-user.c*. This pattern complies with a recent refactoring in QEMU code, where the PCI transport implementations of some devices (scsi, crypto, 9p, etc.) have been extracted from *virtio-pci.c* into separate files in *hw/virtio/*.

Implement slave guest interrupts in response to master virtqueue kicks

According to the device specification, the device must support registering MSI-X interrupts to specific master virtqueue events. This means that whenever a vhost-user kickfd is kicked, the virtio-vhost-user device must support triggering slave guest MSI-X interrupts. This would allow for interrupt-driven vhost-user backends over the virtio-vhost-user transport. The function that triggers the interrupt injection in response to virtqueue kicks is *virtio_vhost_user_guest_notifier_read()* and works as follows:

```
1  /* Handler for the master kickfd notifications. Inject an INTx or MSI-X
   ↪  interrupt
2   * to the guest in response to the master notification. Use the
   ↪  appropriate
3   * vector in the latter case.
4   */
5  void virtio_vhost_user_guest_notifier_read(EventNotifier *n)
6  {
7      struct kickfd *kickfd = container_of(n, struct kickfd,
       ↪  guest_notifier);
8      VirtIODevice *vdev = kickfd->vdev;
9      VirtIOVhostUser *vvu = container_of(vdev, struct VirtIOVhostUser,
       ↪  parent_obj);
10     VirtIOVhostUserPCI *vvup = container_of(vvu, struct
       ↪  VirtIOVhostUserPCI, vdev);
11     VirtIOPCIProxy *proxy = &vvup->parent_obj;
12     PCIDevice *pci_dev = &proxy->pci_dev;
13
14     if (event_notifier_test_and_clear(n)) {
15         /* The ISR status register is used only for INTx interrupts.
           ↪  Thus, we
16          * use it only in this case.
17          */
18         if (!msix_enabled(pci_dev)) {
19             virtio_set_isr(vdev, 0x2);
20         }
```

```
21          /* Send an interrupt, either with INTx or MSI-X mechanism.
            ↪  msix_notify()
22           * already handles the case where the MSI-X vector is NO_VECTOR
    ↪  by not issuing
23           * interrupts. Thus, we don't have to check this case here.
24           */
25          virtio_notify_vector(vdev, kickfd->msi_vector);
26
27
            ↪  trace_virtio_vhost_user_guest_notifier_read(kickfd->guest_notifier.rfd,
28                                                         kickfd->msi_vector);
29      }
30  }
```

This function is being registered as an eventfd handler for each kickfd whenever a new *VHOST_USER_SET_VRING_KICK* message arrives from the master (function *m2s_set_vring_kick()*). In other words, this is the callback that is being called whenever a new event arrives on a kickfd. This callback is hooked up to the kickfd by *event_notifier_set_handler()* function. In QEMU, an event notifier is just a wrapper structure for an eventfd. The code for the association of a kickfd with a callback looks like this:

```
1  /* Initialize the EventNotifier with the received kickfd */
2  event_notifier_init_fd(&s->kickfds[vq_idx].guest_notifier, fd);
3
4  /* Insert the kickfd in the main event loop */
5  if (fd != -1) {
6      event_notifier_set_handler(&s->kickfds[vq_idx].guest_notifier,
7                                 virtio_vhost_user_guest_notifier_read);
```

Use ioeventfd mechanism for the callfds

The callfds are the eventfds that are being used in order to emulate the device interrupts. The vhost device kicks the callfds in order to signal the completion of an I/O request. The virtio-vhost-user device exports this mechanism to the slave guest via doorbells. This means that whenever the slave guest vhost device kicks a doorbell,

which is actually memory-mapped I/O, QEMU emulates the MMIO by kicking the callfd that corresponds to this doorbell. The problem with this mechanism is that the emulation overhead for each doorbell kick is critical, given the fact that it is part of the I/O datapath. What we did to reduce this overhead was to shift the MMIO emulation from QEMU to KVM. This has been achieved by using the ioeventfd mechanism offered by KVM. With the ioeventfd mechanism someone can hook up an eventfd with an MMIO/PIO access. This means that KVM triggers eventfd kicks in response to guest MMIO/PIO accesses. In this way, we avoid the context switch between host kernel space and host user space. In the code, we are registering the callfds as ioeventfds in function *virtio_vhost_user_register_doorbell()*. This function is just a wrapper for *vvu_register_doorbell()*, which is part of the virtio-vhost-user-pci device class. It uses function *memory_region_add_eventfd()* to register the ioeventfd.

```
static void vvu_register_doorbell(VirtIOVhostUserPCI *vvup,
↪  EventNotifier *e, uint8_t vq_idx)
{
    VirtIOPCIProxy *proxy = &vvup->parent_obj;
    hwaddr addr = vq_idx * virtio_pci_queue_mem_mult(proxy);

    /* Register the callfd EventNotifier as ioeventfd */
    memory_region_add_eventfd(&vvup->doorbells.mr, addr, 2, false,
    ↪  vq_idx, e);
}
```

The function *virtio_vhost_user_register_doorbell()* is called by *m2s_set_vring_call()*, that is whenever the master sends a *VHOST_USER_SET_VRING_CALL* message. The code looks like this:

```
/* Initialize the EventNotifier with the received callfd */
event_notifier_init_fd(&s->callfds[vq_idx], fd);

/* Register the EventNotifier as an ioeventfd. */
if (fd != -1) {
    virtio_vhost_user_register_doorbell(s, &s->callfds[vq_idx], vq_idx);
}
```

Implement virtio PCI capabilities for the additional device resources

The virtio-vhost-user device differs from the other virtio devices in that it requires some additional resources. As a result, implementing the device specification in one of the available virtio transports requires implementing these resources. Since it is not certain whether it is possible for all the virtio transports to support these resources, the device specification currently defines how these resources can be standardized and exposed over the PCI transport. According to the device spec, in case of the PCI transport, the additional device resources are being standardized with virtio PCI capabilities. The initial RFC implementation didn't include these capabilities. So we implemented these capabilities, thereby getting the code synchronized with the specification.

The virtio-vhost-user device uses three additional resources, namely "doorbells", "notifications" and "shared memory". Each one of these is standardized via a virtio PCI capability. So, what we need to implement is the virtio PCI capabilities and the *read()*/*write()* MMIO handlers for the corresponding configuration structures. Since both of these are PCI transport-specific operations, the relevant code is part of *virtio-vhost-user-pci.c*.

The capabilities and the corresponding configuration structures are being initialized by function *virtio_vhost_user_init_bar()*, which is called during the instantiation of the virtio-vhost-user-pci device. The additional device resources are placed in MMIO BAR 2. This BAR is initialized as follows:

```
1  const int bar_index = 2;
2  const uint64_t bar_size = 1ULL << 36;
3
4  memory_region_init(&vvup->additional_resources_bar, OBJECT(vvup),
5                     "virtio-vhost-user", bar_size);
6  pci_register_bar(&vvup->parent_obj.pci_dev, bar_index,
7                   PCI_BASE_ADDRESS_SPACE_MEMORY |
8                   PCI_BASE_ADDRESS_MEM_PREFETCH |
9                   PCI_BASE_ADDRESS_MEM_TYPE_64,
10                  &vvup->additional_resources_bar);
```

The configuration structures for the additional device resources are initialized as follows:

```
/* Initialize the VirtIOPCIRegions for the virtio configuration structures
 * corresponding to the additional device resource capabilities.
 * Place the additional device resources in the additional_resources_bar.
 */
VirtIOPCIProxy *proxy = VIRTIO_PCI(vvup);

vvup->doorbells.offset = 0x0;
vvup->doorbells.size = virtio_pci_queue_mem_mult(proxy) *
                       (VIRTIO_QUEUE_MAX + 1 /* logfd */);
vvup->doorbells.size = QEMU_ALIGN_UP(vvup->doorbells.size, 4096);
vvup->doorbells.type = VIRTIO_PCI_CAP_DOORBELL_CFG;

vvup->notifications.offset = vvup->doorbells.offset +
 vvup->doorbells.size;
vvup->notifications.size = 0x1000;
vvup->notifications.type = VIRTIO_PCI_CAP_NOTIFICATION_CFG;

/* cap.offset and cap.length must be 4096-byte (0x1000) aligned. */

vvup->shared_memory.offset = vvup->notifications.offset +
 vvup->notifications.size;
vvup->shared_memory.offset = QEMU_ALIGN_UP(vvup->shared_memory.offset,
 4096);

/* The size of the shared memory region in the additional resources BAR doesn't
 * fit into the length field (uint32_t) of the virtio capability structure.
 * However, we don't need to pass this information to the guest driver via
```

```
25  * the shared memory capability because the guest can figure out the
    ↪  length of
26  * the vhost memory regions from the SET_MEM_TABLE vhost-user messages.
    ↪  Therefore,
27  * the size of the shared memory region that we are declaring here has
    ↪  no
28  * meaning and the guest driver shouldn't rely on this.
29  */
30
31 vvup->shared_memory.size = 0x1000;
32 vvup->shared_memory.type = VIRTIO_PCI_CAP_SHARED_MEMORY_CFG;
```

Here *doorbells*, *notifications* and *shared_memory* are *VirtIOPCIRegion* instances that describe *MemoryRegions* for the configuration structures along with their location in device memory. The *offset* parameter is the starting address in device memory, the *size* is the total size of the configuration structure and the *type* is the type of the virtio configuration structure.

For each MemoryRegion we need a set of *read()*/*write()* handlers for the MMIO emulation. These are declared as follows:

```
1  /* Initialize the MMIO MemoryRegions for the additional device
   ↪  resources. */
2
3  static struct MemoryRegionOps doorbell_ops = {
4      .read = virtio_vhost_user_doorbells_read,
5     .write = virtio_vhost_user_doorbells_write,
6     .impl = {
7         .min_access_size = 1,
8         .max_access_size = 4,
9     },
10    .endianness = DEVICE_LITTLE_ENDIAN,
11 };
12
13 static struct MemoryRegionOps notification_ops = {
14     .read = virtio_vhost_user_notification_read,
```

```
15      .write = virtio_vhost_user_notification_write,
16      .impl = {
17          .min_access_size = 1,
18          .max_access_size = 4,
19      },
20      .endianness = DEVICE_LITTLE_ENDIAN,
21  };
```

Notice that we do not use any *read()*/*write()* handlers for the shared memory region. The reason is that the shared memory region is RAM-backed.

