User-level Services for Multitenant Isolation

A Dissertation

submitted to the designated
by the Assembly
of the Department of Computer Science and Engineering
Examination Committee

by

Georgios Kappes

in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

University of Ioannina
School of Engineering
Ioannina 2021

Dedication

To my wonderful family.

Acknowledgements

First of all, I feel very fortunate to have had Prof. Stergios Anastasiadis as my advisor. His passion and enthusiasm for research was truly motivational and inspired me to start my Ph.D. Stergios provided me with invaluable guidance and support, and he was always available for long and creative brainstorming sessions. He taught me how to do research, how to understand, design, and implement systems software, how to set and pursue high standards, and how to pay attention to details. I hope to be able to match his standard in supervising my own students in the future.

I would also like to thank the members of my examination committee, Prof. Stavros Nikolopoulos, Prof. Panayiotis Tsaparas, Prof. Vassilios V. Dimakopoulos, Prof. Alex Delis, Prof. Vasiliki Kalogeraki, and Prof. Nectarios Koziris, for their insightful, encouraging and kind comments.

The last years at the Computer Systems Lab where amazing. I have been very lucky to meet great people, including Fanis Balaskas, Iordanis-Rafael Fydanakis, Panagiotis Bakochristos, Giannis Bakos, Argiris Katsoulieris, and Vaggelis Dimoulis, that made the office hours more exciting (especially Giannis with his singing talent!). Our Friday lunch breaks where always full of souvlaki and tsipouro.

My wonderful friends where always ready to make each off-work moment special. I am grateful to Sophia Papacharalampous for our amazing trips, and our endless talking sessions, Lefteris Alexiou for the countless hours of training, and Akis Syntakas for making my military service enjoyable.

I also own a special thanks to Rodopi Kosteli for putting color to my quarantine days and making them fun with her enthusiasm, smile, and humor. Our countless hours of running and cycling, together with our wonderful road trips, helped me rest, recharge, and motivate to accomplish every single goal.

Last but not least, I am grateful to my parents Lefteris and Areti, and my sister Antonia for their love, encouragement, and patience throughout my life.

Table of Contents

List of Figures

List of Tables

List of Algorithms

Abstract

Georgios Kappes, Ph.D., Department of Computer Science and Engineering, University of Ioannina, Greece, 2021.
User-level Services for Multitenant Isolation.
Advisor: Stergios V. Anastasiadis, Associate Professor.

In modern cloud environments, virtualization enables multiple tenants to flexibly share the datacenter manycore machines with high processing and memory capacity. Operating-system containers are a lightweight form of virtualization that is commonly used to run the data-intensive applications of different tenants in cloud infrastructures, because they offer flexible virtualization with low overhead.

At a high level, two main reasons make this sharing challenging. First, the system kernel of a cloud machine serving software containers can become a performance bottleneck and an attack surface for the colocated tenants. The problem is caused by both the high utilization of the hardware resources serving the kernel and the software components of the kernel shared across the tenants in non-scalable ways. To tackle this issue, cloud providers take the extreme approach of running containers inside hardware-level virtual machines, losing the benefits that follow from their native execution. Second, the secure management of the enormous user populations that belong to different tenants in a cloud environment necessitates the design of scalable and tenant-aware access control policies at the infrastructure-level. The cloud infrastructure primarily manages administrative entities, but lacks the fine granularity required to manage individual end users. Furthermore, the known file-based solutions face scalability limitations because they either lack support for multiple tenants, or they support multitenancy through inefficient mechanisms, like global-to-local identity mapping.

The focus of this dissertation is to enable multiple tenants to efficiently and securely share the computing, storage, and network infrastructure of the datacenter. To this end, we take the radical approach of moving the data-intensive I/O services at user level from the shared kernel, in order to serve the containers of competing tenants over the same cloud machines. Our contributions consist of innovative methods to handle POSIX-like system calls at user level through a library, the producer-consumer transfer of data and requests over shared memory with efficient memory copy and relaxed lock-free queues, the construction of stacked user-level I/O services, and a multitenant access control mechanism built natively into a distributed filesystem.

In the first part of this dissertation, we present the design of the libservices unified user-level abstraction to dynamically provision per tenant container root filesystems, application data filesystems, and image repositories. We outline several examples of container storage systems whose clients and servers can be composed from libservices. Using libservices, we design the Polytropon toolkit, a collection of user-level components configurable to build several types of filesystems. The toolkit provides an application library to invoke the standard I/O calls, a user-level path to isolate the tenant I/O traffic to private host resources, and user-level filesystem services distinct per tenant.

With the Polytropon toolkit, we build the Danaus client architecture in order to improve the resource sharing at the client hosts running the applications of competing tenants over network storage. Danaus lets each tenant access the container root and application filesystems from network storage through a private host path. We developed a Danaus prototype that integrates per tenant a union filesystem with a Ceph distributed filesystem client and a configurable shared cache. Across different host configurations, workloads, and systems, Danaus achieves improved performance stability, because it handles I/O with the reserved container resources of a tenant and avoids the intensive kernel locking.

In the second part of this dissertation, we focus on the problem of fast communication between software components. This problem becomes particularly important due to the decomposition of monolithic applications into microservices and the relocation of system I/O services from the kernel to the user-level. We examine two key components of the communication mechanism, the queue data structure that is used by producers and consumers for communication and the memory copy operation that is used for data transfer. At the queue side, a typical queueing discipline (e.g., FIFO) removes first the item with the longest time in the queue to minimize waiting and avoid starvation. However, we experimentally confirm that the time-based item ordering is an inherently sequential approach that limits concurrency and delays operations. To tackle this issue, we introduce the Relaxed Concurrent Queues (RCQ) family of queues. The key idea of RCQ is a relaxed ordering model that splits the enqueue and dequeue operations into a stage of sequential assignment to a queue slot and a stage of concurrent execution across the slots. The RCQS algorithm is a provably linearizable lock-free member of the RCQ family. We experimentally show that RCQS achieves factors to orders of magnitude advantage over the state-of-the-art strict or relaxed queue algorithms, across several latency and throughput statistics of the queue operations and item transfers.

At the copy side, existing systems already include multiple memcpy versions to choose from, for instance, according to the features supported by the processor. However, they lack adaptation at a finer grain. We present the Asterope algorithm that automatically produces optimized versions of a memory copy function according to the characteristics of the system processor. We experimentally demonstrate that the memory copy functions produced by Asterope track or even improve the performance of the fastest copy routine on systems with different processors.

In the third part of this dissertation, we focus on the problem of multitenant access control and we present the Dike authorization architecture that combines native access control with tenant namespace isolation and compatibility to object-based filesystems. We introduce secure protocols to authenticate the participating entities and authorize the data access over the network. We alternatively use a local cluster and a public cloud to experimentally evaluate a Dike prototype implementation that we developed and report a limited performance overhead at several thousand tenants.

This dissertation shows that it is possible to allow the data-intensive applications of different tenants to share the same datacenter machines with reduced contention for shared resources. The systems that we propose enable the cloud operation with lower cost, more predictable I/O latency and throughput, increased resilience to attacks from colocated tenants, increased performance for the end users under contention conditions, improved energy efficiency overall as a result of the better resource utilization achieved.

Εκτεταμενη Περιληψη

Γεώργιος Καππές, Δ.Δ., Τμήμα Μηχανικών Η/Υ και Πληροφορικής, Πανεπιστήμιο Ιωαννίνων, 2021.
Υπηρεσίες Επιπέδου Χρήστη για Πολυμισθωτική Απομόνωση.
Επιβλέπων: Στέργιος Β. Αναστασιάδης, Αναπληρωτής Καθηγητής.

Στα σύγχρονα περιβάλλοντα υπολογιστικού νέφους, η εικονικοποίηση επιτρέπει σε πολλούς μισθωτές μιας υπηρεσίας νέφους να διαμοιράζονται με ευελιξία πολυπύρηνους υπολογιστικούς κόμβους με υψηλή επεξεργαστική ισχύ και χωρητικότητα μνήμης. Οι εικονικές μηχανές επιπέδου λειτουργικού συστήματος, γνωστές ως περιέκτες, είναι μια μορφή εικονικοποίησης που χρησιμοποιείται συνήθως για την εκτέλεση εφαρμογών με υψηλές επεξεργαστικές και αποθηκευτικές απαιτήσεις, επειδή προσφέρουν ευέλικτη εικονικοποίηση με χαμηλή επιβάρυνση.

Σε υψηλό επίπεδο, δύο κύριοι λόγοι καθιστούν δύσκολο το διαμοιρασμό κοινόχρηστων υπολογιστικών πόρων από πολλαπλούς μισθωτές μιας υπηρεσίας νέφους. Πρώτον, ο πυρήνας του λειτουργικού συστήματος ενός υπολογιστικού κόμβου που εξυπηρετεί πολλαπλούς περιέκτες, μπορεί να μετατραπεί σε σημείο συμφόρησης και επίθεσης. Αυτό το πρόβλημα προκαλείται τόσο από την αυξημένη χρήση των πόρων υλικού που εξυπηρετούν τον πυρήνα, αλλά και από τον μη κλιμακώσιμο διαμοιρασμό στοιχείων λογισμικού του πυρήνα σε πολλαπλούς μισθωτές. Συνήθως, οι πάροχοι υπηρεσιών υπολογιστικού νέφους για να αντιμετωπίσουν αυτό το ζήτημα καταφεύγουν σε ακραίες λύσεις, όπως η εκτέλεση των εικονικών μηχανών επιπέδου λειτουργικού συστήματος πάνω από παραδοσιακές εικονικές μηχανές επιπέδου υλικού. Ωστόσο, με αυτό τον τρόπο χάνουν τα οφέλη που πηγάζουν από την εκτέλεση των περιεκτών απευθείας στο λειτουργικό σύστημα της φυσικής μηχανής. Δεύτερον, η ασφαλής διαχείριση των τεράστιων πληθυσμών χρηστών που ανήκουν σε διαφορετικούς μισθωτές μιας υπηρεσίας υπολογιστικού νέφους απαιτεί το σχεδιασμό κλιμακώσιμων πολιτικών ελέγχου πρόσβασης στο επίπεδο της υποδομής. Η

υποδομή ενός υπολογιστικού νέφους διαχειρίζεται κυρίως διαχειριστικές οντότητες υψηλού επιπέδου, όπως οι μισθωτές, αλλά δε μπορεί να διαχειριστεί με κλιμακώσιμες μεθόδους τους τελικούς χρήστες του κάθε μισθωτή. Προηγούμενες λύσεις που διαχειρίζονται πολλαπλούς χρήστες στο επίπεδο του συστήματος αρχείων αντιμετωπίζουν προβλήματα κλιμακωσιμότητας, είτε γιατί δεν υποστηρίζουν χρήστες πολλαπλών μισθωτών, είτε γιατί καταφεύγουν σε μη αποτελεσματικές μεθόδους, όπως η χρήση ενδιάμεσων επιπέδων μετάφρασης, για την υποστήριξη πολλών μισθωτών.

Ο στόχος της παρούσας διατριβής είναι να επιτρέψει τον αποτελεσματικό και ασφαλή διαμοιρασμό των συστημάτων υποδομής επεξεργασίας, αποθήκευσης, και δικτύου μιας υπηρεσίας υπολογιστικού νέφους μεταξύ πολλαπλών μισθωτών. Για την επίτευξη αυτού του στόχου μετακινούμε τις υπηρεσίες Ε/Ε από τον κοινόχρηστο πυρήνα ενός συστήματος υποδομής που εξυπηρετεί περιέκτες ανταγωνιστικών μισθωτών στο επίπεδο χρήστη. Οι συνεισφορές της παρούσας διατριβής αποτελούνται από: α) Τη δυναμική δημιουργία σύνθετων συστημάτων αποθήκευσης επιπέδου χρήστη για περιέκτες. β) Καινοτόμες μεθόδους για το χειρισμό κλήσεων συστήματος τύπου POSIX στο επίπεδο χρήστη μέσω βιβλιοθήκης. γ) Το σχεδιασμό ενός μηχανισμού διαδιεργασιακής επικοινωνίας επιπέδου χρήστη για τη γρήγορη και αποδοτική μεταφορά αιτήσεων και δεδομένων μεταξύ διαφορετικών διεργασιών που ακολουθούν το μοντέλο παραγωγού-καταναλωτή μέσω κοινόχρηστης μνήμης. Για το σκοπό αυτό σχεδιάζουμε και υλοποιούμε αποδοτικές μεθόδους αντιγραφής δεδομένων καθώς και δομές δεδομένων τύπου ουράς που υποστηρίζουν παραλληλισμό χωρίς κλειδώματα και χαλαρώνουν τις αυστηρές απαιτήσεις διάταξης. δ) Τη δημιουργία πολυεπίπεδων υπηρεσιών Ε/Ε επιπέδου χρήστη. ε) Και τέλος τη σχεδίαση ενός πολυμισθωτικού μηχανισμού ελέγχου πρόσβασης στο επίπεδο του συστήματος αρχείων.

Στο πρώτο μέρος της διατριβής παρουσιάζουμε μια ενοποιημένη αρχιτεκτονική για τη δυναμική δημιουργία υπηρεσιών αποθήκευσης επιπέδου χρήστη για περιέκτες. Η αρχιτεκτονική που προτείνουμε μπορεί να εφαρμοστεί τόσο για τον πελάτη, όσο και για τον εξυπηρετητή ενός κατανεμημένου συστήματος αποθήκευσης. Ένα βασικό στοιχείο της αρχιτεκτονικής είναι μια μια αφαιρετική βιβλιοθήκη λογισμικού επιπέδου χρήστη που την ονομάζουμε libservices. Κάθε μισθωτής μπορεί να δημιουργήσει ένα σύνθετο σύστημα αποθήκευσης επιπέδου χρήστη με βάση τις ανάγκες των εφαρμογών του, συνδυάζοντας πολλαπλές βιβλιοθήκες libservices σε μια στοίβα, όπου κάθε βιβλιοθήκη υλοποιεί μια επιμέρους υπηρεσία του συστήματος αποθήκευσης. Παραδείγματα συστημάτων αποθήκευσης που μπορούν να χτιστούν με χρήση της βιβλιοθήκης libservices αποτελούν τα συστήματα αποθήκευσης για τα συστήματα αρχείων ρίζας των περιεκτών, τα συστήματα αποθήκευσης για τα δεδομένα των εφαρμογών, και τα συστήματα αποθήκευσης για τα αποθετήρια των ειδώλων των περιεκτών.

Βασιζόμαστε στην αρχιτεκτονική libservices έτσι ώστε να σχεδιάσουμε και να υλοποιήσουμε την εργαλειοθήκη Polytropon, μια συλλογή στοιχείων επιπέδου χρήστη που μπορούν να χρησιμοποιηθούν για την κατασκευή διαφόρων συστημάτων αρχείων που εκτελούνται στο επίπεδο χρήστη ως εφαρμογές. Πιο συγκεκριμένα, η εργαλειοθήκη αποτελείται από μια βιβλιοθήκη που διασυνδέεται με τις εφαρμογές χρήστη και παρέχει υποστήριξη για τις τυπικές κλήσεις Ε/Ε, ένα μηχανισμό διαδιεργασιακής επικοινωνίας επιπέδου χρήστη που μπορεί να χρησιμοποιηθεί για την απομόνωση της μεταφοράς των αιτήσεων και των δεδομένων μεταξύ των εφαρμογών και των συστημάτων αρχείων του κάθε μισθωτή, και τέλος υπηρεσίες συστήματος αρχείων επιπέδου χρήστη που μπορούν να εκτελεστούν ξεχωριστά για κάθε μισθωτή, αξιοποιώντας μόνο τους ιδιωτικούς πόρους που του έχουν ανατεθεί.

Χρησιμοποιώντας την εργαλειοθήκη Polytropon σχεδιάζουμε και υλοποιούμε μια νέα αρχιτεκτονική για τον πελάτη ενός κατανεμημένου συστήματος αρχείων για περιέκτες, την οποία ονομάζουμε Danaus. Ο στόχος μας είναι να βελτιώσουμε τη χρήση των κοινόχρηστων πόρων στα συστήματα που εκτελούνται οι εφαρμογές πολλαπλών μισθωτών. Η αρχιτεκτονική πελάτη που προτείνουμε επιτρέπει στους περιέκτες κάθε μισθωτή να αποκτούν πρόσβαση στα δεδομένα του συστήματος αρχείων ρίζας και εφαρμογών που βρίσκονται αποθηκευμένα σε ένα δικτυακό σύστημα αρχείων μέσω ενός ιδιωτικού μονοπατιού. Υλοποιούμε ένα πρωτότυπο της προτεινόμενης αρχιτεκτονικής, το οποίο διασυνδέει σε επίπεδο χρήστη, ξεχωριστά για κάθε μισθωτή, ένα σύστημα αρχείων ένωσης με τον πελάτη του κατανεμημένου συστήματος αρχείων Ceph. Το σύστημα αρχείων που δημιουργούμε περιλαμβάνει κρυφή μνήμη αποθήκευσης και μπορεί να διαμοιραστεί με μια ομάδα από περιέκτες που ανήκουν σε έναν μισθωτή. Με αναλυτικά πειράματα δείχνουμε ότι η λύση μας βελτιώνει σημαντικά την απομόνωση και επιτυγχάνει σταθερή απόδοση, επειδή χρησιμοποιεί τους ιδιωτικούς πόρους του κάθε μισθωτή για να χειριστεί τις αιτήσεις Ε/Ε, αποφεύγοντας διάφορα σημεία συμφόρησης που εμφανίζονται στον πυρήνα του λειτουργικού συστήματος.

Στο δεύτερο μέρος της διατριβής επικεντρωνόμαστε στο πρόβλημα της γρήγορης επικοινωνίας μεταξύ διαφορετικών στοιχείων λογισμικού που εκτελούνται σε επίπεδο χρήστη. Το πρόβλημα αυτό καθίσταται ιδιαίτερα σημαντικό λόγω της τάσης που επικρατεί για την αποσύνθεση μονολιθικών εφαρμογών σε μικροϋπηρεσίες και τη μεταφορά των υπηρεσιών Ε/Ε από τον πυρήνα στο επίπεδο χρήστη. Μελετούμε δύο βασικά στοιχεία του μηχανισμού επικοινωνίας, τη δομή δεδομένων ουράς που χρησιμοποιείται για την επικοινωνία των αιτήσεων μεταξύ των παραγωγών και των καταναλωτών και τη λειτουργία αντιγραφής περιεχομένων μνήμης που χρησιμοποιείται για τη μεταφορά δεδομένων.

Στην πλευρά της ουράς, μια τυπική αυστηρή διάταξη εξάγει πρώτα το στοιχείο με το μεγαλύτερο χρόνο αναμονής στην ουρά, προσπαθώντας να μειώσει το χρόνο αναμονής και να αποφύγει περιπτώσεις στέρησης εξυπηρέτησης. Ωστόσο, τα πειράματά μας αποδεικνύουν ότι η αυστηρή διάταξη των στοιχείων με βάση το χρόνο εισαγωγής τους στην ουρά είναι μια εγγενώς ακολουθιακή προσέγγιση, η οποία μειώνει τον παραλληλισμό και τελικά καθυστερεί την εξαγωγή τους. Για να αντιμετωπίσουμε αυτό το πρόβλημα παρουσιάζουμε μια νέα οικογένεια ουρών που την ονομάζουμε Relaxed Concurrent Queues (RCQ). Η βασική ιδέα της οικογένειας RCQ είναι ένα χαλαρό μοντέλο διάταξης που διαχωρίζει τις λειτουργίες εισαγωγής και εξαγωγής σε δύο στάδια. Το πρώτο στάδιο αναθέτει σειριακά κάθε λειτουργία σε μια θέση της ουράς και το δεύτερο επιτρέπει την ολοκλήρωση μιας λειτουργίας παράλληλα με άλλες λειτουργίες που έχουν ανατεθεί σε διαφορετικές θέσεις της ουράς. Ο αλγόριθμος RCQS είναι ένας αποδεδειγμένα γραμμικοποιήσιμος αλγόριθμος χωρίς κλειδώματα της οικογένειας RCQ. Με αναλυτικά πειράματα δείχνουμε ότι ο RCQS, συγκριτικά με προηγούμενους αλγορίθμους που χρησιμοποιούν είτε αυστηρή, είτε χαλαρή διάταξη, επιτυγχάνει κατά τάξεις μεγέθους χαμηλότερη καθυστέρηση στις λειτουργίες της ουράς και συνολικά αυξημένη απόδοση κατά τη μεταφορά των αιτήσεων από τους παραγωγούς στους καταναλωτές.

Στην πλευρά της αντιγραφής περιεχομένων μνήμης, τα υπάρχοντα συστήματα περιλαμβάνουν μια σειρά από διαφορετικές εκδόσεις της ρουτίνας αντιγραφής memcpy, όπου κάθε μια είναι προσαρμοσμένη στα βασικά χαρακτηριστικά που υποστηρίζει ένας συγκεκριμένος επεξεργαστής. Ωστόσο, δε διαθέτουν μεθόδους που να επιτρέπουν στη ρουτίνα αντιγραφής να προσαρμοστεί αυτόματα στα λεπτομερή χαρακτηριστικά ενός επεξεργαστή. Στα πλαίσια της διατριβής παρουσιάζουμε τον αλγόριθμο Asterope που παράγει αυτόματα βελτιστοποιημένες εκδόσεις της ρουτίνας αντιγραφής σύμφωνα με τα λεπτομερή χαρακτηριστικά ενός συγκεκριμένου

επεξεργαστή. Αποδεικνύουμε πειραματικά ότι με τη χρήση των ρουτινών αντιγραφής που παράγει ο αλγόριθμος Asterope στον πελάτη ενός κατανεμημένου συστήματος αρχείων, μπορούμε να προσεγγίσουμε ή να βελτιώσουμε την απόδοση της πιο γρήγορης μεθόδου αντιγραφής μνήμης σε συστήματα με διαφορετικούς επεξεργαστές.

Στο τρίτο μέρος της διατριβής επικεντρωνόμαστε στο πρόβλημα του πολυμισθωτικού ελέγχου πρόσβασης σε περιβάλλοντα υπολογιστικού νέφους. Παρουσιάζουμε μια πολυμισθωτική αρχιτεκτονική ελέγχου πρόσβασης που την ονομάζουμε Dike, η οποία συνδυάζει τον εγγενή έλεγχο πρόσβασης με την απομόνωση του χώρου ονομάτων του κάθε μισθωτή. Η αρχιτεκτονική που προτείνουμε είναι συμβατή με κατανεμημένα συστήματα αρχείων που βασίζονται σε αποθήκευση αντικειμένων. Παρουσιάζουμε ασφαλή πρωτόκολλα για την ταυτοποίηση των οντοτήτων του συστήματος και για την εξουσιοδότηση πρόσβασης στα δεδομένα των μισθωτών. Με αναλυτικά πειράματα σε μια τοπική συστοιχία υπολογιστών και σε μια δημόσια πλατφόρμα υπολογιστικού νέφους δείχνουμε ότι η λύση μας εισάγει χαμηλή επιβάρυνση κατά την υποστήριξη χιλιάδων μισθωτών.

Με την παρούσα διατριβή καθιστούμε εφικτή την εκτέλεση των εφαρμογών διαφορετικών μισθωτών μιας πλατφόρμας υπολογιστικού νέφους σε κοινόχρηστους υπολογιστικούς κόμβους αποφεύγοντας τη συμφόρηση κατά τη χρήση κοινόχρηστων πόρων. Τα συστήματα που προτείνουμε και υλοποιούμε επιτρέπουν στις υπηρεσίες υπολογιστικής νέφους να προσφέρουν μειωμένο κόστος ενοικίασης, προβλέψιμη απόδοση και καθυστερήσεις στις λειτουργίες εισόδου/εξόδου, υψηλότερη ανθεκτικότητα σε σφάλματα και επιθέσεις μεταξύ μισθωτών που μοιράζονται κοινοχρήστους πόρους, αυξημένη απόδοση στους τελικούς χρήστες σε συνθήκες συμφόρησης, βελτιωμένη ενεργειακή απόδοση, ως αποτέλεσμα της καλύτερης χρήσης των πόρων των συστημάτων υποδομής.

Chapter 1

Introduction

Big data is getting bigger at a rapid pace, as the Internet of Things connects more and more devices that generate data, such as computers, smartphones, sensors, home appliances, and cars. In addition, data becomes increasingly important, as a huge amount of extremely useful information is hidden in it. As a result, data intensive applications that manage [1, 2, 3, 4], search [5, 6, 7, 8], analyze, and process data [9, 10, 11, 12], in order to extract valuable information from it, are becoming more and more popular.

The cloud is a natural place to store, process, and analyze big data, for improved efficiency, scale, redundancy, availability, flexibility, and security [13, 14, 15, 16, 17, 18, 19, 20, 21, 22]. Cloud computing frees its consumers from purchasing and managing their own hardware resources, by enabling remote access to software services and hardware infrastructure, such as compute, memory, networking, and storage. The infrastructure of a cloud platform typically consists of redundant data centers distributed around the globe. A datacenter incorporates compute nodes for application execution, and storage nodes for data persistence.

The need for speed and efficiency in data storage and access is crucial for data intensive applications to achieve the required performance objectives. To optimize data storage and I/O access in the cloud, the systems community has proposed a plethora of high performance distributed storage systems for data storage [23, 24, 1, 2], caching [25, 26, 27], and deduplication [28, 29, 30]. Such systems are incorporated into the storage nodes for data persistence, and into the compute nodes for data access.

Virtualization is a key technology that is widely used in cloud datacenters to facilitate server consolidation [31, 32, 33]. Resource usage trends in the cloud lead the design of more flexible resource rental models. Such models, allow the dynamic rental and release of fine-grained resources [20] to support heterogeneous workloads with uncertain demand [31]. Therefore, a common use case is to deploy diverse workloads in separate isolated virtual machines in order to improve resource utilization.

Unfortunately, despite its prevalence, cloud computing faces significant scalability and security challenges, which make the handling of competing tenants painful. First, to achieve high resource utilization, cloud providers use virtualization methods to share a datacenter host across multiple tenants. However, the high utilization of the host resources and system-level software components that are shared across the tenants introduce hotspots of contention. Especially in the public clouds, the problem of resource contention is somewhat unpredictable and often undermines the performance and resilience of the cloud-based services. Second, each tenant of the cloud platform consists of a large number of end-users that need access to the cloud services. While the cloud platforms manage administrative entities at the tenant level, they cannot handle the individual end-users in scalable ways.

In the rest of this chapter, we discus in detail the reasons behind the difficulty of supporting competing tenants in modern cloud infrastructures and the contributions of this thesis towards improving datacenter multitenancy.

1.1 Multitenant I/O Management

Software containers are a popular virtualization technique with low operation and management overhead. They achieve efficiency by executing the applications directly on the host operating system, rather than a guest operating system over a hypervisor that hardware virtualization does. Although the containers mostly ran ephemeral and stateless microservices in their early days, today they casually serve data-intensive applications of different tenants over persistent storage in the cloud [34, 35, 36, 37, 38, 39].

Lightweight virtualization is fundamental to reduce the virtualization overheads and achieve high efficiency in cloud datacenters. The increased efficiency cuts operational costs, by enabling higher consolidation density and power reduction, and enables important use-cases, such as just-in-time deployment of applications or cloud functions, a key feature of serverless computing [40].

However, with the prevalence of operating-system-level virtualization through containers, the system kernel of a machine is contended among the colocated tenants. As a result, the hardware resources along with the software modules used by the shared kernel can become hot spots and bottlenecks in the application execution. The problem of resource contention in a cloud environment is somewhat unpredictable and often undermines the performance and resilience of the cloud-based services.

Along the I/O path of an I/O operation from a data-intensive application, the control and data operations face contention at multiple layers of the shared network and storage. A process invokes the host kernel to access the storage servers backing the network filesystem (e.g., AWS EFS) or block volume (e.g., AWS EBS) [41]. The host kernel handles the I/O calls through the local page cache and storage software stack. The storage servers serve multiple clients with the shared page cache, kernel modules, and storage device bandwidth of their local filesystems. The application hosts and storage servers compete for the available bandwidth and buffer space of the network cards and datacenter switch ports. The network performance isolation has been previously addressed by centrally controlled resource allocators, flow endpoint rate limiters and congestion signaling protocols [42, 43, 44].

Although the dynamic resource allocation, kernel service replication and hardware virtualization are actively explored as potential solutions, the multitenant I/O handling through the operating-system kernel remains challenging. Indeed, the system kernel offers a convenient abstraction of the storage software stack, but it also introduces several limitations that make the execution of data-intensive applications problematic:

1. The kernel services consume resources (e.g., memory space, cpu time) that are inaccurately accounted to the colocated processes due to the complex multiplexing or sharing, thus opening up fairness issues [45, 46].

2. The kernel services consume resources that often exceed the reservations of the tenants that cause the respective I/O activities (e.g., cpu time for page flushing).

3. There are consumable resources (e.g., software locks) that remain unaccounted for due to their allocation complexity (e.g., arbitration of synchronization instructions) whose mitigation requires complex restructuring of the application or system code [47, 48].

4. The kernel involvement incurs implicit hardware costs (e.g., mode switch, cache pollution, TLB flushes) that penalize unfairly the collocated tenants regardless of the relative intensity of their I/O activity [49].

5. The monolithic kernel complicates the customized configuration of the system parameters (e.g., page flushing) needed by each tenant [50, 51].

6. The implementation of new filesystems to meet the container storage needs follows the traditional kernel software development that is time-consuming [52, 53, 38].

7. Finally, the shared storage I/O path makes the colocated tenants more vulnerable to attacks or bugs that discourage the wider container adoption [54, 55, 56].

1.1.1 Contributions

We have designed and implemented several systems that allow the tenants of a cloud platform to run data-intensive applications on the same machines with reduced contention for shared resources. The systems that we have designed will enable the cloud operation with lower cost, predictable I/O latency and throughput, better resilience to attacks from collocated tenants, better performance for the end users under contention conditions, and improved power efficiency overall as a result of the higher hardware utilization achieved. Below we provide an overview of each system.

Libservices: Dynamic Storage Provisioning for Multitenant I/O Isolation Libservices is a unified user-level storage abstraction to dynamically provision per tenant container root filesystems, application data filesystems, and image repositories. The goal of libservices is to provide end-to-end performance isolation of the containerized data-intensive applications running on a shared infrastructure by different tenants.

The libservice abstraction is a stackable user-level storage component derived from existing I/O libraries. Based on libservices, we build complex filesystem services that implement the storage client of a host or the local filesystems of the storage servers. We apply the same design pattern at either the client or the server to connect the libservices of a tenant to the application or server container processes through shared-memory interprocess communication [57]. We outline several examples of container storage systems whose clients and servers can be composed from libservices. With an early prototype, we successfully demonstrate that the libservices combine the required efficiency and flexibility to build isolated I/O services on multitenant hosts with superior performance over existing user-level or kernel-level systems.

Polytropon: A User-level Toolkit for Storage I/O Isolation on Multitenant Hosts The Polytropon[1] toolkit is a collection of reusable components that we combine to construct configurable filesystems on multitenant hosts. The *container pool* is a group of containers that a tenant runs on a host. A filesystem is accessed and shared among the processes of a pool with standard semantics (e.g., POSIX). Both the filesystem configuration [51] and the host I/O path of a pool are isolated from the kernel and other pools. The *filesystem service* of a pool is a composition of libservices. At a host, the filesystem service implements the filesystem client of network storage, or a local filesystem. We route the I/O requests to the filesystem services through a *dual interface* whose I/O path runs at user level by default; alternatively, it crosses the kernel (e.g., VFS) for compatibility with legacy software (e.g., static linking).