The MemoryRegions for the configuration structures are being initialized as follows:

```
1  memory_region_init_io(&vvup->doorbells.mr, OBJECT(vvup),
2                        &doorbell_ops, vvup,
3                        "virtio-vhost-user-doorbell-cfg",
4                        vvup->doorbells.size);
5
6  memory_region_init_io(&vvup->notifications.mr, OBJECT(vvup),
7                        &notification_ops, vvup,
8                        "virtio-vhost-user-notification-cfg",
9                        vvup->notifications.size);
```

This basically relates the MemoryRegions with the callbacks.

The last operation is to register the configuration structures that where previously initialized. This involves registering the MemoryRegions as subregions of the MMIO BAR 2 and creating virtio PCI capabilities. The code is the following:

```
1  /* Register the virtio PCI configuration structures
2   * for the additional device resources. This involves
3   * registering the corresponding MemoryRegions as
4   * subregions of the additional_resources_bar and creating
5   * virtio capabilities.
6   */
7  struct virtio_pci_cap cap = {
```

```
    .cap_len = sizeof cap,
};
struct virtio_pci_doorbell_cap doorbell = {
    .cap.cap_len = sizeof doorbell,
    .doorbell_off_multiplier =
        cpu_to_le32(virtio_pci_queue_mem_mult(proxy)),
};
virtio_pci_modern_region_map(proxy, &vvup->doorbells, &doorbell.cap,
                             &vvup->additional_resources_bar,
                             ↪ bar_index);
virtio_pci_modern_region_map(proxy, &vvup->notifications, &cap,
                             &vvup->additional_resources_bar,
                             ↪ bar_index);
virtio_pci_modern_region_map(proxy, &vvup->shared_memory, &cap,
                             &vvup->additional_resources_bar,
                             ↪ bar_index);
```

Implement UUID configuration field

According to the device spec, the virtio-vhost-user device exports a UUID through the device-specific configuration structure, thus allowing for stable device identification by the guest driver, regardless of the assigned PCI address. The UUID is created during the device realization. The code goes as follows:

```
static void virtio_vhost_user_device_realize(DeviceState *dev, Error
↪ **errp)
{
VirtIODevice *vdev = VIRTIO_DEVICE(dev);
VirtIOVhostUser *s = VIRTIO_VHOST_USER(dev);

[...]

/* Generate a uuid */
QemuUUID uuid;
qemu_uuid_generate(&uuid);
memcpy(s->config.uuid, uuid.data, sizeof(uuid.data));
```

```
12
13 [...]
14 }
```

The device-specific configuration structure is the following:

```
1 /* The virtio configuration space fields */
2 typedef struct {
3     uint32_t status;
4 #define VIRTIO_VHOST_USER_STATUS_SLAVE_UP 0
5 #define VIRTIO_VHOST_USER_STATUS_MASTER_UP 1
6     uint32_t max_vhost_queues;
7     uint8_t uuid[16];
8 } QEMU_PACKED VirtIOVhostUserConfig;
```

6.4 Changes in DPDK

DPDK's *librte_vhost* is the library that implements the vhost-user control plane.The following sections give a detailed description of the changes that were made in this library in order to support the virtio-vhost-user transport.

6.4.1 Introduce vhost transport operations structure

In the vhost-user protocol specification, there is no notion of a vhost-user transport. The control plane is based on file descriptor handling (unix sockets, kickfds, callfds, fds for the vhost memory regions, fds for the log memory regions) and socket I/O (vhost-user messages).

The virtio-vhost-user device implements a different mechanism for the vhost-user control plane. The virtio-vhost-user device turns the socket I/O into virtqueue operations. It also handles the task of exposing master VM's resources (kickfds, callfds, vhost memory regions) to the slave guest via PCI device resources.

In order to incorporate both of these transport implementations in *librte_vhost*, we have introduced the following generic interface for the vhost transport operations:

```
struct vhost_transport_ops {
        size_t socket_size;
        size_t device_size;

        int (*socket_init)(struct vhost_user_socket *vsocket, uint64_t
        ↪  flags);
        void (*socket_cleanup)(struct vhost_user_socket *vsocket);
        int (*socket_start)(struct vhost_user_socket *vsocket);
        void (*cleanup_device)(struct virtio_net *dev, int destroy);
        int (*vring_call)(struct virtio_net *dev, struct vhost_virtqueue
        ↪  *vq);
        int (*send_reply)(struct virtio_net *dev, struct VhostUserMsg
        ↪  *reply);
        int (*send_slave_req)(struct virtio_net *dev,
                              struct VhostUserMsg *req);
        int (*process_slave_message_reply)(struct virtio_net *dev,
                                           const struct VhostUserMsg
                                           ↪  *msg);
        int (*set_slave_req_fd)(struct virtio_net *dev,
                                struct VhostUserMsg *msg);
        int (*map_mem_regions)(struct virtio_net *dev,
                               struct VhostUserMsg *msg);
        void (*unmap_mem_regions)(struct virtio_net *dev);
        int (*set_log_base)(struct virtio_net *dev,
                            const struct VhostUserMsg *msg);
        int (*set_postcopy_advise)(struct virtio_net *dev,
                                   struct VhostUserMsg *msg);
        int (*set_postcopy_listen)(struct virtio_net *dev);
        int (*set_postcopy_end)(struct virtio_net *dev,
                                struct VhostUserMsg *msg);
};
```

This interface is a set of function pointers that each transport should implement.

6.4.2 Extract AF_UNIX-specific code from core vhost-user code

The goal is to replace the AF_UNIX-specific code with calls to the transport operations. The AF_UNIX-specific code is everything related to socket I/O and file descriptor handling. The former involves reading and writing vhost-user messages to unix domain sockets. The latter primarily involves the socket management (listening on the server socket, accepting new connections, closing connections), manipulating the kickfds/callfds for master-slave notifications and mmaping/munmapping vhost memory regions.

The AF_UNIX-specific code is located in *socket.c* and *vhost_user.c*. The code in *socket.c* serves two purposes:

1. hold the *librte_vhost* public entry points

2. perform socket management

The main purpose of *vhost_user.c* is the handling of the vhost-user messages.

We moved all socket management code from *socket.c* into *trans_af_unix.c*. We also moved all the fd-related operations from *vhost_user.c* into *trans_af_unix.c*. The fd-related operations are mmaping/munmapping the vhost memory regions and setting up a userfaultfd for postcopy live migration.

6.4.3 Introduce the virtio-vhost-user driver and transport

The virtio-vhost-user transport relies on the virtio-vhost-user device for the master-slave communication. So, a user space driver for this device is necessary. The driver code cannot be placed in *librte_vhost*, because it depends on *rte_bus_pci.h* header file, which gets exported when *drivers/* is built, and *drivers/* is built after *lib/*. Therefore, we have created the folder *drivers/virtio_vhost_user/* and inserted the driver code in there. We have also copied the virtio pci code from *drivers/net/virtio/* into *drivers/virtio_vhost_user/*, because there is no public *lib/virtio/* library that we could use. We have enhanced the virtio pci code so that it handles the virtio PCI capabilities for the additional device resources.

Along with the driver, we have inserted the virtio-vhost-user transport implementation. Both the driver and the transport code are located in file *trans_virtio_vhost_user.c*. The virtio-vhost-user transport needs to have access to some internal functions and data structures of *librte_vhost*, so we have made *vhost.h* and *vhost_user.h* part of *librte_vhost's* public API. Finally, we have updated the Makefiles so that the new transport gets built as expected and gets linked with the apps, both in case of static and in case of shared library build.

In case, of shared library build, we used the *–no-as-needed* flag of the GNU linker. The reason is that we want the virtio_vhost_user library to be linked with the apps unconditionally, although there is no symbol referring explicitly to this library.

6.4.4 Export the virtio-vhost-user transport through librte_vhost public API

We need a mechanism so that *librte_vhost* can allow choosing between the available vhost-user transports. We have added a global transport map in ‘vhost.h‘ which holds pointers to the vhost transport operations of all the available transports.

```
typedef enum VhostUserTransport {
        VHOST_TRANSPORT_UNIX = 0,
        VHOST_TRANSPORT_VVU = 1,
        VHOST_TRANSPORT_MAX = 2
} VhostUserTransport;

/* A list with all the available vhost-user transports. */
const struct vhost_transport_ops *g_transport_map[VHOST_TRANSPORT_MAX];
```

New transports can be registered with *rte_vhost_register_transport()*, which is part of *librte_vhost* public API.

```
int
rte_vhost_register_transport(VhostUserTransport trans,
                const struct vhost_transport_ops *trans_ops)
{
```

```
        if (trans >= VHOST_TRANSPORT_MAX) {
                RTE_LOG(ERR, VHOST_CONFIG,
                        "Invalid vhost-user transport %d\n", trans);
                return -1;
        }

        g_transport_map[trans] = trans_ops;
        return 0;
}
```

An application can choose transport when it requests for a new vhost device. The public entry point for this is function *rte_vhost_driver_register()*. The virtio-vhost-user transport can be requested by passing the *RTE_VHOST_USER_VIRTIO_TRANSPORT* flag as the second argument.

```
int
rte_vhost_driver_register(const char *path, uint64_t flags)
{
        int ret = -1;
        struct vhost_user_socket *vsocket;
        const struct vhost_transport_ops *trans_ops;

[...]

        if (flags & RTE_VHOST_USER_VIRTIO_TRANSPORT) {
                trans_ops = g_transport_map[VHOST_TRANSPORT_VVU];
                if (trans_ops == NULL) {
                        RTE_LOG(ERR, VHOST_CONFIG,
                                "virtio-vhost-user transport is not
                                ↪  supported\n");
                        goto out;
                }
        } else {
                trans_ops = g_transport_map[VHOST_TRANSPORT_UNIX];
        }
```

```
20
21 [...]
22 }
```

6.4.5 Add virtio-vhost-user devices in dpdk-devbind.py

dpdk-devbind.py is a python script that binds/unbinds devices to the kernel vfio-pci driver. We have added the virtio-vhost-user device's vendor and device ID in the list of devices that this script handles.