We implement an efficient user-level interprocess communication (IPC) between the application processes and a filesystem service with a highly concurrent queue, called *Relaxed Concurrent Queue Blocking* (RCQB), and a fast memory copy method, called *Shared Memory Optimized* (SMO). RCQB achieves higher performance than the state-of-the-art queues by 4-52x, while SMO achieves 29-66% higher data transfer throughput than existing methods.

Danaus: Isolation and Efficiency of Container I/O at the Client Side of Network Storage Danaus[2] is a filesystem client architecture that lets each tenant access the

[1]From polytropos, the very first adjective used by Homer in Odyssey to describe the intellectual brilliance, guile, and versatility of Odysseus.

[2]In Greek mythology, Danaus with the advice of Athena built a ship, the first ever existed, and fled with his 50 daughters to Argos, where he became king.

container root and application filesystems from network storage through a private host path. The key idea of Danaus is to *provision a distinct user-level filesystem client per tenant on a host*. We construct each client from a private stack of user-level functionalities based on the abstraction of libservices. A client runs its own code path and data cache privately on the reserved resources of the tenant. We pin the client threads on reserved cores and run them at user level to reduce the mode and context switches. Furthermore, the cloned or collaborating containers take advantage of configurable filesystem sharing to avoid the duplication of memory and storage resources.

We build a Danaus prototype from two libservices running a union filesystem and a distributed filesystem client. We experimentally demonstrate that a mature kernel-based client and union filesystem lead to low performance due to lock contention inside the kernel and I/O handling with resources exceeding those reserved. In contrast, Danaus achieves higher performance combined with low memory and cpu utilization, especially in write-intensive or scaleout workloads, because it handles I/O with the reserved container resources of a tenant and avoids the intensive kernel locking. Although Danaus is slower in cached read, we identified a potential solution for performance improvement through a concurrency optimization of the user-level distributed filesystem client. Overall, Danaus serves 32 tenants with RocksDB over network storage with 7.2-14x lower latency than kernel systems, and reduces up to 2.9x the timespan to execute source-code processing in the containers of 32 tenants. It also offers up to 14.4x higher throughput than a popular kernel-based client under conditions of I/O contention, and in comparison to a FUSE-based user-level client reduces by 14.2x the time to start 256 high-performance webservers.

1.2 Fast Interprocess Communication

The fast communication within the same or across different protection domains has been a long-standing challenge in systems research. Today, it remains mostly relevant due to the high processing capacity of the manycore machines, the low-latency devices attached to them, and the decomposition of monolithic applications into microservices.

The microservice architectural style is becoming increasingly popular [58]. This style proposes the decomposition of monolithic applications into smaller, independent components which can communicate through well defined interfaces. It provides several benefits, such as faster development cycles, improved fault isolation, and increased scalability. Due to its distributed nature, one of the key challenges when designing applications is the interprocess communication (IPC) mechanism by which services communicate with each other.

Typically, IPC involves a producer and a consumer process which need to communicate with each other. The producer-consumer problem commonly appears in a multicore machine when a server process receives requests from application threads and returns back responses. The producers communicate with the consumers through a shared data structure often implemented as a concurrent FIFO queue. The concurrency allows multiple producer and consumer threads to simultaneously access the queue, while the FIFO property guarantees that the consumer removes from the queue the producer request that has waited the longest. The correctness of the concurrent queue is typically reasoned through the linearizability property that requires each queue operation to appear as completed at a single time point between the operation invocation and response [59].

The design and implementation of a practical concurrent queue is challenging because it involves complex tradeoffs between conflicting goals and the abstract reasoning about their properties. The enqueue and dequeue operations should be fast for low latency. A typical queueing discipline (e.g., FIFO) removes first the item with the longest time in the queue to minimize waiting and avoid starvation. However, the time-based item ordering is an inherently sequential approach that limits concurrency and delays operations. In prior research, the comparative benefit of the FIFO fairness over the resulting sequentiality delay remains unclear.

In addition, IPC typically involves a lot of data copying or transferring between the communicating components. When moving data between software components, data copying reduces the need for mutual coordination and increases the flexibility for independent buffer management, but with the downside of the utilized cpu cycles [60]. In cross-domain communication, the transmitted data is typically written by the sender and read by other layers without further change [61]. Depending on the need for data reuse by the sender, the buffers can be transferred with move or copy semantics implemented with page remapping or statically shared memory, respectively. Both approaches reduce the memory copies, but data is still copied between the private and remapped or shared memory.

Existing systems spend substantial I/O time on data copying between the user and kernel memory [62, 63, 48]. The applied POSIX semantics is often mentioned as main reason for suboptimal performance [64, 65]. Faster system interfaces take advantage of asynchronous mechanisms [66, 67] that rely on shared memory between the application and the kernel. Alternatively, user-level message passing is used between writer and reader cores through cache-coherent shared memory [68]. Other mechanisms take advantage of hardware virtualization features [69] or userspace memory protection keys [70] for the fast communication between processes with kernel bypass, or between different kernel subsystems [71]. Additionally, low-latency network [72, 73, 74, 75] and storage devices [76, 77, 63] are directly accessed over shared buffers through application-linked libraries.

Despite all these advances, the memory copy (memcpy) between private and shared buffers remains in the critical path of the application-kernel interaction, interprocess communication, and device access. It is not surprising that the memcpy alone is reported responsible for processor cycle percentage as high as 38% in production cloud benchmarks [78]. The related measurements show high sensitivity to the hardware configuration, the system settings, and the mix of colocated workloads. Thus, the memory access performance has been constantly optimized with prefetching [79, 80, 81], reduced cache pollution [78, 82], non-temporal [83], and other [84, 85] instructions.

Existing systems already include multiple memcpy versions to choose from, e.g., according to the SIMD features supported by the processor [86], but they lack adaptation at a finer grain.

1.2.1 Contributions

We target two of the most critical and fundamental building blocks of an IPC mechanism, the queue data structure that is used for request communication and the memory copy method that is used for data copying. In order to achieve fast communication at high concurrency we design and implement a family of relaxed concurrent queues that split the queue operation into two stages, the sequential assignment of operations to queue slots and their subsequent concurrent execution. Furthermore, in order to optimize the memory copy operation, we introduce a cross-platform framework to experimentally generate optimal memcpy parameters for different system architectures. Below we provide an overview of these algorithms.

RCQs: A Family of Relaxed Concurrent Queues for Low-latency Operations and Item Transfers The Relaxed Concurrent Queues (RCQs) is a family of Relaxed Concurrent Queues that combine fast enqueue and dequeue operations with short item waiting in the queue. We implement a single queue with a preallocated fixed-size array to avoid the overhead of dynamic memory management during the queue operations. We relax the FIFO ordering by splitting each of the enqueue and dequeue operations into two stages. The first stage assigns the operations in sequential order to the queue slots and the second stage completes the operations in order that depends on the thread scheduling and the memory access arbitration. We experimentally evaluate several blocking and lock-free RCQ algorithms that are provably linearizable and deadlock-free under the assumed relaxed ordering. We do extensive performance comparison of RCQ with existing queues and demonstrate that the lock-free RCQ algorithms achieve lower operation and wait latency, reduced standard deviation, and faster item transfer through the queue by several factors to orders of magnitude.

Asterope: A Cross-Platform Optimization Method for Fast Memory Copy Asterope[3] is a cross-platform framework to automatically identify the optimal software and hardware memcpy settings. The memcpy performance depends on multiple hardware components with features that vary across different architectures, vendors, and models. Our key idea is simply to identify the maximum performance per transfer size by applying an exhaustive search over the load, store and register type, and the prefetch size, distance and type. We focus on the x86 architecture, which is popular and supports a wide range of instructions, register sizes and prefetch parameters.

Over two x86-64 models, we experimentally demonstrate 1.5x improved performance at the client of the Ceph distributed filesystem by the mere drop-in replacement of the Glibc memcpy with ours for the cache-to-application transfer.

1.3 Multitenant Access Control

Cloud infrastructures are increasingly used for a broad range of computational needs in private and public organizations, or personal environments. In the datacenter,

[3]In Greek mythology, Asterope whose name means "starry face" is one of the Okeanides (the 3,000 daughters of Okeanos & Tethys).

aggressive systems consolidation is opening up new opportunities for flexible data sharing among the users of one or multiple tenants (e.g., constellations of co-resident applications [87]). For commercial, analytics and social applications, data exchanges at low cost are a key benefit that valuably complements the reduced operational expenses already offered by the cloud. As the cloud landscape is dominated by concerns about the security and backward compatibility of systems software, it remains open problem how to achieve scalable file sharing across different virtualized machines or personal devices serving the same or different tenants.

Existing solutions of cloud storage primarily provide centralized management of entire virtual disks over a common backend (e.g., Ceph RDB, OpenStack Cinder). Although they conveniently provision the capacity elasticity of disk images, they do not facilitate native support of scalable data sharing at file granularity. Similarly, identity management in the cloud selectively grants coarse-grain root authorization to administrative domains, but lacks the fine granularity required to specify the permissions of individual end users. Furthermore, the known file-based solutions face scalability limitations because they either lack support for multiple tenants, rely on global-to-local identity mapping to support multitenancy, or have the guests and a centralized filesystem (or proxy thereof) sharing the same physical host [88, 89, 90].

Multitenant access control of shared files should securely isolate the storage access paths of different users in a configurable manner. Authentication and authorization have already been extensively studied in the context of distributed systems [91, 92, 93]. However, a cloud environment introduces unique characteristics that warrant reconsideration of the assumptions and solution properties according to the following requirements:

1. Each tenant should be able to specify the user identity namespace unrestricted from other tenants. The simple application of identity mapping at large scale adds excessive overhead.

2. The enforcement of authentication and authorization functions should be scalably split across the provider, the tenant and the client. Directory services designed for private environments do not easily support tenants or handle enormous user populations.

3. Unless it is natively supported for flexible and efficient storage access, file sharing requires complex management schemes that are restrictive, error prone and

costly.

4. Access control should take advantage of secure hardware (as root of trust) in machines administered by the provider and the tenant, or in personal user devices. Unless the virtualization execution is adequately trustworthy, the tenants will simply refrain from data sharing.

1.3.1 Contributions

We consider the security requirements of scalable filesystems used by virtualization environments and we introduce a system design to natively support multitenant access control. Below we provide a brief description of this system.

Dike: Multitenant Access Control for Cloud-Aware Distributed Filesystems Dike[4] is a multitenant access control architecture for outsourcing the complex functionality of storage authorization and data sharing to cloud service providers. The proposed design reduces the amount of systems infrastructure maintained by the tenant, and provides an effective interface for storage interoperability among co-located administrative domains. Thus, we enable shared data accesses among numerous users, and facilitate distributed data processing over large cluster installations.

We rely on an object-based, distributed filesystem to scalably store the data and metadata of individual files. A guest directly mounts the shared filesystem without the involvement of a local or network proxy server. Each tenant manages the identities and permissions of its own users independently. By maintaining *separately* the access permissions of each tenant, the filesystem securely isolates the identity namespaces of the tenants but also enables configurable file sharing across different tenants and hosts. We provide the Dike prototype implementation of the above approach over the Ceph distributed filesystem [24]. With microbenchmarks and application-level experiments on a local cluster and a public cloud, we quantitatively demonstrate that our design incurs only limited performance overhead. At several thousand tenants, our prototype incurs limited performance overhead below 21%, unlike a solution from industry whose multitenancy overhead approaches 84% in some cases.

[4]Dike was the goddess of justice and moral order in Greek mythology.

1.4 Outline

The remainder of this dissertation is structured as follows. Chapter 2 describes the environment that we study in this dissertation and provides relevant background material. Chapter 3 motivates the problem of multitenant I/O management and describes the solution that this dissertation proposes. Chapter 4 introduces the libservices unified user-level storage abstraction for the dynamic provisioning of per-tenant container storage systems. Chapter 5 presents the Polytropon toolkit, a collection of user-level components configurable to build several types of filesystems that both run and are accessible at user-level. Chapter 6 presents Danaus, a user-level client architecture that provides isolation and efficiency for the container root and application filesystems. In Chapter 7 we examine user-level IPC and motivate the need to design a highly parallel queue data structure and to automatically optimize the memory copy operation according to the low-level characteristics of the underlying system architecture. Chapter 8 presents the Relaxed Concurrent Queues (RCQ) family of queues that relax queue operation ordering in order to achieve fast communication at high concurrency. Chapter 9 introduces the Asterope algorithm that experimentally generates optimal memory copy algorithms for different processors. In Chapter 10 we focus on multitenant access control and motivate the need to natively support it at the filesystem-level. Chapter 11 presents the Dike system for native multitenant access control at the filesystem. Chapter 12 presents an extended review of literature and systems that are related to this dissertations. Chapter 13 presents interesting directions in which this work could be extended in the future. Finally, Chapter 14 summarizes the lessons learned and concludes this dissertation.

Chapter 2

The Cloud Environment

Cloud computing is changing the way we deploy and consume software by relocating the software services, computation, and data storage to centralized and location-transparent cloud platforms. An increasing number of enterprises are adopting cloud computing and move their workloads to cloud platforms in order to leverage their expertise in deploying, managing, and improving common services. Examples of such services include virtual machines [94], distributed storage [95], caching [96], databases [97], and application development environments [98].

In this chapter, we present useful background material to familiarize the reader with the cloud environment that we study in this dissertation. We first define multitenancy (§ 2.1). Then, we focus on the organization of a cloud platform (§ 2.2), and we delve into its key software layers, namely the operating system (§ 2.3), the virtualization (§ 2.4), the storage (§ 2.6), and the security (§ 2.7). We discuss a comprehensive presentation of related work in Chapter 12.

2.1 Multitenancy

Serving multiple customers from a single pool of shared resources is known as *multitenancy*. Multitenancy constitutes the key to achieve economies of scale, as the physical resources become transparent to consumers. It also enables scalability and elasticity, as resources can be added or removed on demand to deal with load variations. Additionally, it offers flexibility and automation, as the cloud platform automates the deployment of a service and the customers do not have to setup the necessary infrastructure [13].

Next, we discuss four system properties that we regard as essential requirements for a multitenant system, namely isolation, efficiency, flexibility, and compatibility.

2.1.1 Isolation

Isolation is a foundational concept of multitasking computer systems and refers to the ability of a system to ensure that a system component has a private view of resources and cannot view, affect or use the resources of other system components without permission. Isolation can be categorized into three areas: security, fault, and performance. Security isolation ensures that if a system component is compromised, the security of other system components is not affected. Fault isolation ensures that a fault in a system component cannot affect the correct operation of other system components. Performance isolation ensures that a system component uses a private set of system resources and its execution cannot generate performance interference to the execution of other system components [99, 100, 101, 102, 103, 104, 105, 106, 107, 45].

2.1.2 Efficiency

Efficiency refers to the ability of a system to perform a task fast with the minimum amount of resources. In fact, efficiency links performance and resource usage together. Efficiency is also strongly tied with scalability. Scalability is the capability of a system to handle an increased or expanding workload. If efficiency decreases significantly as the workload grows, then the system cannot provide scalability. A system should provide multitenant support without introducing performance overheads, or wasting resources, such as cpu, memory, bandwidth, and storage space [50, 108, 109, 110, 45, 111].

2.1.3 Flexibility

Flexibility in a cloud platform refers to the ability of a system to adapt to various requirements of its tenants. A cloud platform should allow its tenants to define custom policies regarding the management, handling, and security of their services and data. For instance, a cloud platform should provide support for flexible access control to permit configurable service and data sharing [92, 16, 112, 51].

2.1.4 Compatibility

Compatibility refers to the ability of a system to run common applications without modifications to their source code. To provide compatibility, a system should minimize the introduction of new application interfaces, but instead leverage interfaces (e.g., POSIX) that are widely used by existing applications. Compatibility is especially important when moving existing applications to a cloud platform. A tenant should assess how any interface restrictions of a cloud platform will impact a target application and estimate the cost and time required to modify the application [113, 114].

2.2 Cloud Platforms

Cloud platforms are commonly deployed in a private or public setting. In a *private cloud* deployment, an organization owns, manages, and exclusively uses a cloud platform. In contrast, in a *public cloud* deployment, an organization rents a bundle of resources from a publicly accessible cloud platform that is owned and managed by a separate organization. Other deployments constituting combinations of the above, are also possible. For instance, in a *hybrid cloud* deployment, the cloud platform combines two or more cloud platforms of different deployment settings that remain unique entities [115].

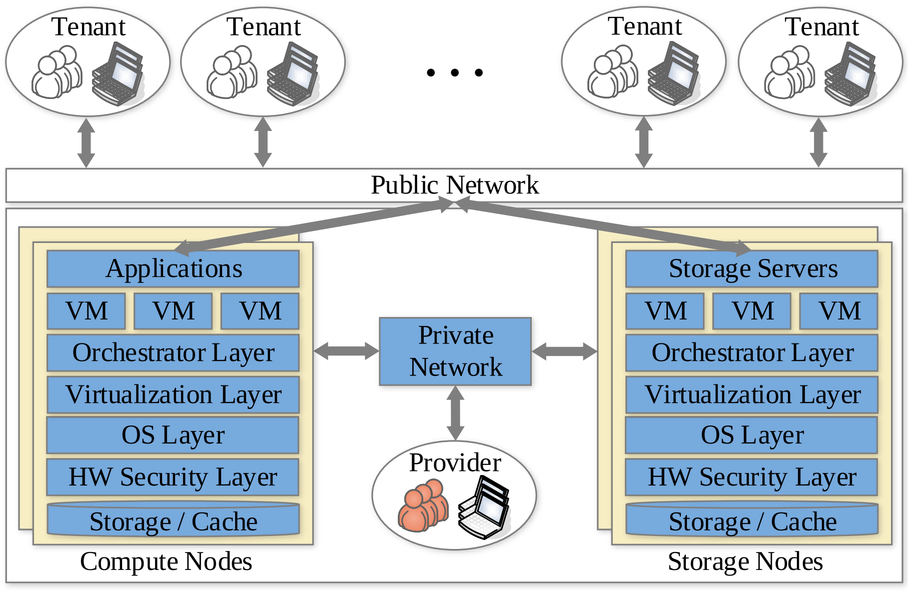


Figure 2.1: **The organization of the studied cloud environment.** This figure depicts the key components that form up the studied cloud environment.

Figure 2.1 depicts the organization of the cloud environment that we study in this dissertation. An independent *cloud provider* owns and operates the infrastructure of a cloud platform. The infrastructure consists of multiple physical machines that incorporate computational, storage, and network resources. The provider typically relies on *virtualization* to multiplex the resources of a physical machine, which we call *host*, over a number of *virtual machines (VMs)*[1]. A VM emulates a physical machine and transparently shares the resources of its host with other VMs.

A collection of *compute nodes* accommodate applications in VMs. In addition, a collection of *storage nodes* store and provide access to persistent application data. Application data may be also cached locally on the compute nodes or the VMs for fast access.

Each *tenant* that holds an account with the cloud provider is typically an independent enterprise constituting of a large number of end users. The tenants rent from the provider a set of storage resources and a collection of VMs to run their applications. A tenant may access the rented storage resources through a VM inside the cloud infrastructure, or through an external node (e.g., a mobile device).

In the rest of this chapter we focus on three key software layers of a cloud infrastructure: (i) the operating-system layer, (ii) the virtualization layer, (iii) the storage

[1]The terms virtual machine, and guest are used interchangeably in the literature.

layer, and (iv) the hardware security layer.

2.3 The Operating-system Layer

The operating system (OS) is the interface between a user and a hardware architecture of a computer system. It consists of the *kernel space* (or *kernel*), and the *user space*. The role of the kernel is to interact with the hardware resources and provide an *execution environment* for running programs. The role of the user space is to provide fundamental application programming interfaces (APIs) and facilities in the form of libraries and utilities, in order to facilitate the development and execution of user application programs [116].

To differentiate the execution of the kernel and user space, the hardware offers at least two different execution modes: the *kernel mode*, which is a privileged mode for the execution of the kernel, and the *user mode*, which is a non-privileged mode for the execution of user space. The kernel provides an interface that consists of *system calls* to user space applications, in order to let them request kernel services. A system call is an instruction that causes two *mode switches*: the execution mode of a program switches from user mode to kernel mode, and a kernel function is executed. When the execution of the kernel function completes, the execution mode is changed back to user mode and the result of the function is made available to the program.

The operating system uses the *process* abstraction to isolate the execution of each running program into a private set of memory addresses known as *address space*. It shares the limited system resources among multiple running processes (e.g., cpu cores) with *scheduling* and provides the illusion of simultaneous execution of multiple processes by switching from one process to the other. The *process switch*, or *context switch*, is a relatively costly operation because it involves changes to the processor state (e.g., cpu registers) and pollution of the processor caches.

2.4 The Virtualization Layer

The multiplexing of physical resources over a number of VMs is known as *consolidation*. Consolidation can help the provider to substantially decrease expenditures on datacenter hardware and operating costs, by running a reduced set of physical machines at higher utilization rates. A subsequent benefit is the reduction of power consumption leading to an environmental-friendlier business [31].

The key technology behind consolidation is virtualization [31, 32, 33]. The concept of virtualization emerged in the mid 1960's when mainframe computer systems dominated the computer industry. At the time, industry professionals and academic researchers found that VMs provided a compelling way to logically partition the scarce resources of these large centralized systems among different applications and users. Each user was given a separate virtual computer which allowed him to work in an isolated environment without affecting other users.

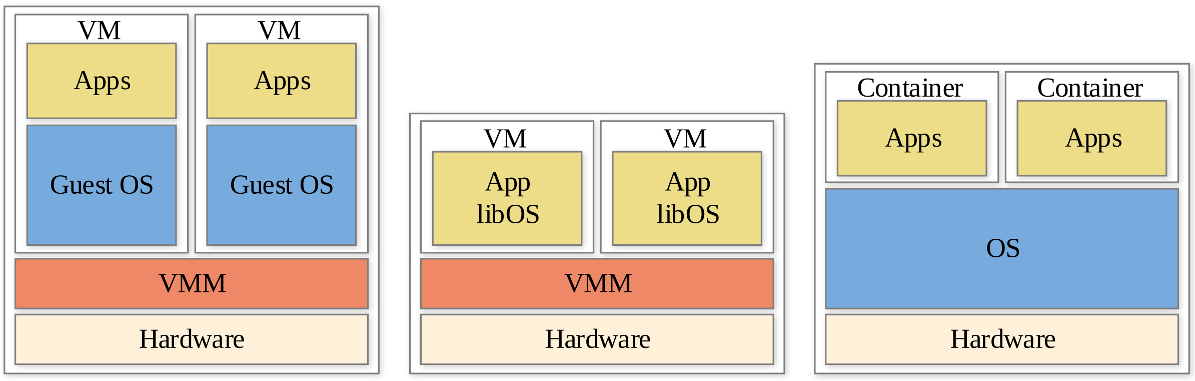
The next decades brought personal computers and minicomputers with modern multitasking operating systems, making virtualization a technology of the past. However, the increasing power of modern systems in combination with the rise of datacenters and cloud computing contributed to its renaissance. Service providers can use virtualization to consolidate various workloads of different tenants on a single physical server maximizing resource utilization.

2.4.1 Hardware-level Virtualization

Virtualization abstracts the operating system and user applications from the underlying hardware resources of a physical host. It provides multiple isolated execution environments for operating system execution in the form of VMs, that share the underlying hardware resources. Traditionally, the operating system controls resource sharing. However, in a virtualized server, a special software layer which is called the *Virtual Machine Monitor (VMM)*[2] is typically placed beneath the hardware and the operating system. Its main objective is to manage system resources and allocate them to one or more VMs [117].

The VMM enables virtualization by solving at software-level various challenges that arise during the virtualization of the processor [118, 119], memory [120, 121, 118, 122, 123, 124, 119], and I/O devices [125, 118, 126, 127, 128, 119, 129]. Nonetheless, virtualization at the software-level is unable to keep up with the performance and security demands of modern datacenters. To this end, hardware manufacturers deploy extensions to their products in order to assist virtualization. Therefore, processors

[2]The terms Virtual Machine Monitor and hypervisor are used interchangeably in the literature.



a) Hardware-level Virtualization with commodity guests
b) Hardware-level Virtualization with libOS guests
c) Operating-system-level Virtualization

Figure 2.2: **Comparison of virtualization approaches.** This figure includes a high-level comparison of different virtualization approaches towards lightweight virtualization. In (a), commodity guests with full operating system installations run on top a VMM. In (b), lightweight guests that consist of a single application and a libOS run on top of a VMM. Lastly, in (c), containers that consist of user-space applications an libraries run on top of a shared operating system.

that provide essential virtualization features to enhance virtualization performance and security [130, 131, 132, 133], as well as I/O devices that perform virtualization in hardware to provide a large number of virtual devices [134], are becoming common.

The VMM provides to a VM the illusion of running on its own dedicated physical machine. A VM typically comprises a stack made up from a commodity operating system (also known as the guest operating system) and user applications. The guest operating system may contain unmodified or virtualization-aware device drivers to access the I/O devices that are made available to the VM. One of the VMs that are co-hosted on a physical server is privileged. It includes a management interface that lets privileged users to manage the unprivileged VMs [118].

2.4.2 Lightweight Virtualization

The extensive use of virtualization in datacenters brings some unique efficiency concerns. While hardware-level virtualization offers a great degree of compatibility to the end user who can deploy his preferred operating system and applications (Figure 2.2a), commodity operating systems used in guests carry numerous device drivers and protocols, which were historically required to support a diverse set of hardware platforms, but which are no longer needed in a modern virtualization environment [135, 136, 137, 138, 139, 140, 63, 73]. Additionally, virtualization raises a number of security issues because it increases the number and size of software layers that comprise a physical server. As a result, each software layer may contain bugs that can be exploited by an attacker to gain unauthorized access to the infrastructure [141, 142, 143, 144, 145, 146, 147, 148, 149].

An approach to reduce the size of the guest operating system is to remove the kernel services that are not needed for the execution of the applications. To assist the operating system tailoring process researchers have explored the manual or automatic creation of call graphs, which represent all the relations among the procedures from the application down to the operating system. Call graphs can be generated automatically by running a test run of the target workload and observing its behavior through system call tracing [150, 151, 152]. However, such approaches can miss functionality that the execution of the observed workload did not triggered (e.g., error handling), and hence still require from the user to create a white-list of features that should be included in the final configuration.

A different approach (Figure 2.2b) moves operating-system functionality to a *library OS* (or *libOS*). The libOS is a user space library that can be linked with the source code of an application into a bootable application binary. The application binary contains the necessary system services (e.g., device drivers, filesystem, network protocols), the modules explicitly referenced in its configuration files, and other applications which must work together (e.g., the database backend of a web server), all linked as libraries into the application. As a result, the operating system kernel is completely removed from the software stack of the VM. This increases efficiency, due to the reduction of data copy operations and mode switches, and security, due to the minimization of the software stack [153, 135, 136, 137, 138, 139, 140, 63, 73]. A drawback is that the application runs as a single process in a single address space. To enable multi-process application support, researchers have proposed an architecture which allows multiple library OSes to cooperate and appear as a single, shared operating system to unmodified legacy applications. However, such cooperation comes with a cost in memory usage and performance, when compared to a traditional kernel [105].

2.4.3 Operating-system-level Virtualization

Researchers have also explored ways to reduce the size of the VMM [143, 144, 146, 147], or to take advantage of virtualization extensions in hardware devices to remove it completely from the device access path [148]. For instance, Nova [144] and CloudVisor [146] introduce separate VMMs for each VM that handle non-critical tasks and decrease the size of the shared VMM that only handles critical tasks. NoHype [148] uses a temporary VMM to statically assign resources to VMs during boot and then removes it during the normal execution of VMs. The VMs utilize virtualization extensions in hardware to access the hardware devices directly, achieving increased performance. However, the removal of the VMM complicates dynamic resource allocation, a key feature of virtualization for achiving high utilization and accommodate heterogeneous workloads [31].

A more drastic approach, which is known as *operating-system-level virtualization*, draws the virtualization line higher in the software stack (Figure 2.2c). According to this approach, the operating-system kernel is modified to support virtualization and accommodate multiple VMs, which are called *containers*, without the need for a VMM. As a result, there is no virtualization for process execution inside a container, as the processes execute natively on the processor, and the shared kernel handles the system calls. In addition, each container consists only of user space libraries and applications, eliminating the duplication of system services, such as device drivers, filesystems, and network protocols [154]. The interface between a container and the shared kernel consists of the kernel supported system calls [154].

Containers can be classified into two types: (a) *System-level containers*, and (b) *application-level containers*. In the former type, each container includes an entire installation of an operating system user-space stack, while in the later type each container includes a single application or service with the minimal dependencies (e.g., libraries) required by that application.

Several studies show that the performance of operating-system-level virtualization exceeds that of hardware-level virtualization, when considering single applications (single tenant) [155, 156, 46, 45]. However, in a multitenant setting where multiple applications are running in containers on the same physical server, the co-located applications may cause high interference [45].

The operating-system kernel should incorporate virtualization mechanisms in order to provide a private resource view (security isolation) [156, 157, 106] and specific resource usage limits (performance isolation) [158, 104, 159] to each container. Ideally, a container process should not be able to escape the container and access, view, or modify resources that belong to other containers. In addition, a container process should not be able to interact with the processes in other containers, disrupt their execution, or cause interference. Specifically, the Linux kernel provides three mechanisms to support containers, Namespaces [160], Control Groups (cgroups) [160], and Capabilities [161].

1. *Namespaces*. Namespaces is a virtualization mechanism of the Linux kernel which virtualizes shared kernel resources and provides an isolated view of a virtualized resource to each container.

2. *Control Groups (cgroups)*. Cgroups is a kernel mechanism that limits, accounts for, and isolates the resource usage (e.g., cpu, memory, I/O, network) of a collection of processes [162].

3. *Capabilities*. They provide fine-grained control over superuser permissions, allowing use of the root user to be avoided. Normally, when the user ID of a process matches the user ID of the root user, the process gets full privileges to act on resources. However, capabilities provide a subset of the root privileges to a process. This effectively breaks up the root privileges into smaller and distinct units, which can then be independently granted to processes.

Namespaces Namespaces split the global resource identifiers and other structures of the Linux kernel into distinct instances. This partitions resources into separate instances in order to provide a unique view of each resource to processes. Namespaces can be combined together into a collection to create a filter across resources for how a collection of processes, or a container, view the system.

The Linux kernel supports namespaces for six global resources as discussed below: filesystem mount points, interprocess communication, host name, process identifiers, user identifiers, and network stacks.

1. **Mount namespace**. Allows processes in different mount namespaces to have different views of the filesystem hierarchy by isolating filesystem mount points.

2. **IPC namespace**. Isolates interprocess communication (IPC) resources, such as System V IPC objects and POSIX message queues.

3. **UTS namespace**. Isolates the host name and the domain name, returned by the *uname* system call.

4. **PID namespace**. Isolates the process identifiers, in order to allow processes in different PID namespaces to have the same identifiers.

5. **User namespace**. Isolates the user and group identifiers. As a result, the user and group IDs of a process can be different inside and outside a user namespace.