6.4.6 Export the virtio-vhost-user transport choice to the end user

The virtio-vhost-user transport is part of DPDK's *librte_vhost* and we are interested in using it from SPDK. However, it should also be possible to use the transport from the DPDK apps. Therefore, we have updated the testpmd app and the vhost-scsi example application so that the end user can choose between the two available transports.

6.5 Changes in SPDK

6.5.1 Integrate the virtio-vhost-user transport in libspdk_vhost

Given the integration of the virtio-vhost-user transport in DPDK's *librte_vhost*, the SPDK vhost library can choose between the transports via the function *rte_vhost_-driver_register()*. However, there is currently no API for the end user to choose transport.

SPDK offers two applications that allow creating vhost targets, the vhost app and the spdk_tgt app. The latter is just a universal application that encapsulates all the SPDK applications, including the vhost app. The user can create vhost targets either at runtime, through JSON RPC calls, or statically, with a configuration file. We have extended both interfaces, so that the transport can be chosen from the vhost controller's name. Specifically, if the controller's name is a DomBDF PCI address, then the user wants to use the virtio-vhost-user transport, and the given PCI address is expected to

be the address of the virtio-vhost-user device. In any other case, the *AF_UNIX* transport is used. The function *spdk_vhost_dev_register()* checks if the controller's name is a PCI address by using the function *spdk_pci_addr_parse()*. The code looks like this:

```
1  int
2  spdk_vhost_dev_register(struct spdk_vhost_dev *vdev, const char *name,
3  const char *mask_str,
4                          const struct spdk_vhost_dev_backend *backend)
5  {
6          char path[PATH_MAX];
7          struct stat file_stat;
8          struct spdk_cpuset *cpumask;
9  #ifndef SPDK_CONFIG_VHOST_INTERNAL_LIB
10         struct spdk_pci_addr pci_addr;
11         uint64_t transport = 0;
12 #endif
13         int rc;
14
15 [...]
16
17 #ifndef SPDK_CONFIG_VHOST_INTERNAL_LIB
18         if (spdk_pci_addr_parse(&pci_addr, name) == 0) {
19                 transport = RTE_VHOST_USER_VIRTIO_TRANSPORT;
20         }
21 [...]
22 }
```

For example, in case of the JSON RPC API, an example use case would look like this:

$ scripts/rpc.py construct_vhost_scsi_controller –cpumask 0x1 vhost.0

$ scripts/rpc.py construct_vhost_scsi_controller –cpumask 0x1 0000:00:07.0

In the first case, we are using the default *AF_UNIX* transport. In the second case, we are using the virtio-vhost-user transport and we are specifying the PCI address of the virtio-vhost-user device that we want to use.

6.5.2 Add support for vfio no-IOMMU mode

SPDK contains a set of user space device drivers controlling physical storage devices. As a result, it completely bypasses the Linux kernel storage stack. SPDK supports both uio and vfio for passing device control to user space.

Vfio is more secure that uio, because it makes use of the IOMMU for DMA translations. It is generally preferred against uio whenever possible. However, in case of the SPDK vhost target with virtio-vhost-user transport, the Storage Appliance VM is dedicated to I/O processing. Therefore, there is no need for DMA protection inside the Storage Appliance VM. In fact, adding a vIOMMU would increase the overall software latency. So, someone would argue that we could use uio in order to eliminate the need for a vIOMMU. The problem is that we can not use uio, because it does not support MSI interrupts, which are used by the virtio-vhost-user device.

Recently, the vfio driver has been enhanced with a new mode of operation called "no-IOMMU mode". In this mode of operation, no DMA translations occur, regardless of the existence or not of an IOMMU in the system. This feature is available on kernel versions newer that 4.5.

So, we fixed the above problem by adding support for vfio no-IOMMU mode in SPDK. This involved two changes in the code. Firstly, we had to make some changes in the setup script (*scripts/setup.sh*). The setup script performs two tasks:

1. allocate hugepage memory that is necessary for the operation of the SPDK process
2. bind all supported PCI devices in one of the uio and vfio kernel drivers

So, what we did was to change the criteria for the choice between uio and vfio drivers. In detail, the setup script chooses between vfio against uio only if an IOMMU is present in the system. It does so by checking for any iommu_groups under */sys/kernel/iommu_groups*. I changed this criterion. In specific, we want vfio to be preferred against uio in case vfio module is loaded in the kernel in no-IOMMU mode. The new check looks like this:

```
1 elif [[ -n "$(ls /sys/kernel/iommu_groups)" || \
2       (-e /sys/module/vfio/parameters/enable_unsafe_noiommu_mode && \
```

```
3       "$(cat /sys/module/vfio/parameters/enable_unsafe_noiommu_mode)" ==
        ↪  "Y") ]]; then
4         driver_name=vfio-pci
```

The second change we had to make was in the vtophys mapping. DMA remapping does not make sense in vfio no-IOMMU mode, because the IOMMU is not used. This implies that physical DMA addresses have to be used instead of IOVAs. So, we have to store physical addresses in the vtophys mapping in case of vfio no-IOMMU. This is actually the same code path as when the uio is used as a kernel driver, where we use physical addresses for the DMA operations as well. So what we did was to check during the vtophys mapping initialization whether vfio no-IOMMU is enabled. In such case, we follow the same code path as with uio. That is use physical addresses for the DMA mappings.

6.5.3 Support registering non-2MB aligned virtual addresses

Description of the SPDK memory map and its relevance with the vhost target

SPDK contains a set of user space drivers for storage devices. Storage devices serve I/O requests with DMA operations. This implies that the device driver has to explicitly ask from the device to perform DMA from a specific physical memory address (or an IOVA address in presence of an IOMMU) where the I/O buffer is stored. However, SPDK is a user space process, thus having its own virtual address space. Therefore, in case DMA is necessary, there is a need for SPDK to find the physical address per virtual address and keep a mapping with the virtual-to-physical memory address translations. And in case there is an IOMMU in the system, SPDK keeps virtual-to-IOVA mappings. In this sense, the SPDK mapping can be considered as a user space page table. In the SPDK's code, page tables are referred to as "memory maps". SPDK defines a 2-level page table structure for its internal memory maps. The first level has 1GB granularity and the second level has 2MB granularity. The definition of the memory map structure lives in *lib/env_dpdk/memory.c* and goes like this:

```
1 /* Translation of a single 2MB page. */
2 struct map_2mb {
```

```
        uint64_t translation_2mb;
};

/* Second-level map table indexed by bits [21..29] of the virtual
address.
 * Each entry contains the address translation or error for entries that
   haven't
 * been retrieved yet.
 */
struct map_1gb {
        struct map_2mb map[1ULL << (SHIFT_1GB - SHIFT_2MB)];
};

/* Top-level map table indexed by bits [30..47] of the virtual address.
 * Each entry points to a second-level map table or NULL.
 */
struct map_256tb {
        struct map_1gb *map[1ULL << (SHIFT_256TB - SHIFT_1GB)];
};

/* Page-granularity memory address translation */
struct spdk_mem_map {
        struct map_256tb map_256tb;
        pthread_mutex_t mutex;
        uint64_t default_translation;
        struct spdk_mem_map_ops ops;
        void *cb_ctx;
        TAILQ_ENTRY(spdk_mem_map) tailq;
};
```

SPDK allocates and uses two distinct memory maps. These are the “registration map” and the “vtophys map”. Their difference comes from the data kept in the *translation_2mb* field. The registration map keeps track of the process virtual address ranges that have been registered as DMA-able memory. The public API for the memory registration is function *spdk_mem_register()*. A 2MB memory range is marked as registered

by using the MSB bit in the *translation_2mb* field of the corresponding 2MB PTE. On the other side, the vtophys map keeps the VA-to-PA (or VA-to-IOVA) translations for each 2MB virtual address range. The PA (or IOVA) is kept in the *translation_2mb* field. The function *spdk_mem_register()* refreshes the vtophys map alongside with the registration map via a notification callback whenever a new address range is registered. This is done in this code section:

```
1 TAILQ_FOREACH(map, &g_spdk_mem_maps, tailq) {
2         rc = map->ops.notify_cb(map->cb_ctx, map,
          ↪  SPDK_MEM_MAP_NOTIFY_REGISTER, seg_vaddr, seg_len);
3         if (rc != 0) {
4                 pthread_mutex_unlock(&g_spdk_mem_map_mutex);
5                 return rc;
6         }
7 }
```

The code for the manipulation of the SPDK vtophys map is located in *lib/env_dpdk/memory.c*. The manipulation of the vtophys map involves finding the PA corresponding to a VA, or using the vfio API to register a virtual address range as DMA-able memory. Both for the registration map and for the vtophys map, each new registered address has to be 2MB aligned due to the 2MB granularity of the SPDK's memory map structure.

Considering the operation of the vhost target, the vhost-user master sends a set of vhost memory regions to the slave containing the driver's virtqueues and I/O buffers in master guest physical memory. Eventually, the processing of the I/O requests from the vhost-user slave will be delivered to a storage device and the storage device will perform DMA directly from master VM's memory. Therefore, the mapped vhost memory regions have to be registered to the vtophys map during the vhost datapath initialization. The registration is done with function *spdk_vhost_session_mem_register()*. This function registers each mapped vhost memory region to the vtophys map. This implies that the vhost memory regions have to be mapped to 2MB aligned virtual addresses.

Description of the Problem

This is not a problem in case of vhost with *AF_UNIX* transport and given that the vhost memory regions are backed by hugepages. In this case, the kernel will map the master VM's memory to a 2MB aligned virtual address in SPDK's process address space for sure, because this is the only way for having a single page table entry per hugepage in the hardware's multi-level page tables.