6. **Network namespace**. Isolates system resources associated with networking. Each network namespace has its own network stack, network drivers, IP routing tables, port number, IP addresses, and */proc/net* directory.

Cgroups Control groups (cgroups) is a kernel mechanism for applying hardware resource limits to collections of processes and account their resource usage. The cgroups mechanism provides a hierarchical, inheritable, and optionally nested mechanism of resource control that isolates and limits a resource over a collection of processes.

The cgroups mechanism allows for separate *subsystems* or *controllers*, that each one can monitor, account, and isolate the usage of a given resource. These subsystems can control, among others, cpu and memory allocation, block devices, network devices, and device visibility to process groups or containers. There are also subsystems that facilitate checkpoint and resume of processes.

The cgroups interface of the Linux kernel is provided through a virtual filesystem called *cgroupfs*. This filesystem can be used to control process grouping, set and view resource limits, and monitor process resource usage.

The version 1 of cgroups (cgroups v1) organizes processes into multiple hierarchies, and applies separate controllers to each hierarchy to enforce resource policies. Allowing for multiple hierarchies provides high flexibility for process organization. However, it can be expensive for the kernel to keep track of all the controllers that apply to a given process that belongs to multiple hierarchies. Moreover, controllers cannot effectively cooperate with each other, since they typically operate on different hierarchies.

In version 2 of cgroups (cgroups v2) all controllers reside in a single unified hierarchy. Controllers can be enabled or disabled separately for specific sub-trees of the single hierarchy. Processes can be attached only as leaves to each sub-tree of the hierarchy.

Below we provide a short description of the major cgroup controllers of both versions of cgroups that are implemented in the Linux kernel:

1. **cpu (cgroup v1, v2).** This controller controls how the scheduler shares cpu time among different process groups by tracking and enforcing weight and absolute bandwidth limits.

2. **cpuset (cgroup v1).** This controller limits the set of processors and memory nodes on which each process in a group may run and allocate memory.

3. **memory (cgroup v1, v2).** The memory controller controls memory allocation and limits for a group of processes. The following types of memory usages are tracked: i) Page cache and anonymous memory, ii) kernel data structures (e.g., dentries, inodes), and iii) TCP socket buffers.

4. **blkio or io (cgroup v1, v2).** It implements I/O control policies, including I/O bandwidth, throttling, and queue usage.

5. **pid (cgroup v1, v2).** It limits the number of processes or tasks that can be created in a given group.

6. **devices (cgroup v1).** It imposes access control for accesses to device-special files (e.g., block or character devices).

7. **network (cgroup v1).** The network controllers provide a method to classify network packets with tagging (*net_cls*) and to specify priorities for each network interface (*net_prio*).

8. **freezer (cgroup v1).** It allows the system to pause and resume a container by suspending and resuming its processes.

Capabilities Traditional UNIX distinguishes two categories of processes for the purpose of performing permissions checks: *privileged* (whose effective user ID is zero, also referred as root), and *unprivileged* (whose effective user ID is nonzero). Privileged processes have complete control over the system as they bypass all kernel permission checks. In contrast, unprivileged processes are subject to full permission checking based on their credentials. Over the history of Linux, privilege escalation vulnerabilities to root have proved a recurring problem, either due to setuid which changes the effective user ID of a process to uid 0, or due to violating the principle of least privilege by running an application as root in the first place. To solve this issue, Linux divides the privileges traditionally associated with root into distinct units known as capabilities, which can be independently enabled or disabled per process thread.

The Linux kernel maintains three capability sets for each process thread. The *permitted* set is a superset for the effective capabilities that a thread and its children may assume. The *effective* set contains the capabilities utilized by the kernel to evaluate permission checks. Finally, the *inheritable* set contains the capabilities that can be preserved across an exec system call.

2.5 The Orchestrator Layer

Tenant workloads are commonly composed of small, specialized processes, often referred to as microservices, that interact with each other to provide services to end-users. These workloads are deployed on a cluster of nodes, rather than on a single machine. As a result, organizations are increasingly relying on specialized software known as orchestrators to automate the task of deploying, scaling, and managing tenant workloads across a cluster of machines.

Containers are a natural fit for running microservices as they provide a lightweight application deployment unit and a self-contained execution environment. Containers enable software to move between environments. They include the application, and all its dependencies, such as, configuration files, libraries, and other binaries. However, when scalability is required, container orchestration solutions must come into play. Container orchestration software permits container lifecycle management, particularly in large and constantly changing environments, as it automates the deployment, management, scaling, and networking of containers. Specifically, the orchestrator is a software layer that provisions and manages cluster resources, performs application scheduling, and automates a plethora of application-related tasks, such as, configuration, load-balancing, service discovery, cross-node communication, scaling, self-healing, updates and rollbacks, and storage management. Kubernetes, Docker

Swarm, Mesos, and OpenShift are some of the most popular container orchestration systems.

For instance, the Kubernetes orchestration system manages the complexity of container scheduling through consistent APIs, design patterns, and decoupling [110]. A Kubernetes cluster consists of nodes. A node is a physical or virtual machine with a container runtime (or engine). Each node runs an agent process known as kubelet that is responsible for managing the local resources and states of the node. The kubelet relies on the container runtime to allocate local resources to containers and start or stop local containers based on instructions from the control plane. The control plane manages the scheduling and deployment of tenant workloads across nodes and runs on a number of dedicated nodes known as masters. The basic scheduling unit in Kubernetes is a pod. The pod is a group of containers that share resources, such as, network and storage. Kubernetes schedules each pod to execute on one node, but a pod can be replicated to multiple nodes for scalability and fault tolerance purposes. Kubernetes execute tenant workloads inside pods.

2.6 The Storage Layer

Data storage in cloud environments is a challenging problem due to the constantly growing volume, variety, and velocity of data captured by tenants. The volume refers to the quantity of data that reach the backend storage system of a cloud environment. The variety refers to the diversity in data size, type and format. For instance, structured data has a well defined format, and is usually stored in a relational database. On the other hand, unstructured data does not follow a specific format, and is usually stored in a hierarchical filesystem or a non relational database. Finally, the velocity refers to the rate at which the data is arriving to the backend storage.

Tenants use the backend storage system either directly to store unstructured data, or indirectly as the last layer of a multi-tier storage system that manages structured data (e.g., HBase over HDFS [163]). As a result, the backend storage systems used in cloud environments are a critical component for the overall performance and security of the hosted applications. To support the diverse needs of tenant applications, the backend storage systems provide various interfaces that enable data access at block, object, or file granularity.

2.6.1 Block-level Storage

A block-level storage interface exposes a block device to the application and allows the writing and reading of fixed-size blocks. It offers the advantages of strong isolation, compatibility, versioning, and mobility. These advantages have made block-level storage very popular in cloud environments. For instance, VMs can use a block-level interface to access virtual disks which are typically stored in a central location. Virtual disks are offered to VMs as direct attached disks through a block I/O interface, or as volumes through a storage area network mounted by the host [164, 25, 165, 26].

2.6.2 Object-level Storage

An object-level storage interface exposes objects typically through a REST API [166]. Each object typically includes the data itself, a variable amount of metadata, and is uniquely identified with a global identifier. Object-level storage abstracts some of the lower layers of storage, such as the filesystem hierarchy away from the applications making the storage of data simpler. As a result, object-level storage provides better scalability and elasticity than other forms of storage by dropping support for low-level storage features, such as POSIX semantics. It also supports manageability, availability, and reliability [167].

2.6.3 File-level Storage

A file-level storage interface exposes the file and folder structure of a shared filesystem directly to applications. The filesystem is a higher-level logical structure that maps higher-level objects, which are typically called files onto disk blocks. A file-level interface provides the advantage of configurable VM isolation and sharing support at fine granularity. In addition, remote storage access through a file-level interface often improves the performance of VMs in comparison to block-based access, and facilitates semantics awareness, which strengthens the consistency and fault isolation of filesystems [89, 90, 168, 88, 18].

The backend filesystem of a cloud environment can be any distributed filesystem that is designed to support scalability, performance, availability, and reliability. The Ceph filesystem (CephFS) [24], the Google File System (GFS) [23], and the Hadoop Distributed File System (HDFS) [169] are some examples of filesystems widely used in cloud environments.

2.6.4 The Ceph Distributed Filesystem

Ceph is a distributed storage platform whose sophisticated architecture provides high availability, scalability, and reliability. The Ceph distributed filesystem [170] consists of four components: the *clients* provide access to the filesystem, *the metadata servers (MDSs)* manage the namespace hierarchy, the *object-storage devices (OSDs)* store objects reliably, and the *monitors (MONs)* manage the server cluster map [24]. Although data and metadata are managed separately, they are both redundantly stored on OSDs.

The MDSs create a filesystem hierarchy on top of objects and manage file and folder metadata (e.g., access permissions, timestamps). Each MDS caches recently used metadata in local memory for fast subsequent access, but it also stores the metadata reliably in the OSDs. The MDS maintains a journal of recently-updated metadata for failure-recovery.

The OSDs store file data and metadata in object form. They either rely on a kernel filesystem (e.g., XFS [171]) to store the objects, or on a user-level approach that combines a raw device with a key-value store (e.g., BlueStore [172]).

A client provides access to the Ceph filesystem. It interacts with one or more MDSs for filesystem metadata operations. The MDSs provide POSIX compatible metadata to the client, as well as a set of capabilities that describe the operations that the client is allowed to perform. To handle filesystem data operations, the client calculates object placement and replication using the CRUSH algorithm [173] and interacts with one or more OSDs. It provides them with information about the operation to be performed, and with the capabilities obtained from the MDSs.

2.7 The Hardware Security Layer

Trust generally refers to a binary relation between two entities, the source and the target. The source verifies that it is the entity that it claims to be, and the target verifies that the source is really who claims to be, and then believes that the source will act or intend to act beneficially [174]. A core concept of trusted computing is the *root of trust*. It is a set of unconditional trusted functions which are essential to perform tamper-resistance actions. The root of trust can be either hardware-based or software-based.

As the hardware-based root of trust provides greater security properties (e.g., it is physically secured), it is widely used to implement secure environments.

The trusted hardware and software components of a computer system that are critical to its security constitute its Trusted Computing Base (TCB) [175]). Bugs or vulnerabilities occurring inside the TCB might compromise the security properties of the entire system. As a result, the smaller the TCB, the fewer the possible bugs that may contain [176].

One of the basic building blocks of hardware-based root of trust is the Trusted Platform Module (TPM) [177]. The TPM is a secure co-processor that includes a passive I/O device, a set of special registers, and a set of operations for data sealing and unsealing, remote attestation, and other cryptographic services, such as key generation.

A trusted platform typically provides four core security operations. *Protected execution* creates an isolated execution environment for applications that no unauthorized software on the platform can observe or tamper with. *Sealing* provides the ability to encrypt and store sensitive data such as keys within the hardware. This data can be decrypted only in an environment with exactly the same configuration as the one where it was encrypted. *Verified launch* lets the creation of a hardware-based chain of trust that enables the measured launch of the components that form up the software stack. Hence, software components are launched into a known good state. Finally, *remote attestation* provides platform measurement credentials, which can be used by a tenant to verify trust. Examples of widely used hardware-based security mechanisms that provide the above security operations are the Intel TXT [178], the Intel SGX [179], the ARM TrustZone [180], and the AMD Secure Technology [181].

2.8 Summary

A cloud infrastructure should provide support for isolation, efficiency, flexibility, and compatibility, to better serve the needs of a constantly rising number of tenants. A typical cloud environment that we study in this dissertation consists of multiple software layers. Its top layer is the operating system that runs tenant applications. Next, is the virtualization layer that enables resource consolidation, by efficiently and securely sharing the resources of a single computer system among a diverse set of workloads. Hardware-level virtualization relies on the VMM to manage, and allocate system resources to isolated VMs. While this approach provides high compatibility, by allowing its users to incorporate their operating systems of choice, it introduces several performance and security issues. A more efficient alternative to hardware-level virtualization is the operating-system-level virtualization, in which the operating system incorporates virtualization mechanism to run various applications into isolated containers. The orchestrator layer manages containers and cluster resources at large scale. The storage layer caches, persists and manages application data. Storage consolidation can be performed at the block-level, at the object-level, or at the file-level. File-level storage consolidation is increasingly advocated in cloud environments due to its inherent performance, flexibility and sharing. Finally, the security layer incorporates mechanisms for system trust that ensure the integrity and confidentiality of tenant data.

Chapter 3

Multitenant I/O Management

In this chapter, we focus on the storage I/O management of data-intensive applications of different tenants that run in containers on the same cloud infrastructure.

Containers are commonly used to run the data-intensive applications of different tenants in cloud infrastructures, High Performance Computing (HPC) environments, and in resource constrained environments, as they enable resource consolidation at low overhead. The storage I/O of the colocated tenants is typically handled by the shared system kernel of the container host. When a data-intensive container competes with a noisy neighbor, the kernel I/O services can cause performance variability and slowdown. This is a challenging problem for which several approaches have already been explored. Although the dynamic resource allocation, kernel structure replication, and hardware-level virtualization are helpful, they incur costs of high implementation complexity and execution overhead. As a result, containers are still used in combination with more mature virtualization techniques in multitenant settings, such as hardware-level virtualization and rule-based control [110, 45, 182, 183].

As a realistic cost-effective alternative, we propose to move parts of the system kernel to user level and provide separate instances of critical services to each tenant on every machine. Accordingly, we have come up with a unified architecture that allows each tenant to run both the applications and I/O services at user level on private resources (e.g., cores, memory, ports) and avoid contention with colocated entities.

3.1 Background

Storage is a critical part for running stateful containers. In this section we first provide relevant background on container storage management and then we delve deep into the storage stack of the container host.

3.1.1 Container Storage Management

The cloud ecosystem offers several options to support the container storage needs. These include the image repository, the root filesystem and the application data, respectively known as registry storage, storage driver and volume storage in Docker [184, 53]. An *image* is a read-only set of application binaries and system packages organized as a stacked series of file archives (*layers*) [185]. It is made available from the registry running on an independent machine.

During the creation of a new container, a host copies the container image to local storage from the repository, or access it directly from a network storage system (SAN or NAS). The root filesystem and the application data are also kept in local storage, or accessed from network storage [186, 185, 37, 38]. The root filesystem of the container is prepared on a private folder at the host and it is typically combined with an additional writable layer to enable file modifications by the container execution. A storage driver [37] of the container runtime allows sharing image layers across different containers through a union filesystem or snapshot storage. The storage is *ephemeral* if it is deleted at container termination, or *persistent* if it exists independently of the container lifetime. The ephemeral storage lives at the local filesystem of the host, and the persistent storage is served by a network storage system.

A container accesses the application data on a filesystem mounted by the runtime from the host (e.g., bind mount) or over the network (e.g., volume plugins) [37, 38]. A

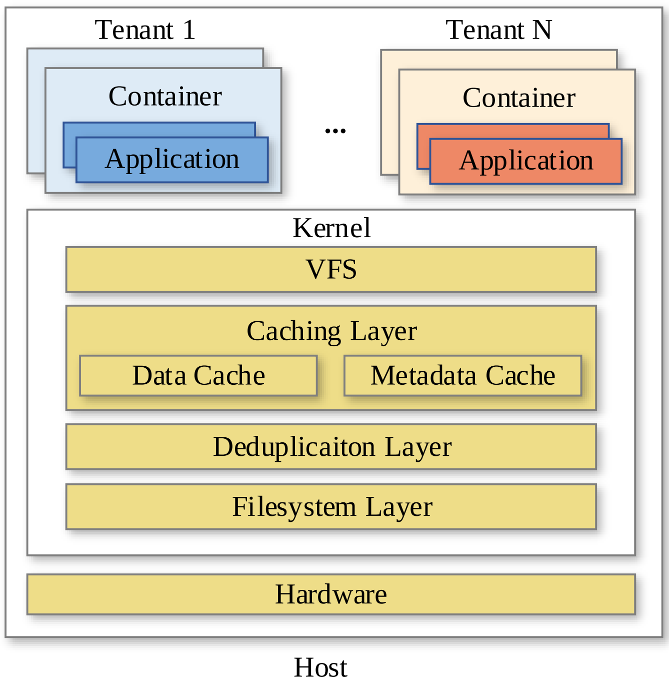


Figure 3.1: **The shared storage stack.** This figure depicts the layers of the legacy storage stack that are shared among the collocated containers.

bind mount accesses a local filesystem, or the client of a distributed filesystem. From a network storage cluster, a volume plugin mounts a distributed filesystem (e.g., MS Azure [187], Amazon EFS [188]), or a block volume (e.g., Amazon EBS [188]) formatted with a local filesystem. The volume plugin is executed by the container runtime *with the necessary support from the kernel* (e.g., NFS kernel module). Finally, a container application may access cloud object storage (e.g., Amazon S3) through a RESTful API.