However, this is not true in case of vhost with virtio-vhost-user transport. In this case, the virtio-vhost-user device maps the vhost memory regions sent by the master and exposes them to the slave guest as an MMIO PCI memory region. So, instead of mapping hugepage backed memory regions, the vhost target, running in slave guest user space, maps segments of an MMIO BAR of the virtio-vhost-user device. Thus, the mapped addresses are not necessarily 2MB aligned. Note that this is generally known as "peer-to-peer DMA". SPDK/VVU is just one type of it.

What is more, note that, in case of vhost with *AF_UNIX* transport, the vhost shared memory is 2MB-aligned given that the VM's memory is hugepage backed. But the restriction of hugepage-backed memory is unnecessary. The whole configuration would still work if the VM's memory was backed by a tmpfs file (normal 4KB pages) given that we use vfio with IOMMU support. The vhost memory regions would be mapped by the SPDK vhost target and then registered to vfio as DMA-able memory with *MAP_DMA* ioctl. Vfio would take care of making this memory DMA-able. This basically involves pinning the memory and updating the device's IOVA domain to grant access to this memory. So, if we support registering non-2MB aligned virtual addresses in the SPDK vtophys map, then we could use normal pages for the master VM's memory (losing though the better performance that hugepages offer).

Solution

To sum up, extending the SPDK memory map structure to allow registering non-2MB aligned mappings would enable:

1. adding support for peer-to-peer DMAs in SPDK
2. using normal 4KB pages for the master VM's memory instead of hugepages

Note that the 2MB granularity of the memory map has created implications in other parts of the project as well. So, it is certain that the memory map will eventually be refactored.

However, refactoring the memory map is not a trivial issue. It requires thoroughly examining possible solutions and reaching to a consensus with the core maintainers. This is definitely going to be time consuming. For the time being, we are resorting to a hack in the DPDK code to overcome this problem. We are explicitly mapping all the BARs of all the PCI devices (including the virtio-vhost-user devices) to 2MB-aligned virtual addresses.

6.5.4 Register the virtio-vhost-user device as a DMA-capable device

Description of the Problem

In SPDK/VVU, the master vhost memory regions are exposed to the slave guest as regions of the memory space of the virtio-vhost-user device. In case of vfio no-IOMMU mode, SPDK stores the VA-to-PA translations in the vtophys map. The physical addresses associated with the vhost memory regions point out to the virtio-vhost-user device memory regions. Therefore, it is necessary that the virtio-vhost-user device is registered to the SPDK's internal PCI device list (*g_vtophys_pci_devices*) so that SPDK can find the VA-to-PA translations. Note that SPDK uses the *sysfs* to find the physical addresses.

Solution

The virtio-vhost-user device is being handled by the virtio-vhost-user driver in DPDK. We need to find the *rte_pci_device* instance (internal representation of a PCI device) associated with the virtio-vhost-user device. For this purpose, we use DPDK's *rte_pci_dev_iterate()* function. This function searches in DPDK's internal PCI device list using a PCI BDF address and, in case of a match, it returns the corresponding *rte_device* instance. We then use function *spdk_vtophys_pci_device_added()* to register the device in SPDK's internal PCI device list.

The code is the following:

```
int
spdk_vhost_dev_register(struct spdk_vhost_dev *vdev, const char *name,
                        const char *mask_str,
                        const struct spdk_vhost_dev_backend *backend)
{

[...]

        /* Register the vvu PCI device to SPDK's internal list of
        ↪  DMA-capable devices.
         * This will enable finding the physical addresses of the
    ↪  vhost-user memory
         * regions in case of vfio no-IOMMU mode. Search for
    ↪  `rte_device` instance
         * in DPDK's internal PCI device list using a key-value pair for
    ↪  the BDF PCI
         * address.
         */
        struct rte_bus *bus;
        struct rte_device *dev;
        struct rte_pci_device *pci_dev;
        char kv_pci_addr[PATH_MAX];
        int key_len = sizeof("addr=") - 1;
        int kv_len = key_len + strlen(name);

        bus = rte_bus_find_by_name("pci");
        if (bus == NULL) {
                SPDK_ERRLOG("Cannot find bus (pci)\n");
                rc = -ENOENT;
                goto out;
        }
        if (snprintf(kv_pci_addr, sizeof(kv_pci_addr), "addr=%s", name)
        ↪  != kv_len) {
                SPDK_ERRLOG("Failed to copy PCI address '%s' for
                ↪  controller.\n", name);
```

```
30                  rc = -EINVAL;
31                  goto out;
32          }
33
34          dev = bus->dev_iterate(NULL, kv_pci_addr, NULL);
35          if (dev == NULL) {
36                  SPDK_ERRLOG("Cannot find virtio-vhost-user device with
                    ↪  BDF PCI address %s\n", name);
37                  rc = -EINVAL;
38                  goto out;
39          }
40          pci_dev = RTE_DEV_TO_PCI(dev);
41          spdk_vtophys_pci_device_added(pci_dev);
42
43  [...]
44
45  }
```

7 Evaluation

7.1 Disk metrics

The most common criterion for comparing various storage virtualization solutions is the virtualization overhead. Virtualization overhead is the software overhead inserted by the device emulation software. In this chapter we are going to have a look at the disk metrics and examine SPDK/VVU's virtualization overhead.

There are three storage metrics for measuring the storage performance. These are:

1. throughput (MB/s)
2. IOPS (reqs/s)
3. latency (ms)

Throughput is the amount of data you can write to a device in a given period of time. IOPS is the number of I/O requests that the device can handle in a given period of time. Latency is the time that the device needs to serve an I/O request.

In some way, all these metrics measure the same thing. However, each metric shows different aspects of the system.

There are also some other metrics that are of great interest in cloud environments. These are:

4. efficiency: this is how many CPUs we need in order to reach a certain amount of IOPS. Increasing efficiency allows for greater VM density, because more CPUs are available for other computations.

5. scalability: this is about scaling out the IOPS by either increasing the number of CPUs that perform I/O or increasing the number of storage devices.

In cloud environments the hardware is emulated. This means that all the above metrics are being affected by the storage virtualization overhead. The storage virtualization overhead and the software overhead in general, did not use to be considered with the older spinning disks. The reason is that spinning disks have orders of magnitude bigger latency that the software. Spinning disks have latency in the order of milliseconds, while the software latency is in the order of microseconds. This has changed. Modern PCIe NVMe SSDs have average latency in the order of microseconds. This implies that the software latency is now proportional to the hardware latency. It is a common misconception that storage virtualization is not a problem anymore, because the media itself has become extremely fast. In fact, the opposite holds. Since the media have become faster, the software overhead has turned into a bottleneck.

It is worth noting that measuring the storage performance is a multi-variable problem. Except for the inherent restrictions of the media and the storage virtualization overhead, one has to tweak with the following parameters to have a complete view:

- number of I/O queues
- queue depth
- number of cores submitting I/O (blk-mq, scsi-mq)
- request size
- I/O pattern (eg. random reads, sequential reads, etc.)
- notification mechanism (interrupts, block layer strict I/O polling, block layer hybrid I/O polling)

7.2 Virtualization I/O overhead

Emulating a storage device inevitably incurs a virtualization overhead. The performance-critical part of this overhead is the overhead in the virtualized I/O datapath. This is what users are noticing in cloud storage services and this is what really matters to be reduced. The overhead in the I/O datapath is coming from the following operations:

- guest notifications

  Guest notifications is the mechanism used by the guest device driver in order to inform the device about new I/O requests. The mechanism for implementing guest notifications determines the emulation overhead. Usually, guest notifications are implemented as Port I/O or Memory Mapped I/O operations on the device resources. However, the actual implementation varies. For example, the virtio specification defines a more lightweight guest notification mechanism by essentially reducing the number of PIO/MMIO operations per I/O request. This is basically the performance advantage of virtio devices against the fully emulated devices. The virtio specification also defines a mechanism so that the device itself can reduce the frequency of notifications from the driver. This feature is called "virtqueue notification suppression" with event index.

- device interrupts

  Device interrupts is the mechanism used by the device in order to inform the guest device driver about the I/O completions. There are various mechanisms for implementing virtual interrupts, each one incurring a different emulation overhead. The overhead of each mechanism depends on the amount of the hypervisor's intervention, which depends on the hardware assistance. In older hardware, the interrupts from passed-through devices where handled by the host kernel (KVM) and injected into the guest. The newer processors allow interrupts to be injected directly into the VM, bypassing the host entirely. The number of interrupts invoked by the device is also important. For example, many devices perform interrupt coalescing, in order to reduce the overall virtualization overhead incurred by interrupt emulation. The virtio specification allows the guest driver to determine the frequency in which the device incurs

interrupts. This feature is called “virtqueue interrupt suppression” with event index.

- dataplane emulation

  The dataplane emulation encapsulates the whole emulation process for each I/O request. As we have already elaborated in section 1.3 of the introductory chapter, there are various approaches and many ways to categorize them. First of all, the dataplane emulation depends on the storage protocol (SCSI, BLK, NVMe). Secondly, it is possible to handle the I/O requests inside QEMU (eg. QEMU SCSI target) or in the host kernel (eg. QEMU LIO Target used by vhost-scsi). Regarding the QEMU user space SCSI target, QEMU can interact with the physical storage backend in many ways through the host kernel system call interface. The possible alternatives are:

  - reading/writing to a file on the host filesystem
  - reading/writing directly to a block device (*/dev/sdX*)
  - sending guest SCSI commands directly to a host SCSI target via the SCSI generic driver (*/dev/sgX*)

  The host kernel interface exposes some mechanisms that QEMU can utilize in order to improve the I/O performance, like asynchronous I/O and direct I/O. What is more, it is possible to assign a physical device on the host directly to a guest, either using the legacy passthrough interface (pci-assign) or using the vfio driver. There is also a slight variation of this that relies on the host kernel to split a device into multiple virtual devices. This is called “Mediated Passthrough”. Last but not least, it is possible to offload the dataplane emulation from the hypervisor and implement it either in the host kernel (eg. vhost + LIO) or in a separate host user space process (eg. SPDK vhost target).

The aforementioned operations incur additional software latency in the I/O datapath. This software latency comes from the following operations:

- I/O request submission: the guest driver informs the device about new available requests via a device doorbell. This implies MMIO/PIO emulation. The emulation overhead depends on the dataplane implementation. In case of the

in-kernel vhost target, we get a VMEXIT, the execution flow goes to kvm and kvm kicks the eventfd that has been hooked up to the doorbell's address. This is how the host kernel vhost subsystem gets notified and starts processing the new guest I/O requests. This is often called a "lightweight VMEXIT". In case of the QEMU user space target, we get a VMEXIT plus a context switch (*KVM_EXIT*) to host user space. Then, QEMU takes control. This is often called a "heavyweight VMEXIT", because it involves both a world switch (VMEXIT) and a context switch for returning to host user space.

- I/O submission to storage backend: in case QEMU implements the dataplane, QEMU uses the host kernel system call API to submit I/O requests to the underlying storage backend. This triggers context switches.
- I/O completion: the storage backend sends an interrupt to signal the I/O completion. Each interrupt causes a context switch and the execution of an in-kernel interrupt handler. The SPDK vhost target avoids this overhead because it works with polling.

Except for the latency, the storage virtualization approach affects the scalability as well. In case of QEMU for example, the hypervisor handles all I/O requests from within the same I/O thread. Regardless of the number of guest vCPUs submitting I/O requests to the emulated storage device, QEMU uses one and only thread to handle all these requests. So, it goes without saying that this is a scalability bottleneck for QEMU.

7.3 SPDK/VVU Virtualization overhead

SPDK/VVU has improved virtualization overhead compared to the existing storage virtualization solutions. In detail, its benefits are the following:

- **guest notifications**

  SPDK suppresses the guest notifications because it relies entirely on polling. The SPDK vhost target continuously polls on the vhost virtqueues and checks for any available descriptors. Therefore, there is no need for guest notifications from the master guest driver. SPDK disables guest notifications in order to avoid unnecessary VMEXITs in the master side. This is done with the following function:

```
int
rte_vhost_enable_guest_notification(int vid, uint16_t queue_id, int
enable)
{
        struct virtio_net *dev = get_device(vid);

        if (dev == NULL)
                return -1;

        if (enable) {
                RTE_LOG(ERR, VHOST_CONFIG,
                        "guest notification isn't supported.\n");
                return -1;
        }

        dev->virtqueue[queue_id]->used->flags =
        ↪  VRING_USED_F_NO_NOTIFY;
        return 0;
}
```

The virtio specification provides a specific flag for disabling/enabling guest notifications for virtio devices, which is called *VRING_USER_F_NO_NOTIFY*. This flag concerns a specific virtio ring and is set in the flags field of the used vring structure (definition of used vring is in */usr/include/virtio_ring.h*). However, according to the virtio spec, this feature it's unreliable, so it's simply an optimization.

The SPDK vhost target has access to the vhost virtqueues, which are part of the master guest physical memory, through the MMIO BAR 2 of the virtio-vhost-user device. This BAR is RAM-backed, hence touching it does not trigger VMEXITs in the slave side. In other words, virtqueue polling does not incur any virtualization overhead in the slave side. The only additional virtualization overhead related to virtqueue polling comes from the memory virtualization mechanism[80]. Each memory access on the RAM-backed BAR incurs an address translation in the MMU. The overhead comes from the TLB misses and depends on whether we are using shadow page tables or extended page tables.

Let's have a closer look at this.

Shadow page tables[81][82] is a memory virtualization technique that is purely software-based. This means that the MMU cannot distinguish between root and non-root mode of operation. So, in case of non-root mode, the CR3 register (this is the register that always points at the head of the page table that is currently being used) points to a page table which keeps GVA→HPA translations. This page table is called "Shadow page table". The overhead for each memory address translation is the following:

- page table hit: same overhead with root mode
- page table miss: VMEXIT, page table updated by KVM
- page table updates by the guest kernel: VMEXIT, KVM replaces the newly inserted page table entries (GVA→GPA) with GVA-to-HPA translations

On the other side, Extended page tables[78][83] (or Nested page tables) is a memory virtualization technique that is hardware-based. This means that the MMU differentiates its behavior in case of non-root mode. In specific, it keeps track of two page tables: the guest page table with GVA→GPA translations, and the KVM page table with GPA→HPA translations. The overhead for each memory address translation is the following:

- page table hit: double overhead in comparison with root mode because two page tables need to be traversed
- page table miss: page fault handled by the guest kernel, no need to rely on KVM
- page table updates by the guest kernel: no additional cost

In conclusion, extended page tables with hugepage-backed memory seems like the best approach in order to reduce the memory virtualization overhead in our use case. However, note that this is a general conclusion that applies to any VM setup. Hugepages imply less TLB misses, because each TLB entry corresponds to a bigger portion of contiguous physical memory, and also hugepages are pinned, hence having fixed physical addresses.[84]

- **device interrupts**

The SPDK vhost target implements virtio devices. Hence, it supports sending interrupts to the master guest driver for the I/O completions. The actual need for device interrupts depends on the type of the master guest driver, that is whether it is interrupt-driven or poll-driven.

If the master guest uses the in-kernel virtio driver to control the vhost-user device, then there is a need for interrupting the guest for each I/O completion. The vhost-user protocol defines that this is done via an eventfd called "callfd". In SPDK, the actual implementation for accessing the callfd is transport-specific. In SPDK/VVU, that is in case of the virtio-vhost-user transport, the vhost target kicks a doorbell of the virtio-vhost-user device that corresponds to the virtqueue it wants to notify. The code path is the following:

```
1 spdk_vhost_vq_used_signal() ⟶
2 rte_vhost_vring_call() ⟶
3 vhost_vring_call_split() ⟶
4 trans_ops->vring_call()
```

The doorbell is an MMIO address that has been hooked up to the callfd via the KVM ioeventfd mechanism. So, when the vhost target kicks the doorbell, we get a VMEXIT, KVM kicks the callfd and re-enters the guest. In the master side, the same callfd has been hooked up to an MSI vector with the irqfd mechanism. So, when the slave KVM kicks the callfd, the master KVM injects a virtual interrupt to the master VM in the next interrupt window. In conclusion, device interrupts triggered by the slave guest involve just a lightweight VMEXIT due to the combination of the ioeventfd and irqfd mechanisms.

On the other side, we could completely alleviate the need for device interrupts by replacing the interrupt-driven virtio device driver with a poll-driven virtio driver. The obvious solution is to use SPDK in the master VM as well. This is straightforward since SPDK already has virtio-scsi and virtio-blk poll-mode drivers. These poll-mode drivers suppress the device interrupts by using the *VRING_AVAIL_F_NO_INTERRUPT* flag. With this solution, we can completely bypass the hypervisor (QEMU/KVM) in the datapath. We can also have a full end-to-end user space storage stack, if we use a passed-through storage backend.

Finally, even when the SPDK vhost target needs to send interrupts to the master guest driver, it does not invoke a callfd kick per I/O completion. SPDK uses interrupt coalescing based on time intervals. This is an optimization that does not affect the guest I/O submission throughput, but it reduces the overall interrupt loads.

- **device emulation**

  First of all, the SPDK/VVU storage virtualization solution completely bypasses the slave guest kernel storage stack. SPDK implements an efficient user space storage stack instead. If this is accompanied with a passed-through storage backend - this means that SPDK in slave guest user space has direct access to the physical device without the intervention of the hypervisor - then SPDK/VVU carries all the advantages of SPDK in comparison with the kernel storage stack. For more on this, refer to chapter 3.

  SPDK/VVU can be set up with various types of storage backends. SPDK supports many types of storage backends. For example, it supports NVMe devices, NVMe-oF devices, virtio-scsi devices, Linux AIO block devices, malloc RAM-disks, etc. However, considering the fact that SPDK/VVU runs inside a VM, these block devices are either emulated or physical passed-through devices. So, alongside the supported storage backends in SPDK, there is also a wide range of alternatives for the emulation of those storage backends. For example, we could have emulated NVMe PCIe disks, virtio-scsi devices, virtio-blk devices, vhost-scsi devices, vhost-blk devices, passed-through physical devices, etc. In most cases, SPDK uses **zero-copy DMA** in the vhost-user I/O datapath. This means that the storage backend performs DMA directly from the master guest memory. Consequently, eliminating the data copies in the datapath reduces the total virtualization overhead.

  Apart from zero-copy in slave side - that means no data copies in SPDK hugepage memory - the vhost-user protocol allows having zero copy in the master side as well. This means that the master guest driver does not have to move the data around in guest memory so that the vhost device backend can access them. The vhost-user protocol specification defines that the master exposes the whole guest physical memory to the slave. Therefore, the slave has access to all guest mem-

ory, thereby being able to access the I/O buffers wherever they are allocated in the first place, thus eliminating the need for data copies in guest memory. In detail, the I/O buffers could either be allocated in guest user space memory (in case a process performs I/O with *O_DIRECT*) or in the guest kernel page cache. In any case, the guest device driver inserts new descriptors in the virtqueue that point to virtio-scsi request structures, which in turn point to SCSI CDBs and the I/O buffers.

This approach - exposing all master guest physical memory to the slave - is actually a compromise between performance and security. The vhost device backend has access to all guest memory though it would be sufficient to have access only to the vhost virtqueues and the I/O buffers. This would be the best solution in terms of security. However, the problem with this solution would be that the guest driver should have to continuously copy the I/O buffers into a fixed portion of guest kernel memory, which would be shareable with the slave. Making copies of the data implies performance degradation.

What is more, SPDK/VVU suppresses the storage backend's interrupt notifications, because SPDK uses poll-mode drivers that poll on the storage backends for I/O completions. This also contributes in minimizing the total virtualization overhead.