Subsequently, containers rely on the shared storage stack of the host to access image and application data (Figure 3.1). The Virtual File System (VFS) layer provides a common storage interface to applications. The caching layer caches data in local memory for fast access. The deduplicaiton layer enables efficient storage usage. The filesystem layer communicates with local storage devices or with a remote storage system to store or retrieve data and metadata.

3.1.2 The Virtual File System Layer

In Unix-like operating systems, the highest layer of the storage stack is the VFS. Its role is to provide a common filesystem interface to applications and transparently support various filesystem implementations [189] through a common file model. The common file model consists of four data structures. The superblock defines the structure and characteristics of a filesystem. The inode maintains metadata information (e.g., access permissions, timestamps, block addresses) for a file or a folder. The dentry maps a file or folder path name to an inode. Finally, the file represents opened files and is used by the VFS to support file sharing across different processes.

The VFS uses a series of locks to protect its data structures from concurrent access. Many of these locks were identified as a source of scalability bottleneck by multiple researchers and the Linux kernel developers [190, 191]. To reduce lock contention, they made significant efforts to either eliminate completely the contented locks, or make them more fine-grained [192, 193, 194].

3.1.3 The Caching Layer

The VFS layer commonly caches filesystem data and metadata to hide access latency for persistent storage. It uses a dentry cache and an inode cache to cache filesystem metadata, and a page cache to cache filesystem data. These caches form up the caching layer of the storage stack.

The dentry cache remembers in-memory dentries that map file and folder names to in-memory inodes in order to improve the performance of path name lookups. The dentry cache acts as a controller to the inode cache, which remembers in-memory inodes to speed up access to file or folder metadata. The VFS implements each cache with a system-wide hash table and LRU lists for dentry or inode eviction.

The page cache caches recently accessed data into memory at the granularity of pages. Its goal is to provide reduced latency for I/O operations by exploiting spatial and temporal locality of accesses. Achieving good parallelism and a high hit rate in the page cache are two critical factors for meeting the performance requirements of I/O intensive applications.

To service a read request, the VFS first checks if the requested data is in the page cache. If it is, the VFS returns the cached data. Otherwise, the VFS reads the data from the persistent storage and populates the page cache with the received data by requesting free memory pages from the kernel memory allocator.

To service a write request, the VFS can follow either the write-back, or the write-through strategy. In the write-back strategy, each write operation modifies the data in the page cache, marks each modified page as dirty, and completes. Each dirty page is added to a per-superblock dirty list. Periodically, a collection of kernel threads, known as *flusher threads*, check the dirty lists for dirty pages. The flusher threads write the dirty pages on persistent storage if they were dirty for a threshold interval, if the number of dirty pages has exceeded a threshold, or if there are no free pages to service read requests. A dirty page can be also written to persistent storage explicitly through a system call (e.g., *sync*). In contrast, in the *write-though* strategy, each write operation modifies the data both in the page cache and in the persistent storage before its completion is confirmed.

3.1.4 The Deduplication Layer

Copy-on-write (COW) is a mechanism to efficiently create copies by deferring the copy operation to the first write. When a source resource is copied to a target, the target simply points to the source, as long as the target is not modified. If the target is modified, then a new private resource is created for it, as a copy of the source resource.

Many filesystems support deduplication natively at the granularity of blocks with snapshots. A snapshot is a read-only copy of a part or of the entire filesystem hierarchy. Snapshots are widely used by container platforms to manage container images. They store the layers of an image as separate snapshots. The snapshots that form up the image are unified to form up a single filesystem hierarchy (e.g., using Btrfs subvolumes), or can be flattened to a single snapshot. Different containers that share image layers use a separate snapshot of each common layer.

A different category of filesystems, known as union filesystems, support deduplication at the granularity of files. These filesystems work by unifying multiple folders of the same or separate underlying filesystems known as *branches* into a single unified folder containing a coherent filesystem view. The unified folder is the mount point of the union filesystem. A branch can be accessed in read-only or read-write mode. Typically, branches are stacked. The upper branch is writable, and all the others are read-only. To access a file or folder, the union filesystem traverses the branches from top to bottom. As a result, the files and folders of an upper branch hide the files and folders with common names and paths in lower branches. When a file on a read-only branch is modified, it is copied up to the top writable branch and modified there.

The copy operation also includes the creation of any missing components of the file path on the writable branch. The deletion of a file or folder from a branch is typically implemented with whiteout entries. A whiteout is a hidden entry that covers up all entries of a particular name from lower branches.

A union filesystem can be used by containers to access images with common layers. Each image layer is stored in a separate read-only branch. A writable branch, which is initially empty, is created for each container to capture its filesystem modifications. Each container uses its own union filesystem instance through a private mount point to access its image.

3.1.5 The Filesystem Layer

The last layer of the storage stack is the filesystem layer which provides access to local storage devices or a remote storage system. The legacy storage stack typically uses kernel-level filesystems, which are implemented in the shared kernel. Alternatively, the filesystem may also follow a hybrid design which is split into a kernel-level part, and a user-level part, as well as a user-level design in which the filesystem lies entirely at user-level.

Kernel-level filesystem A kernel-level filesystem is implemented directly in the shared kernel and accessed by user space applications through the VFS. When a filesystem is mounted, it is registered with the VFS. User space applications issue I/O requests to the VFS, that either services them using its local caches, or invokes the respective I/O handlers of the filesystem. The kernel passes data to the application (e.g., during a read operation) by copying the data into an application provided user space buffer.

Hybrid filesystem A hybrid filesystem is divided into two parts: the one part is implemented in the kernel and runs in kernel mode, and the other is implemented at user-level and runs at user mode in a dedicated process. When the filesystem is mounted, its kernel part registers with the VFS. Applications access the filesystem through the VFS, that either services the I/O requests from its caches, or invokes the respective filesystem handlers of the kernel part. The filesystem handlers of the kernel part forward the I/O request to the user-level part, that eventually handles the request. The I/O response follows the same path in reverse [195, 196, 48].

Library-based user-level filesystem with per-process client The filesystem can be also implemented entirely at user level, as a user-level library. The library is linked with an application, which can access the filesystem directly, without trapping to the kernel. The code of the filesystem client and the application execute in the same process at user-level. However, each application process incorporates its private filesystem which complicates sharing functionalities, like caching [197].

Library-based user-level filesystem with centralized client Finally, the filesystem client can be also implemented as a user-level server that executes in a dedicated process. Each application links with a lean library whose job is to forward the application I/O requests to the local server using an interprocess communication (IPC) mechanism. This design enables shared functionality, such as caching, which can be integrated inside the server and shared across multiple applications [198, 199].

3.2 Storage Requirements

In this section we analyze the storage requirements of containers. The application hosts and storage servers require different types of data, filesystems, sharing, caching or deduplication.

3.2.1 Container Storage Systems

A storage system consists of clients and servers, dynamically provisioned (e.g., root, application) or permanently operated (e.g., repository) by the provider. They serve the (i) image repository, (ii) the root filesystems that boot the containers, or (iii) the application filesystems with the application data (Figure 3.2). The image repository stores the container images of the root or application filesystem servers, and the applications. The root filesystem servers of a tenant are launched by having their images copied from the repository. The applications and filesystem servers are launched through clients accessing their images from the root filesystem servers. The application data is accessed through clients from the application filesystem servers. An application runs with the following steps: (i) Identify the hosts of the root or application filesystem servers, and the applications. (ii) Use the container engines to transfer and launch the images of the root filesystem servers. (iii) Store at the root

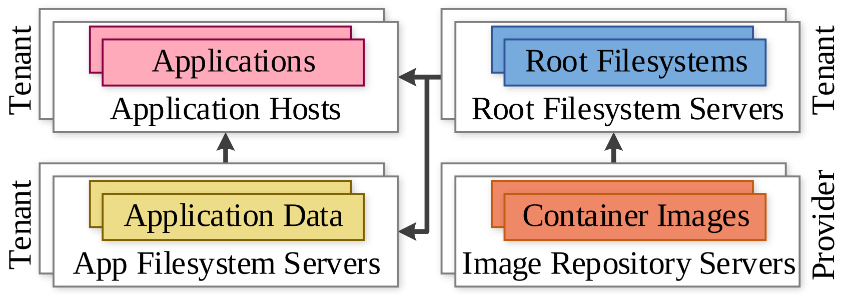


Figure 3.2: **The container storage systems.** The tenant applications run in containers that depend on two storage systems: (a) The Root Storage System for access to the root filesystem tree, and (b) the Application Storage System for access to the application data. The Image Repository is a third storage system that is used by the Root Storage System for access to the container images.

filesystem servers the images of the application filesystem servers and the application. (iv) Launch the application filesystems and the applications over the root filesystem clients.

3.2.2 File-based vs Block-based Storage

A network storage client accesses the binaries or data in the form of blocks, files, or objects [52]. We focus on block-based or file-based clients, which can efficiently serve the container storage needs. A *block-based client* serves the block volume on which we run a local filesystem. Treating entire volumes as regular files facilitates common management operations, such as migration, cloning and snapshots, at the backend storage. Despite this convenience, a block volume is only accessible by a single host and incurs the overhead of mounting the volume and running on it a local filesystem. In contrast, a *file-based client* natively accesses the files from a distributed filesystem. Multiple hosts directly share files through distinct clients, but with the potential server inconvenience of managing application files rather than volumes. The file-based or block-based clients are widely used in container storage and we should support them both.

3.2.3 Caching and Deduplication

The root filesystems of a tenant should be isolated and managed efficiently. The efficiency refers to the *storage space* of the backend servers, the *memory space* of the container hosts, and the *memory* or *network bandwidth*. The *block-based storage* accommodates a root filesystem over a separate block volume possibly derived from an image template. The backend storage supports volume snapshots to efficiently store the same blocks of different volumes only once. Under the volume clients, a shared cache transfers the common volume blocks of the tenant over the network only once [185]. A separate cache in the root filesystem lets the container reuse the recently accessed blocks without additional traffic to the shared client cache. Alternatively, the *file-based storage* accommodates a root filesystem over a shared network filesystem accessed by the client of the tenant [186]. A separate union filesystem over each root filesystem deduplicates the files of different container clones to store them only once at the backend. The common files of the container root filesystems are transferred only once to the host with a shared cache inside the distributed filesystem client. The above caching and deduplication procedures can be similarly applied to application filesystems.

3.3 Problem Definition and Motivation

The applications in the datacenter typically run on client machines that access server machines over the local network (Figure 3.3). In existing infrastructures, the applications run inside containers over the shared operating system of each client machine. Correspondingly, the server machines run common services (e.g., distributed storage) that are shared across the different tenants.

The persistent storage of container images and data through the kernel introduces several isolation, efficiency, and flexibility issues. Examples include the suboptimal performance [104, 50, 200], security [157, 201, 202] and fault-containment [203], the excessive use of processing, memory and storage resources [108, 109, 45, 185], and the constrain of containers to common system policies and configuration [50, 51]. In the rest of this section we discuss these issues in more detail.

3.3.1 Isolation

The containers cannot isolate the I/O performance of competing workloads (e.g., noisy neighbor [49, 45]) because they share a common kernel path for the critical I/O operations of request routing [104] (e.g., VFS), page cache [51], filesystem [204], and

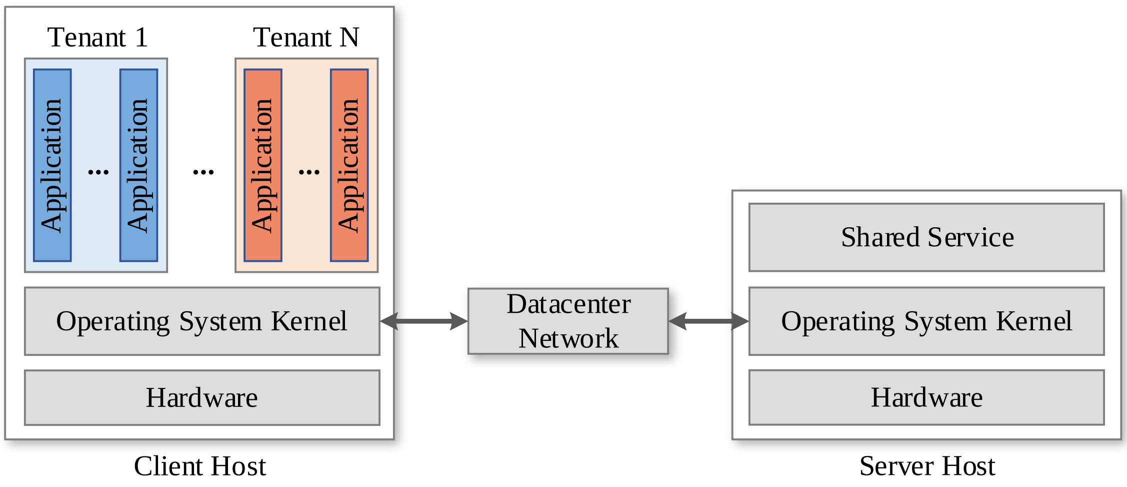


Figure 3.3: **The legacy cloud stack.** In existing systems, the containers of different tenants typically run on the shared operating system of the client machine and access shared remote services over the datacenter network.

device scheduling [159]. Additionally, the security isolation and fault containment of the colocated containers are limited due to the potential vulnerabilities from the shared operating system [157, 201, 202, 182]. A shared storage stack raises security isolation issues due to vulnerabilities from colocated containers, the host operating system and the network. In particular, the large size of the legacy storage stack that is shared among the colocated containers increases the size of the shared TCB. The larger the TCB, the more likely it contains bugs and vulnerabilities that an adversary can exploit to breach the security of a container. In addition, an adversary may exploit the shared caching layer of the VFS to launch cache side-channel attacks. Finally, fault isolation is an additional major concern, as a fault in a software component of the storage stack can affect the whole host. For instance, if the execution of a container triggers a bug in the code of a kernel-level filesystem, its effect may have a catastrophic impact on the entire host.

A library operating system (libOS) reduces the isolation dependence from the kernel resource management because it shifts functionality from the kernel to the user level. Yet, the typical linking of a libOS to a single process complicates the multiprocess state sharing of typical applications [153, 196, 105, 205, 70].

We examine the performance variation of multiple data-intensive workloads colocated over the same host [104, 107]. We run the Filebench [206] Fileserver (FLS) over a storage cluster accessed through the CephFS kernel-based client (setup in

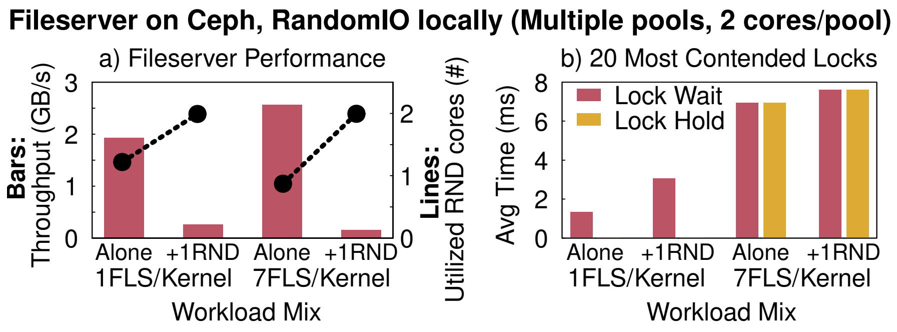


Figure 3.4: **Shared kernel contention.** This figure depicts the dramatic drop of the Fileserver throughput due to the core (a) and lock (b) contention inside the shared host kernel.