Last but not least, using hugepage-backed memory as the master guest's memory would certainly reduce the address translation overhead, because less TLB and IOTLB entries would be required. Using hugepage-backed memory is necessary in case we want to use legacy passthrough mode for the underlying storage backend. In any other case, it is optional though more efficient. Tmpfs-backed memory could be used as well.

- **scalability**

  SPDK/VVU can be deployed with various storage backends underneath, either emulated or passed-through. In the former case, the hypervisor continues to be a scalability bottleneck due to the I/O thread. The reason is that, in case of an SMP guest, where we have a guest that is trying to submit I/O in parallel, even if we have a separate I/O thread implementing a device's dataplane (refer to virtio-blk/virtio-scsi dataplane), there is always one and only host thread handling the

traffic coming from all the device's request virtqueues. However, in the latter case, the hypervisor gets completely out of the picture, and the I/O datapath can scale out either by adding vCPUs or by adding more disks. Here, scaling means increasing the throughput or the IOPS by dedicating more CPUs to I/O, or adding more disks to the system.

- **efficiency**

  It is a fact that SPDK, and consequently SPDK/VVU, is more efficient than the kernel blk-mq/scsi-mq in terms of CPUs per IOPS. This has a dual meaning. The first meaning is that with the same number of CPUs dedicated to I/O, where each CPU is delegated with the exclusive management of a separate I/O queue, SPDK achieves higher IOPS than the kernel. The other meaning is that the kernel needs more CPUs than SPDK in order to achieve the same number of IOPS with SPDK. Thus, SPDK/VVU is a storage virtualization solution that allows for bigger **VM density** by basically saving CPUs for other compute workloads.

7.4 Security

It has already been mentioned in the introductory chapter that security is a great issue in cloud environments, where multiple users are running their workloads on the same physical machines. Originally, prior to the insertion of the virtio-vhost-user transport, SPDK used to support running the SPDK vhost target on host user space alongside the QEMU process. This implies having new potential vulnerability zones.

On the contrary, SPDK/VVU eliminates this problem. The SPDK vhost target gets containerized by running inside a dedicated Storage Appliance VM instead of host user space. So, given the fact that we already trust the hypervisor's code, SPDK/VVU is as must secure as the hypervisor is.

7.5 User-defined Storage

Maybe the most important feature of SPDK/VVU in comparison with the other common storage virtualization solutions is what we call "User-defined Storage". This ba-

sically means that the end user can configure and customize the virtualized storage infrastructure for his VMs. Here is how it works:

we assume that a user runs some I/O workloads in separate VMs in the cloud. Except for these Compute VMs, the user also owns a Storage Appliance VM, which runs locally alongside the Compute VMs. The Storage Appliance VM has some storage devices attached and some virtio-vhost-user devices to communicate with the Compute VMs. Given all the above, the user can run the SPDK vhost target inside the Storage Appliance VM and create virtio-scsi disks, which he can later export to his Compute VMs. Ultimately, what we have is a user that controls the virtual storage devices that his Compute VMs access.

User-defined storage should not be confused with the term "Software-defined Storage". Software-defined Storage refers to the higher-level block services (snapshoting, thin provisioning, etc.) offered by software running on top of commodity hardware. A typical example is Ceph[85].

The range of combinations and tuning parameters of SPDK/VVU are endless. The user can choose from a wide range of storage backends (malloc RAM-disk, local NVMe disk, NVMe-oF, iSCSI, Linux AIO, virtio-scsi, virtio-blk), block layer utilities (GPT, logical volumes, RAID, compression, encryption, snapshoting, thin provisioning) and level of parallelism (I/O queues per poller). These are all inherent capabilities of SPDK. So, based on the workload, the user can choose the right combination to match the needs of the workload. For example, streaming applications need bandwidth more than latency. This means that the spinning disks are a good choice for them. On the contrary, databases rely on low latency, thus matching perfectly with NVMe devices. Some other workloads need some scratch space to operate on. This is where a malloc RAM-disk could fit in.

A schematic representation of a general use case would look like this:

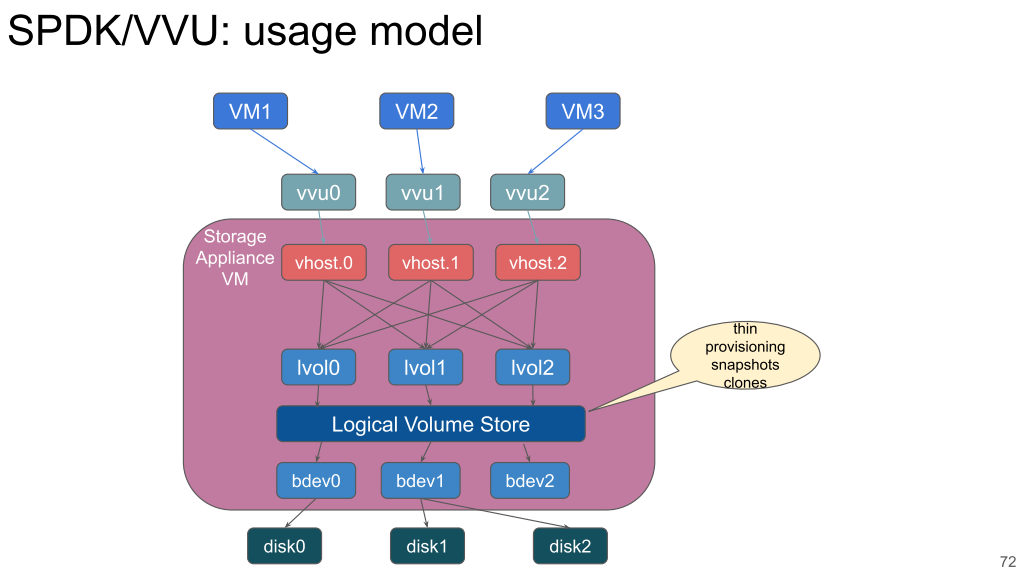


Figure 7.1: *SPDK/VVU Usage Model*

In conclusion, SPDK/VVU introduces the concept of User-defined Storage in the cloud. In other words, the management of the storage infrastructure shifts from the cloud provider to the end user. Though someone could doubt about its usefulness, there is no question that this is a breakthrough.

8 Conclusion

In this final chapter, we are going to assess our current state and outline the next steps. We are also going to give a list of suggestions for potential improvements/enhancements on SPDK/VVU.

8.1 Concluding Remarks

As it has been explained in the Design of SPDK/VVU chapter, the design and implementation of SPDK/VVU required working on multiple open source projects and interacting with the corresponding communities. The reason why we decided to externalize our work is that we wanted to educate ourselves by contributing to a real open source project. The interaction with other engineers and, most improrantly, the social difficulties that arrise from this, were very educational.

Over the last months, we have managed to make significant progress towards our end goal, which is pushing SPDK/VVU upstream. We have managed to merge some patches in SPDK [1] and reach to an agreement on some others [2] that will be merged soon after the DPDK patches are merged. We have also pushed a long patchset [3] in DPDK that adds the virtio-vhost-user transport. This patchset has been previously reviewed by Darek Stojaczyk (software engineer at Intel and code maintainer of SPDK). We have already received some review comments on this and we are working towards pushing a second version. As far as the virtio-vhost-user device is concerned, we have pushed

[1] https://review.gerrithub.io/q/owner:+Dragazis+status:merged
[2] https://review.gerrithub.io/q/status:+open+owner:+Dragazis+repo:+spdk/spdk
[3] http://mails.dpdk.org/archives/dev/2019-June/135116.html

the device spec in the virtio-dev mailing list [4]. We have received some comments and we have already submitted revised versions. Last but not least, we have sent an introductory email in qemu-devel mailing list [5] that attempts to reinitiate the discussion on the virtio-vhost-user QEMU device code. Basically, we are asking for comments on our revised version of the code.

Though significant progress has been made, there are still enough things to be done. The review process on the DPDK patchset is estimated to last long, because the patchset is quite extensive. The plan is to incorporate it in the 19.11 release version of DPDK [6]. The patchset for the virtio device spec has not been adequately reviewed, because it depends on another on-going patchset that attempts to introduce a shared memory capability. When that other patchset gets merged, we will send a reminder. Lastly, we have not yet received any comments on our QEMU device code. The reason is that the device spec has to be approved prior to the actual device implementation. We will send a reminder on that too when the spec gets approved. We may also need to make some changes in the code in case we have previously made changes on the spec. The final step in our plan is to merge the remaining patches in SPDK. This is straightforward since the core maintainers of SPDK have already approved these patches.

Overall, this diploma thesis was very educational. We have hit into real bugs [7], we have learned how to write software patches, we have worked with different development workflows (GerritHub, git email), we have performed live demos [8] and, most importantly, we have built some experience on how to properly interact with other developers.

8.2 Future Work

There are several changes/improvements that could potentially be done in SPDK/VVU. They are enumerated in the following sections.

[4] https://lists.oasis-open.org/archives/virtio-dev/201906/msg00036.html
[5] https://lists.gnu.org/archive/html/qemu-devel/2019-04/msg02910.html
[6] http://mails.dpdk.org/archives/dev/2019-June/135337.html
[7] https://bugs.dpdk.org/show_bug.cgi?id=85
[8] https://lists.01.org/pipermail/spdk/2018-December/002808.html

8.2.1 Add CI tests in SPDK test pool

SPDK uses a test pool to automate the continuous integration of the newly submitted patches. So, it would be very useful to add some tests for the SPDK/VVU use case.

8.2.2 Integrate SPDK/VVU with Katacontainers and Kubernetes

Katacontainers[86] is a containerization framework that combines the security of VMs with the speed of containers. Essentially, it runs containers inside VMs. It uses some techniques to minimize the boot time and memory footprint of the VMs, so that they resemble to containers as much as possible. So, it would be great to use the SPDK/VVU in order to expose high-performance mountpoints inside each container's root filesystem. And later, it would be great to add APIs in Kubernetes[87] for the automation of these setups. Note that Katacontainers is already pluggable to Kubernetes as a container runtime.