§6.6.1, §6.6.2). Alternatively, we use the Stress-ng [207] RandomIO (RND) to generate random I/O over a local kernel-based filesystem. Each workload instance runs on a distinct reservation (*container pool*) of 2 cores and 8GB RAM. We activate 4 or 16 cores of the host to respectively run 1 or 7 instances of FLS (1FLS, 7FLS) alone or next to 1 RND.

At the bar chart of Figure 3.4a, we observe a dramatic performance drop when FLS is colocated with RND. The FLS throughput drops 7.4x from 1FLS to 1FLS+1RND, and 16.5x from 7FLS to 7FLS+1RND. We attribute this behavior to two reasons. First, the system kernel utilizes the activated cores of the entire host to flush dirty pages. The line chart of Figure 3.4a confirms that the cores of RND are utilized 122% by 1FLS and 87% by 7FLS, when either runs alone. With the RND also running, the FLS throughput plummets because the kernel cannot "steal" any more the cores of RND to serve FLS. The second reason is the locking activity inside the kernel. In Figure 3.4b, we show the average wait and hold time that the kernel spends *per lock request*. It is notable that the lock wait time of 1FLS grows 2.3x in 1FLS+1RND and 5.2x in 7FLS. Therefore, the colocation of 1FLS with either 1 RND or 6 FLS instances increases the locking overhead by several factors.

We also run real-world data-intensive workloads into multiple pools on a host to demonstrate the multitenant isolation limitations of filesystems based on kernel modules. A pool reserves 2 cores and 8GB RAM of the host to run one container with the RocksDB [208] storage engine serving a 50-50 mix of put/get operations (setup details in §6.6.1, §6.6.3). Each pool uses its own FUSE-based [48] or kernel-based Ceph [24] client to mount a container root filesystem from a Ceph storage cluster.

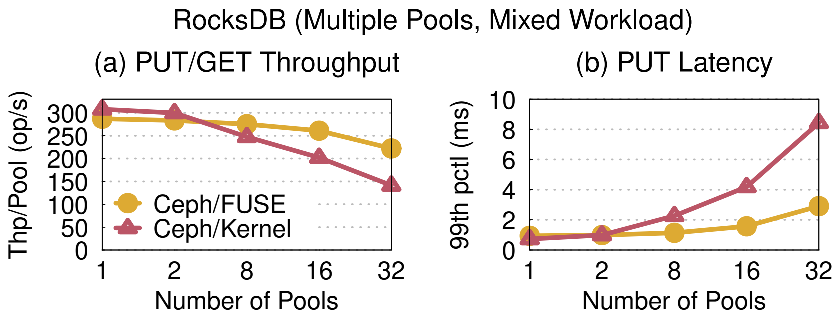


Figure 3.5: **Limited multitenant isolation.** This figure depicts the limited I/O isolation shown by the decreasing throughput and increasing tail latency per pool of RocksDB over the Ceph storage.

The root filesystem accommodates both the container image and the RocksDB data files.

Ideally, the performance per pool should be constant at number of pools up to the host capacity. However, as we increase the pools from 1 to 32 in Figure 3.5a, the per-pool throughput (higher is better) drops by 23% and 54% in the FUSE and the kernel, respectively. Correspondingly, the 99%ile of the put latency (lower is better) increases up to 3.1x and 11.5x with respect to a single pool (Figure 3.5b).

With extensive kernel profiling, we found that the kernel path limits the tenant I/O isolation for two reasons. First, the pools compete in the kernel on shared data structures (e.g., filesystem metadata) causing excessive lock wait time. Second, the I/O handling leads to kernel background activity (e.g., dirty page flushing) on resources (e.g., cores) of pools unrelated to the activity.

3.3.2 Efficiency

A filesystem provides flexible resource management when it is accessed through the kernel but runs at user level. However, it sacrifices performance and efficiency due to the extra data copying and mode or context switching (e.g., FUSE [48]). As container applications access a filesystem client through the VFS, the kernel needs to copy data from a user buffer to a kernel buffer or the opposite. The number of copies that the kernel performs to service an I/O request can be large (e.g., FUSE [48]). Copy operations can put pressure on the memory bus and pollute the CPU caches. Another issue relates to execution mode and context switching. If a kernel-level filesystem is used, every I/O request issued by an application leads to two mode switches: the execution mode of the application process changes from user mode to kernel mode, the kernel handles the I/O, and the execution mode of the application process is switched back to user mode. Worse, if a hybrid filesystem is used, every I/O request causes four mode switches in the worst case, if it is serviced by the user space part of the filesystem. Mode switches cause direct (e.g., saving registers) and indirect (e.g., TLB flushes) performance costs. Hybrid filesystems can also cause context switches, because applications and the filesystem client run in different processes.

The sharing of the legacy storage stack also raises resource accounting issues. For instance the Linux kernel uses the cgroups mechanism to limit and account the resource usage for a set of processes. However, the cgroups mechanism has been criticized for inaccurate consumption accounting of shared kernel resources (e.g., page cache [107]).

Additional inefficiencies arise in cloned containers running on the same host. Indeed, their concurrent execution introduces duplication in the utilized memory and storage space, or the memory and I/O bandwidth. The multi-layer union filesystems and block-based copy-on-write snapshots reduce the waste of storage space. Yet, the duplication of the memory space and memory or I/O bandwidth is not always prevented by the union or snapshots [209, 109, 185].

3.3.3 Flexibility

A shared storage stack hinders flexibility by constraining the collocated containers to common policies regarding data consistency, page replacement, and naming. This lack of flexibility often arises due to the fact that features, for which flexibility might be desirable, tend to be broadly scoped and have a system-wide effect.

The caching layer of the kernel exports various parameters to the user space that control its consistency policy by specifying the amount of dirty data in the system, the duration that a page can be dirty, and the triggering of writeback. There are also kernel parameters that control the page replacement policy of the kernel by specifying the aggressiveness of swapping. These parameters have a system-wide effect and affect all the collocated containers in a host. However, applications that run in different containers may have different needs regarding consistency and page replacement and a common policy may impact their consistency level or performance.

Additionally, the containers of a tenant may use an unprivileged user namespace to name users and groups. This precludes a container process from safely mounting a kernel-level or a hybrid filesystem (e.g., FUSE), as the mount is a privileged operation that can be performed only by processes that belong to the user namespace of the host. The unprivileged mounting of a filesystem is a long-standing challenge due to possible corruption and security issues [210].

3.3.4 Takeaway

The resource contention and the inflexible sharing of the system kernel reduces the container I/O isolation, increases the waste of shared datacenter resources, and constrains tenants to common policies. Current systems explicitly reserve resources for user-level execution, but they are blurry in the allocation and fair access of the kernel resources. Given the undesirable variability consequences, we argue that the effective container isolation requires two types of system support: *first, the explicit allocation of the hardware resources utilized at both the user and kernel level; and second, the fair access to the kernel operations and data structures.*

3.4 Solution: Per-tenant User-level Filesystems

The container isolation and resource efficiency can be addressed with improved scalability in the state management of the system kernel. Nevertheless, refactoring the kernel for scalability is a long-standing challenging problem [192].

As a pragmatic alternative, we propose to move basic I/O system services from the kernel to user level and provide distinct user-level services to the colocated tenants to allow configuration, performance and security isolation, and flexibility. Accordingly, we let each process group stick to their reserved hardware resources and run their own system services at user level.

In our solution, we recognize that both the applications and their remote services can run fully at user level rather than rely heavily on the local operating system kernel. Essentially, both the client and server machines run containers for the application and the server processes, respectively (Figure 3.6). Our approach can be beneficial for a broad range of system components, including the local and remote access to different types of data storage, or the network communication.

The application and server containers run at user level and access over user-level

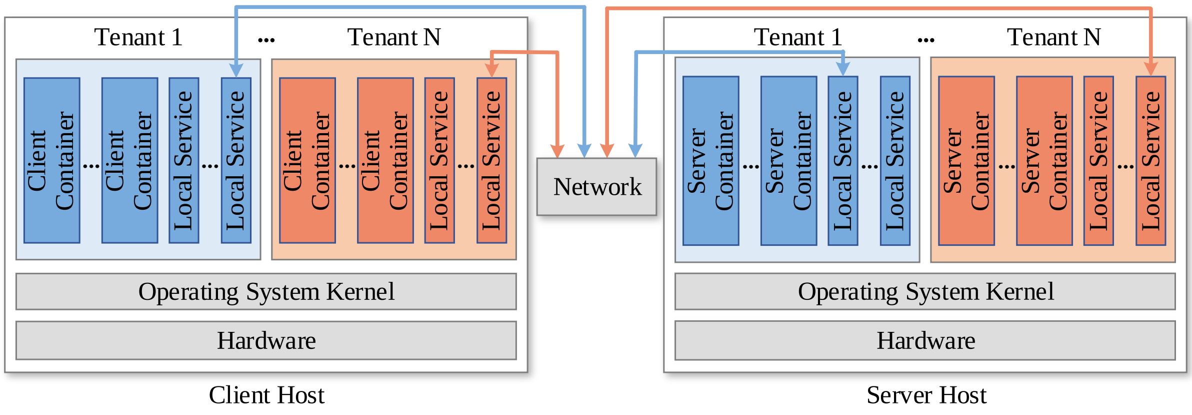


Figure 3.6: **Per tenant user-level filesystems.** In our solution, for performance and security isolation each tenant uses its own containers and local user-level services at both the client and server side inside the datacenter.

interprocess communication their local services. The local services themselves run at user level and provide access to local devices, such as storage, network and accelerator cards. In our architecture, we devote to each tenant its own application and server containers along with their interprocess communication and local services.

We introduce the libservices framework (§4) as a unified user-level storage abstraction to dynamically provision per tenant container root filesystems, application data filesystems, and image repositories. Based on libservices, we design the Polytropon toolkit (§5) as a collection of user-level components configurable to build several types of filesystems. The toolkit provides an application library to invoke the standard I/O calls, a user-level path to isolate the tenant I/O traffic to private host resources, and user-level filesystem services distinct per tenant (Figure 3.7). We use the Polytropon to build the Danaus client architecture (§6) to let each tenant access the container root and application filesystems from network storage through a private host path. Danaus integrates per tenant a union filesystem with a Ceph distributed filesystem client and a configurable shared cache. Our approach achieves several benefits:

1. We offer predictable behavior because each tenant depends on private resources and software components. Practically, this means that the throughput and latency perceived by the end users can be configured through the allocated resources and the running software rather than the activity of neighboring tenants.

2. Additionally, we achieve resilience because we minimize the reliance on the common attack surface of the operating-system kernel.

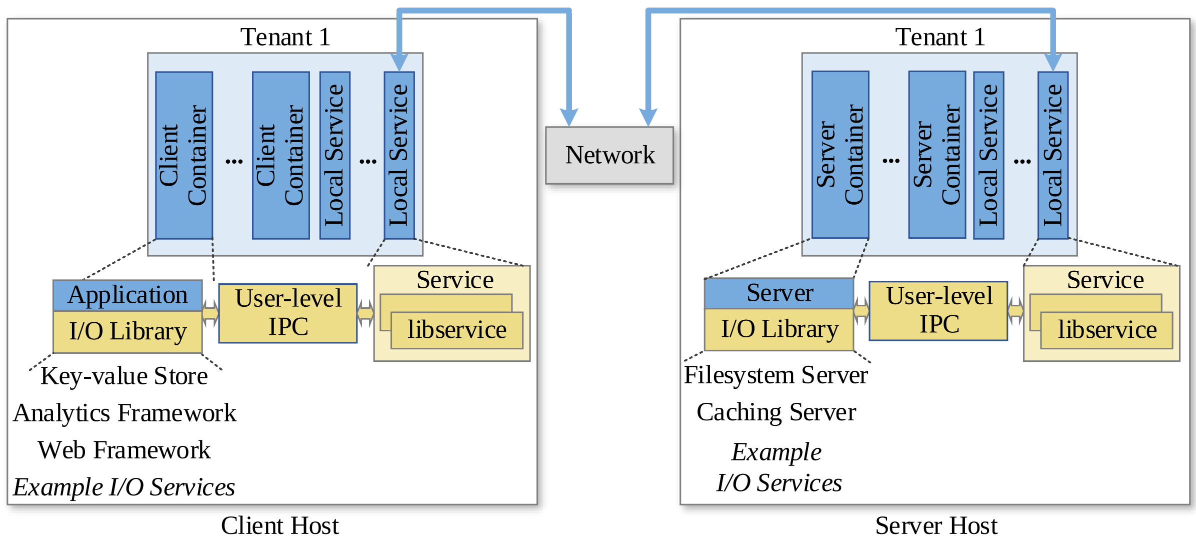


Figure 3.7: **Tenant storage provisioning.** At both the client and server side over the datacenter network, a tenant runs its own containers, interprocess communication, and local services at user level.

3. The integration of a union filesystem with a filesystem client at user level allows the containers to use the allocated resources more efficiently by preventing resource duplication at the client-side, and avoiding the costly interactions with the host kernel, like memory copies and mode switches.

4. Moreover, we provide elasticity because we can deploy the software infrastructure for each tenant dynamically. This includes not only the applications, but also the client and server side of distributed storage systems and other facilities (e.g., storage engine, local filesystem, network stack software).

3.5 Summary

Containers are increasingly adopted in cloud environments as they enable resource consolidation at low overhead. Although their benefits follow from the native resource management of the operating system, they are still used in combination with hardware-level virtualization for strong isolation.

The containers cannot isolate the I/O performance of competing workloads (e.g., noisy neighbor [49, 45]) because they share a common kernel path for the critical I/O operations of request routing [104] (e.g., VFS), page cache [51], filesystem [204], and device scheduling [159]. Indeed, the system kernel of a cloud machine serving software containers can become performance bottleneck and attack surface for the colocated tenants. The problem is caused by both the high utilization of the hardware resources serving the kernel and the software components of the kernel shared across the tenants in non-scalable ways. We experimentally verified that the host kernel limits the performance of multiple colocated containers even when they run on distinct resources. This limitation arises from the contention in the shared data structures of the kernel and the common hardware resources on which the kernel services run.

The security isolation and fault containment of the colocated containers are also limited due to the potential vulnerabilities from the shared operating system [157, 201, 202, 182]. What is more, relying on the shared kernel of I/O handling leads to inefficient and inflexible usage of the host resources.

Our key idea is to isolate the I/O path of each tenant by running dedicated storage systems at user level on reserved resources. In the following chapters, we introduce the libservices framework to dynamically provision per-tenant user-level storage systems, the Polytropon toolkit to build user-level filesystems, and the Danaus client architecture to provide client-side filesystem support to container hosts.

Chapter 4

The libservices Framework

In this chapter, we introduce the libservices as a unified user-level storage abstraction to dynamically provision per-tenant container root filesystems, application data filesystems, and image repositories. We outline several examples of container storage systems whose clients and servers can be composed from libservices. With a filesystem client prototype that we built, we experimentally demonstrate the improved multitenant performance of libservices in comparison to existing client systems based on kernel modules. Unlike libservices, the kernel client utilizes unallocated cores to achieve high filesever performance. As a result, its performance drops up to 12.9x, when the fielserver container runs next to a data-intensive container that performs random I/O. We also use RockDB over network storage to serve up to 32 tenants and conclude that the libservices keep its throughput stable, as the number of tenants increases, and up to 2.1x higher in comparison to a kernel-level client.