8.2.3 Enhancements in the virtio-vhost-user code in QEMU

Currently, the virtio-vhost-user device code is still under development. Though my device code is fully functional and compliant with the revised spec, there are several more enhancements that could be made, like cross-endian support, live migration, irqfds, etc.

8.2.4 Implement the virtio-vhost-user device over more transports

Currently, the virtio-vhost-user device has been implemented over the PCI transport. However, we could also implement the device over the other two transports that virtio supports, that is the MMIO and CCW transports.

8.2.5 Implement Filesystems for SPDK

Recall that SPDK implements a full-userspace high-efficient storage stack. However, there is a part of the kernel storage stack that SPDK misses and that is filesystems. This

means that SPDK works exclusively on a block level. However, since SPDK is a storage stack, we think it would make sense to be able to interact directly with other user space processes. For example, a great use case would be to run it inside the master VM in the SPDK/VVU setup. If SPDK could directly interact with I/O intensive processes inside the master VM, then we would have an end-to-end full-userspace storage stack, hence we could completely bypass the Linux kernel.

So, the idea is to implement a user space filesystem sitting on top of the SPDK Block Layer. An application should be able to seamlessly interact with *read()*/*write()* calls and file-based semantics with SPDK.

8.2.6 Refactor the SPDK's memory map structure

Currently, the SPDK memory maps have 2MB granularity. This essentially means that DMA operations can only be done from hugepage memory. However, using hugepage memory is more of an optimization than a prerequisite in most cases. So, it would make sense to extend the memory map so that it can support saving translations for regular 4KB pages. This would also allow for peer-to-peer DMA scenarios.

8.2.7 Rewrite SPDK's API for the vhost-user transport

The current situation is that the transport is detected based on the vhost controller's name, which is either passed through JSON RPC calls or through a static configuration file. A more robust solution would be to insert a *–trtype* option, both in the RPC calls, and in the configuration file, so that the user can explicitly choose between the *AF_UNIX* and the virtio-vhost-user transports.

Bibliography & References

[1] Wikipedia – The Free Encyclopedia, *3D XPoint*, https://en.wikipedia.org/wiki/3D_XPoint [Online; accessed on the July 5th, 2019].

[2] *Virtual I/O Device (VIRTIO) Version 1.1*, OASIS, December, 20th, 2018, https://docs.oasis-open.org/virtio/virtio/v1.1/csprd01/virtio-v1.1-csprd01.html [Online; accessed on the July 3rd, 2019].

[3] *VFIO - "Virtual Function I/O"*, Linux Kernel Documentation, https://www.kernel.org/doc/Documentation/vfio.txt [Online; accessed on the July 3rd, 2019].

[4] Bo Peng, Haozhong Zhang, Jianguo Yao, Yaozu Dong, Yu Xu, Haibing Guan, *MDev-NVMe: A NVMe Storage Virtualization Solution with Mediated Pass-Through*, USENIX ATC '18, https://www.usenix.org/system/files/conference/atc18/atc18-peng.pdf [Online; accessed on the July 5th, 2019].

[5] Stefan Hajnoczi, *QEMU Internals: vhost architecture*, Open source and virtualization blog, September, 7th, 2011, http://blog.vmsplice.net/2011/09/qemu-internals-vhost-architecture.html [Online; accessed on the July 3rd, 2019].

[6] *Vhost-user Protocol*, QEMU Documentation, https://git.qemu.org/?p=qemu.git;a=blob_plain;f=docs/interop/vhost-user.rst;hb=HEAD [Online; accessed on the July 3rd, 2019].

[7] Wikipedia – The Free Encyclopedia, *Memory-mapped I/O*, https://en.wikipedia.org/wiki/Memory-mapped_I/O [Online; accessed on the July 5th, 2019].

[8] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman, *Linux Device Drivers, Third Edition*, O'Reilly Media, Inc., 2005, https://lwn.net/Kernel/LDD3/ [Online; accessed on the July 3rd, 2019].

[9] X Window System Internals, *The role of PCI*, http://xwindow.angelfire.com/page13_1.html [Online; accessed on the July 5th, 2019].

[10] BIOS CENTRAL, *BIOS and Boot Sequences*, http://www.bioscentral.com/misc/biosbasics.htm [Online; accessed on the July 5th, 2019].

[11] Ravi Budruk, Don Anderson, Tom Shanley, *PCI Express System Architecture*, MindShare, Inc., 2003.

[12] Darmawan Salihun, *System Address Map Initialization in x86/x64 Architecture Part 2: PCI Express-Based Systems*, Reverse Engineering, January, 9th, 2014, https://resources.infosecinstitute.com/system-address-map-initialization-x86x64-architecture-part-2-pci-express-based- [Online; accessed on the July 6th, 2019].

[13] Alex Williamson, *VFIO interrupts and how to coax Windows guests to use MSI*, VFIO tips and tricks, September, 22th, 2014, http://vfio.blogspot.com/2014/09/vfio-interrupts-and-how-to-coax-windows.html [Online; accessed on the July 6th, 2019].

[14] Alex Williamson, *IOMMU Groups, inside and out*, VFIO tips and tricks, August, 25th, 2014, http://vfio.blogspot.com/2014/08/iommu-groups-inside-and-out.html [Online; accessed on the July 6th, 2019].

[15] LWN.net, *ATS capability support for Intel IOMMU*, https://lwn.net/Articles/319205/ [Online; accessed on the July 6th, 2019].

[16] *QEMU the FAST! processor emulator*, https://www.qemu.org/ [Online; accessed on the July 3rd, 2019].

[17] Stefan Hajnoczi, *QEMU Internals: Overall architecture and threading model*, Open source and virtualization blog, March, 5th, 2011, `http://blog.vmsplice.net/2011/03/qemu-internals-overall-architecture-and.html` [Online; accessed on the July 3rd, 2019].

[18] *Kernel Virtual Machine*, `https://www.linux-kvm.org/page/Main_Page` [Online; accessed on the July 6th, 2019].

[19] LWN.net, *Ten years of KVM*, `https://lwn.net/Articles/705160/` [Online; accessed on the July 6th, 2019].

[20] *The Definitive KVM (Kernel-based Virtual Machine) API Documentation*, Linux Kernel Documentation, `https://www.kernel.org/doc/Documentation/virtual/kvm/api.txt` [Online; accessed on the July 3rd, 2019].

[21] LWN.net, *Using the KVM API*, `https://lwn.net/Articles/658511/` [Online; accessed on the July 6th, 2019].

[22] Jan Kiszka, *Architecture of the Kernel-based Virtual Machine (KVM)*, `http://www.linux-kongress.org/2010/slides/KVM-Architecture-LK2010.pdf` [Online; accessed on the July 6th, 2019].

[23] Stefan Hajnoczi, *QEMU Internals: Big picture overview*, Open source and virtualization blog, March, 9th, 2011, `http://blog.vmsplice.net/2011/03/qemu-internals-big-picture-overview.html` [Online; accessed on the July 3rd, 2019].

[24] Linux Programmer's Manual, *EVENTFD*, `http://man7.org/linux/man-pages/man2/eventfd.2.html` [Online; accessed on the July 6th, 2019].

[25] Stefan Hajnoczi, *Towards Multi-threaded Device Emulation in QEMU*, KVM Forum, 2014, `https://www.linux-kvm.org/images/a/a7/02x04-MultithreadedDevices.pdf` [Online; accessed on the July 6th, 2019].

[26] Wikipedia – The Free Encyclopedia, *Direct memory access*, `https://en.wikipedia.org/wiki/Direct_memory_access` [Online; accessed on the July 6th, 2019].

[27] SPDK Documentation, *Direct Memory Access (DMA) From User Space*, https://spdk.io/doc/memory.html [Online; accessed on the July 6th, 2019].

[28] SPDK Mailing List, *Questions about vhost memory registration*, https://lists.01.org/pipermail/spdk/2018-November/002647.html [Online; accessed on the July 6th, 2019].

[29] *The Userspace I/O HOWTO*, Linux Kernel Documentation, https://www.kernel.org/doc/html/v4.15/driver-api/uio-howto.html [Online; accessed on the July 6th, 2019].

[30] Alex Williamson, *An Introduction to PCI Device Assignment with VFIO*, The Linux Foundation Events, http://events17.linuxfoundation.org/sites/events/files/slides/An%20Introduction%20to%20PCI%20Device%20Assignment%20with%20VFIO%20-%20Williamson%20-%202016-08-30_0.pdf [Online; accessed on the July 6th, 2019].

[31] LWN.net, *Safe device assignment with VFIO*, https://lwn.net/Articles/474088/ [Online; accessed on the July 6th, 2019].

[32] VFIO Users Mailing List, *Application examples using vfio ?*, https://www.redhat.com/archives/vfio-users/2018-February/msg00013.html [Online; accessed on the July 6th, 2019].

[33] *Dynamic DMA mapping Guide*, Linux Kernel Documentation, https://www.kernel.org/doc/Documentation/DMA-API-HOWTO.txt [Online; accessed on the July 6th, 2019].

[34] Linux Kernel Documentation, *Transparent Hugepage Support*, https://elixir.bootlin.com/linux/latest/source/Documentation/admin-guide/mm/transhuge.rst [Online; accessed on the July 6th, 2019].

[35] Andrea Arcangeli, *Transparent Hugepage Support*, KVM Forum, Boston, August, 9th, 2010, https://elixir.bootlin.com/linux/latest/source/Documentation/admin-guide/mm/transhuge.rst [Online; accessed on the July 6th, 2019].

[36] LWN.net, *Four-level page tables*, https://lwn.net/Articles/106177/ [Online; accessed on the July 6th, 2019].

[37] LWN.net, *Locking and pinning*, https://lwn.net/Articles/600502/ [Online; accessed on the July 6th, 2019].

[38] LWN.net, *The final step for huge-page swapping*, https://lwn.net/Articles/758677/ [Online; accessed on the July 6th, 2019].

[39] VFIO Users Mailing List, *Question about DMA from user space buffers*, https://www.redhat.com/archives/vfio-users/2018-December/msg00000.html [Online; accessed on the July 6th, 2019].

[40] Fedora Project, *Features/KVM Huge Page Backed Memory*, https://fedoraproject.org/wiki/Features/KVM_Huge_Page_Backed_Memory [Online; accessed on the July 6th, 2019].

[41] LWN.net, *Huge pages part 2: Interfaces*, https://lwn.net/Articles/375096/ [Online; accessed on the July 6th, 2019].

[42] M. Jones, *Virtio: An I/O virtualization framework for Linux*, IBM Developer articles, January, 29th, 2010, https://developer.ibm.com/articles/l-virtio/ [Online; accessed on the July 6th, 2019].

[43] Arthur Kiyanovski, *The Real Difference Between Emulation and Paravirtualization of High-Throughput I/O Devices*, Technion - Computer Science Department - M.Sc. Thesis MSC-2017-19 - 2017, http://www.cs.technion.ac.il/users/wwwb/cgi-bin/tr-get.cgi/2017/MSC/MSC-2017-19.pdf [Online; accessed on the July 6th, 2019].

[44] Ing. Vincenzo Maffione, *Virtio networking: A case study of I/O paravirtualization*, http://lettieri.iet.unipi.it/virtualization/2015/io-paravirtualization-tour.pdf [Online; accessed on the July 6th, 2019].

[45] Luigi Rizzo, Giuseppe Lettieri, Vincenzo Maffione, *Speeding Up Packet I/O in Virtual Machines*, http://www.iet.unipi.it/~a007834/papers/20130903-rizzo-ancs.pdf [Online; accessed on the July 6th, 2019].

[46] Wikipedia – The Free Encyclopedia, *SCSI*, https://en.wikipedia.org/wiki/SCSI [Online; accessed on the July 6th, 2019].

[47] Ashish A. Palekar, Robert D. Russell, *DESIGN AND IMPLEMENTATION OF A SCSI TARGET FOR STORAGE AREA NETWORKS*, University of New Hampshire, Department of Computer Science, May, 2001, http://www.cs.unh.edu/~rdr/tr0101.pdf [Online; accessed on the July 6th, 2019].

[48] *SCSI Architecture Model - 3 (SAM-3)*, T10 Technical Committee, http://www.t10.org/ftp/t10/document.02/02-119r0.pdf [Online; accessed on the July 6th, 2019].

[49] Linux Kernel Documentation, *SCSI Interfaces Guide*, https://www.kernel.org/doc/html/v4.13/driver-api/scsi.html [Online; accessed on the July 6th, 2019].

[50] Linux Kernel Documentation, *libATA Developer's Guide*, https://www.kernel.org/doc/html/v4.13/driver-api/libata.html [Online; accessed on the July 6th, 2019].

[51] *NVM Express Base Specification*, Revision 1.4, June, 10th, 2019, https://nvmexpress.org/wp-content/uploads/NVM-Express-1_4-2019.06.10-Ratified.pdf [Online; accessed on the July 6th, 2019].

[52] THOMAS KRENN WIKI, *Linux Multi-Queue Block IO Queueing Mechanism (blk-mq)*, https://www.thomas-krenn.com/en/wiki/Linux_Multi-Queue_Block_IO_Queueing_Mechanism_(blk-mq) [Online; accessed on the July 6th, 2019].

[53] SPDK Documentation, *Submitting I/O to an NVMe Device*, https://spdk.io/doc/nvme_spec.html [Online; accessed on the July 6th, 2019].

[54] Benoît Morgan, Eric Alata, Vincent Nicomette, Mohamed Kaâniche, *Bypassing IOMMU Protection against I/O Attacks*, HAL archives, December, 20th, 2016, https://hal.archives-ouvertes.fr/hal-01419962/document [Online; accessed on the July 6th, 2019].

[55] *Intel Virtualization Technology for Directed I/O*, Intel, Inc., June, 2019, https://software.intel.com/sites/default/files/managed/c5/15/vt-directed-io-spec.pdf [Online; accessed on the July 3rd, 2019].

[56] LWN.net, *Bounce buffer for untrusted devices*, https://lwn.net/Articles/782845/ [Online; accessed on the July 6th, 2019].

[57] QEMU Wiki, *Features/VT-d*, https://wiki.qemu.org/Features/VT-d#Device_Assignment_In_General [Online; accessed on the July 6th, 2019].

[58] Wikipedia – The Free Encyclopedia, *Single-root input/output virtualization*, https://en.wikipedia.org/wiki/Single-root_input/output_virtualization [Online; accessed on the July 6th, 2019].

[59] Ben-Ami Yassour, Muli Ben-Yehuda, Orit Wasserman, *On the DMA Mapping Problem in Direct Device Assignment*, http://www.mulix.org/pubs/iommu/dmamapping.pdf [Online; accessed on the July 6th, 2019].

[60] Yi Liu, *Shared Virtual Memory in KVM*, LinuxCon, China, November, 2017, https://www.lfasiallc.com/wp-content/uploads/2017/11/Shared-Virtual-Memory-in-KVM_Yi-Liu.pdf [Online; accessed on the July 6th, 2019].

[61] TECH DEVIANCY, *Demystifying Unix Domain Sockets*, Thomas Stover, November, 11th, 2011, http://www.techdeviancy.com/uds.html [Online; accessed on the July 6th, 2019].

[62] *Storage Performance Development Kit*, https://spdk.io/ [Online; accessed on the July 6th, 2019].

[63] *Data Plane Development Kit*, https://www.dpdk.org/ [Online; accessed on the July 6th, 2019].

[64] *Introduction to the Storage Performance Development Kit (SPDK)*, https://software.intel.com/en-us/articles/introduction-to-the-storage-performance-development-kit-spdk [Online; accessed on the July 6th, 2019].

[65] Ben Walker, Jim Harris, May, 6th, 2019, *10.39M Storage I/O Per Second From One Thread*, https://spdk.io/news/2019/05/06/nvme/ [Online; accessed on the July 6th, 2019].

[66] SPDK Documentation, *User Space Drivers*, https://spdk.io/doc/userspace.html [Online; accessed on the July 6th, 2019].

[67] SPDK Documentation, *Message Passing and Concurrency*, https://spdk.io/doc/concurrency.html [Online; accessed on the July 6th, 2019].

[68] LWN.net, *Improvements in the block layer*, https://lwn.net/Articles/735275/ [Online; accessed on the July 6th, 2019].

[69] SPDK Documentation, *Event Framework*, https://spdk.io/doc/event.html [Online; accessed on the July 6th, 2019].

[70] Daniel Verkamp, *SPDK: Under the Hood*, https://s3.us-east-2.amazonaws.com/intel-builders/day_1_spdk_under_the_hood.pdf [Online; accessed on the July 6th, 2019].

[71] Linux Piter #4, *SPDK and Nutanix AHV: Minimising the Virtualisation Overhead*, Dr Felipe Franciosi, November, 2018, https://linuxpiter.com/system/attachments/files/000/001/558/original/20181103_-_AHV_and_SPDK.pdf?1543328586 [Online; accessed on the July 6th, 2019].

[72] SPDK Documentation, *Virtualized I/O with Vhost-user*, https://spdk.io/doc/vhost_processing.html [Online; accessed on the July 6th, 2019].

[73] RedHat Enterprise Linux Documentation, *DIRECT I/O*, https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/5/html/global_file_system/s1-manage-direct-io [Online; accessed on the July 6th, 2019].

[74] IBM Knowledge Center, *Considerations for the use of direct I/O (O_DIRECT)*, https://www.ibm.com/support/knowledgecenter/STXKQY_5.0.0/com.ibm.spectrum.scale.v5r00.doc/bl1adm_considerations_direct_io.htm [Online; accessed on the July 6th, 2019].

[75] Linux Kernel Documentation, *Overview of the Linux Virtual File System*, https://www.kernel.org/doc/Documentation/filesystems/vfs.txt [Online; accessed on the July 6th, 2019].

[76] LWN.net, *A block layer introduction part 1: the bio layer*, https://lwn.net/Articles/736534/ [Online; accessed on the July 6th, 2019].

[77] LWN.net, *Block layer introduction part 2: the request layer*, https://lwn.net/Articles/738449/ [Online; accessed on the July 6th, 2019].

[78] *Intel 64 and IA-32 Architectures Software Developer's Manual*, Intel, Inc., Semptember, 2016, https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3c-part-3-manual.pdf [Online; accessed on the July 3rd, 2019].

[79] Robert Love, *Linux Kernel Development, Third Edition*, Addison-Wesley, 2010.

[80] KVM Documentation, *KVM Memory*, https://www.linux-kvm.org/page/Memory [Online; accessed on the July 7th, 2019].

[81] Linux Kernel Documentation, *The x86 kvm shadow mmu*, https://www.kernel.org/doc/Documentation/virtual/kvm/mmu.txt [Online; accessed on the July 7th, 2019].

[82] Johan De Gelas, *Memory Management*, https://www.anandtech.com/show/2480/7 [Online; accessed on the July 7th, 2019].

[83] Johan De Gelas, *The second generation: Intel's EPT and AMD's NPT*, https://www.anandtech.com/show/2480/10 [Online; accessed on the July 7th, 2019].

[84] James E. Smith, Ravi Nair, *Virtual Machines: Versatile Platforms for Systems and Processes*, Morgan Kaufmann, 1 edition, June, 17th, 2005.

[85] *Ceph*, https://ceph.com/ [Online; accessed on the July 7th, 2019].

[86] *Kata containers*, https://katacontainers.io/ [Online; accessed on the July 9th, 2019].

[87] *Kubernetes*, https://kubernetes.io/ [Online; accessed on the July 9th, 2019].

[88] Daniel P. Bovet, Marco Cesati, *Understanding the Linux Kernel, 3rd Edition*, O'Reilly Media, Inc., November, 2005.

[89] SPDK Documentation, *vhost Target*, `https://spdk.io/doc/vhost.html` [Online; accessed on the July 3rd, 2019].