4.1 Overview

We target the data-intensive applications running in a multitenant cloud infrastructure. Both the applications and their storage servers run in containers launched by an orchestration system over images retrieved from a scalable storage system.

We introduce a unified framework to provision per tenant container storage systems with the following goals:

1. **Isolation** refers to the resource utilization of the tenants limited by their configured reservations.

2. **Elasticity** refers to the dynamic adjustment of the tenant resource allocation according to the configured reservations and utilization measurements.

3. **Efficiency** refers to virtualization cost approximating an operating system running on bare metal.

4. **Compatibility** refers to unmodified application binaries running on stock system software and hardware.

The performance isolation of the data-intensive applications requires fair handling of the I/O requests across the colocated tenants. The system kernel provides configurable limits (e.g., Linux cgroups) of the hardware resources across the different process groups. However, it does not guarantee the fair allocation of the high-level system services. A striking example is the shared page cache of the host [51]. We can use the Linux cgroups to pin the process groups to specific cores and assign the resident pages to the processes that first requested them. Nevertheless, when the dirty pages exceed specific thresholds, the kernel flusher threads run on arbitrary system cores rather than those reserved by an I/O-intensive process (§3.3). Moreover, the I/O activity of a process may saturate a kernel data structure (e.g., inode cache) that a different process requires to handle the I/O activity within its reserved resources.

In addition to hardware resources (e.g., core shares, memory space, I/O bandwidth), the system allocation policy should provide fair access to the kernel operations and the underlying shared data structures. Every lock that a process requests should be fairly granted according to the fraction of the host capacity reserved by the process. This objective requires enormous engineering effort to refactor the entire kernel software for container performance isolation. The related research has been

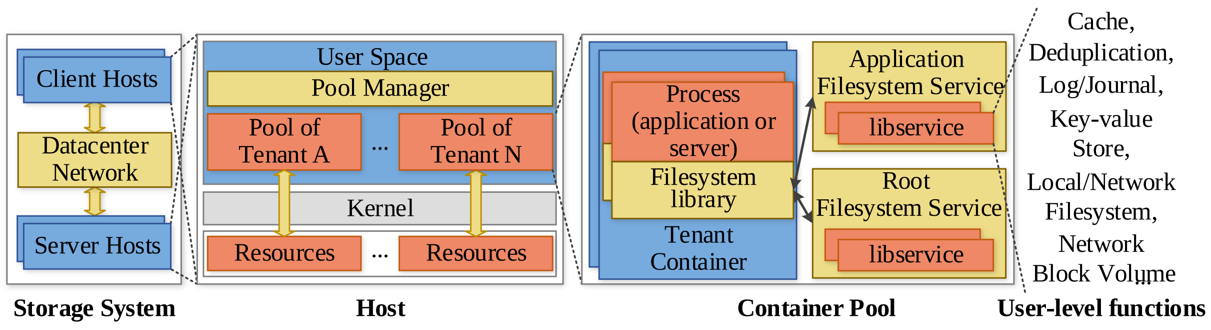


Figure 4.1: **Per-tenant storage provisioning.** The container pool of a tenant uses dedicated filesystem services built from user-level libservices. The figure depicts that the same design pattern can be used both at the client hosts that accommodate the application containers of a tenant and at the server hosts that accommodate the service containers of a tenant-provisioned distributed storage system.

productive but long-standing due to the complexity of the structural redesign and code implementation [192].

Our key idea is to let each tenant run both the applications and system services at user level over reserved hardware resources. A distributed storage system provides elasticity over multiple data and metadata servers. The application hosts provide per tenant shared storage and efficient caching. The storage servers support efficient copy-on-write deduplication (e.g., for image cloning). The resource allocations adapt dynamically to the tenant requirements and the current utilization. Our unified framework achieves multitenant I/O isolation at reasonable engineering effort. It consists of the user-level services derived from existing user-level codebases. The framework provides direct access to local storage devices or filesystems, and client access to remote storage devices or distributed filesystems. An application or server process accesses the service framework over a user-level communication component through a linked library.

The *container pool* is a collection of containers and namespaces on a machine from a tenant (Figure 4.1). The *filesystem service* is a collection of user-level I/O services implementing a local or network filesystem. The *libservice* is a user-level layer implementing a particular filesystem functionality. The *pool manager* is responsible to allocate resources elastically according to the recorded reservations and utilizations. A process obtains access to the filesystem service through a preloaded or linked *filesystem library* and communicates with the filesystem service over shared memory.

4.2 System Design

The users of a dynamically provisioned storage system are the applications, the pool managers, and the orchestration system. The clients and servers of a storage system consist of libservices running in the container pools of the tenant (Figure 4.1). The system kernel provides access to the local network or storage devices and reserves the hardware resources of each tenant.

4.2.1 Libservices

The libservice is a standalone functionality that serves as component of a filesystem service. It provides a storage function, such as caching, deduplication, key-value store, local filesystem, network access to a file or block storage system. A libservice is derived from an existing library that runs a storage function at user level and makes it accessible through a POSIX-like interface (e.g., FUSE [48], LKL [211], Rump [196]).

An existing library cannot be used out-of-the-box as libservice for several reasons. First, if a library is linked or preloaded directly to a process (e.g., Direct-FUSE [212]), it complicates the multiprocess sharing. Second, the I/O routing through the kernel (e.g., FUSE) introduces overheads from frequent mode switches and memory copies. The cost is even higher if more than one libraries are combined to a complex service. Third, the I/O routing through the kernel causes the shared structure contention that we strive to avoid. We overcome the above limitations by separating the filesystem libraries from their framework interface (e.g., FUSE). We port the library to an independent libservice with POSIX-like I/O functions expanded to include the libservice object as parameter without dependencies from global variables. A filesystem service is constructed from a list of stacked libservices, or a tree of libservices more generally.

4.2.2 Client

The client side serves the data-processing applications, the pool managers that instantiate the application or server containers of the tenants, and the orchestration system that loads the container images to the root filesystem servers. The filesystem service of an application filesystem provides either (i) a shared cache with the client of a network filesystem, or (ii) a shared local filesystem with cache on a network block volume. Instead, the filesystem service of a container root filesystem provides either (i) a union filesystem on a shared network filesystem client with cache, or (ii) a local filesystem with cache over a network block volume with shared block cache underneath.

4.2.3 Server

The filesystem service of a storage server maintains data and metadata chunks on the local storage devices. Depending on their I/O characteristics (e.g., size, rate), the chunks can be stored over a single local filesystem, or a local filesystem and a key-value store [213]. Each local filesystem or key-value store can have a separate cache. For fast recovery from a crash, the received chunk updates can be appended to the write-ahead log of the key-value store and the journal of the local filesystem, or a write-ahead log over a common local filesystem used for both the data and metadata. We can use a separate libservice to implement each of the local filesystem, key-value store, write-ahead log or journal. The server processes communicate over shared memory with the filesystem service and over the network with the clients. The server processes and filesystem services run isolated in the container pool of the tenant on the storage server machines.

4.2.4 Interprocess Communication (IPC)

A filesystem service is invoked by an application or server process through user-level IPC to minimize mode switches and processor cache stalls. The IPC component is implemented with a circular queue over shared memory inside the IPC namespace of the container pool. The I/O requests of the application and server processes are placed in the circular queue to be extracted by the filesystem service. The request passes through the top libservice of the filesystem service before it will reach the local storage device or the network client if necessary. The response is sent back without queue involvement through a shared-memory buffer prepared by the sender and referenced by the request.

4.2.5 Local Resource Management

The resource and device management at each host critically isolates the container pools running the applications, servers, and their libservices. The resource reservation approximately guarantees a specified amount of cpu shares, memory space, local storage space, and I/O bandwidth of local network or storage devices. The resource management tracks the resources, accounts them to processes and dynamically allocates them according to the current reservation and utilization. The device management provides protected operation of the local devices among the processes of the colocated tenants.

For the resource and device management we rely on kernel mechanisms, such as the Linux cgroups (v1/v2) and device drivers. This decision is justified by the satisfactory accuracy of their hardware resource accounting to user-level processes [36]. Based on the above kernel mechanisms, we build the filesystem services and their interprocess communication running at user level. Alternatively, it is possible to also manage the network cards and storage devices directly from user level. This approach has been investigated extensively for the fast access of high-performance devices, such as persistent memory and low-latency network cards [214, 63, 198].

We can construct libservices based on existing libraries to run the network protocols or access the storage and network devices in a filesystem service. Although not included in our prototype, this approach is likely to improve the I/O isolation as a result of the reduced software contention inside the system kernel. However, it requires direct device access from the filesystem service. As a result, it opens up interesting cross-tenant coordination and security issues that can be complex and warrant further investigation [70].

4.3 Example Storage Systems

In this section we describe examples of container storage systems that can be built from libservices. The clients and servers are initiated by the pool managers or the orchestration system.

4.3.1 Container Image Storage System

The Container Image Storage System is a repository of container images maintained by the provider or a tenant. The server stores image archive files over libservices for persistent local storage with cache and block-level deduplication (e.g., ZFS [215]). The client uses a libservice for network storage access with cache (e.g., NFS). The system distributes the container images of the root filesystem servers, the application filesystem servers, or the applications.

4.3.2 Root Filesystem Storage System

The Root Filesystem Storage System is dynamically provisioned to boot the applications and application filesystem servers. The server provides persistent local storage with cache and block-level deduplication. It is based on libservices of a local journaled filesystem (e.g., ext4) or a recoverable key-value store (e.g., RocksDB [208]). The client supports efficient container cloning based on libservices of a union filesystem (e.g., OverlayFS [216]) over a shared network storage client and cache (e.g., CephFS [24]). Alternatively, it is built from the libservices of the journaled filesystem with cache (e.g., ext4) on a network block device (e.g., RBD [24]) and a shared cache (e.g., Redis [217]). A pool manager starts a client in a pool to mount the root filesystems of the containers running the applications or the application filesystem servers.

4.3.3 Application Filesystem Storage System

The Application Filesystem Storage System is dynamically provisioned to serve the data-intensive or stateful applications. The server provides persistent local storage with cache based on the libservice of a local journaled filesystem (e.g., ext4) or a recoverable key-value store (e.g., RocksDB). The client libservice provides network storage access with cache (e.g., CephFS).

4.4 System Prototype

We have built a prototype of the libservices framework to provision the client side of the root filesystem storage system. It consists of a filesystem service, the user-level shared-memory IPC, and the filesystem library. The applications preload the filesystem library to override the standard I/O function calls without recompilation. The filesystem service uses one libservice to implement the user-level client and object cache of the Ceph distributed filesystem [24]. The libservice implementation is based on the open-source libcephfs library.

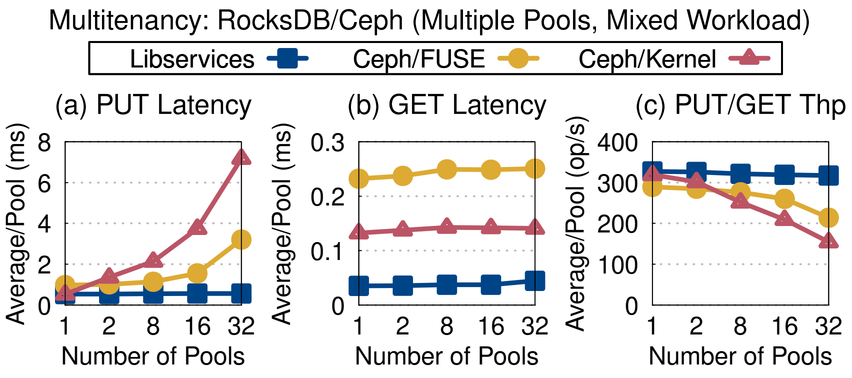


Figure 4.2: **Multitenant contention.** The isolated libservices lead to faster response and flat throughput/pool of RocksDB/Ceph.

4.5 Experimental Evaluation

In this section we experimentally demonstrate the improved multitenant performance of libservices in comparison to existing client systems based on kernel modules.

4.5.1 Experimentation Environment

Our comparison includes libservices, the kernel-based client (CephFS), and the FUSE-based client (cephfs-fuse) of Ceph. As client and server of Ceph, we use two Linux (v5.4.0) quad-socket x86-64 machines of 64 physical cores clocked at 2.4GHz and 256GB RAM each, connected with a bonded pair of 10Gbps Ethernet links. The server is partitioned into 6 storage servers (OSD), 1 metadata server (MDS), and the host, with 8 cores and 32GB each, using Xen. Each OSD and MDS uses 24GB RAM as ramdisk for fast local storage.

The results are averages of experiments repeated (up to 10 times) until the half-length of the 95% confidence interval stays within 5% of the measured average.

4.5.2 Multitenant Contention

We evaluate the multitenant contention by representing each tenant with a pool of 2 cores and 8GB RAM. Each pool uses a distinct Ceph client to mount a private root filesystem from the storage cluster and run RocksDB [208] on it. For configuration simplicity, we store the data files of RocksDB on the container root filesystem, although instantiating a separate application filesystem on the storage cluster would be relatively straightforward. We seed each database with 1GB key-value pairs of 128B

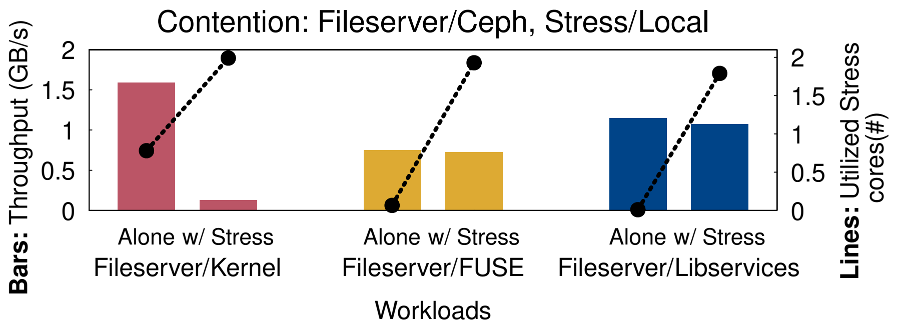


Figure 4.3: **Sensitivity to neighbor noise.** The performance of the kernel Ceph client is sensitive to contention with the Stress noisy neighbor.

key and 128KB value size, before we perform 100K random puts and 100K random gets. In 1-32 pools at the client host (Figure 4.2a), the libservice keeps the put latency in the tight range 536-571μs, unlike the FUSE and kernel that take up to 5.6x and 12.6x longer. From Figure 4.2b, the libservice keeps the get latency in 35-44μs instead of 3.9x and 6.7x longer by the kernel and FUSE. The libservice achieves faster I/O response and higher stability at increasing pools because it fully runs the client I/O at user level. On the contrary, the user-level FUSE client routes the I/O through the kernel (Linux VFS), and the kernel client fully handles the I/O inside the kernel. The improved isolation of libservices is further demonstrated by the flat throughput per pool in Figure 4.2c, unlike the 1.5x and 2.1x slowdown by FUSE and the kernel, respectively.

4.5.3 Host Utilization

We also explore how the aggressive core utilization by the kernel affects the I/O performance isolation (Figure 4.3). We configure a container with 2 cores and 8GB RAM to run the Stress (stress-ng [207]) benchmark with random 512B reads/writes on 2 local disks in RAID0. We activate 4 cores at the client machine and run a Fileserver (Filebench [206]) container of 2 cores and 8GB RAM either alone or next to the Stress container. The Fileserver throughput drops by factor of 12.9x when the kernel Ceph client runs in parallel to Stress. Instead, the Fileserver throughput with FUSE or libservices drops less than 7%. The kernel-based Fileserver performance varies substantially next to Stress because the respective I/O utilizes all the 4 enabled cores (including those of Stress) rather than only those reserved for the Fileserver.

Indeed, the system kernel handles the container I/O with all the available cores.

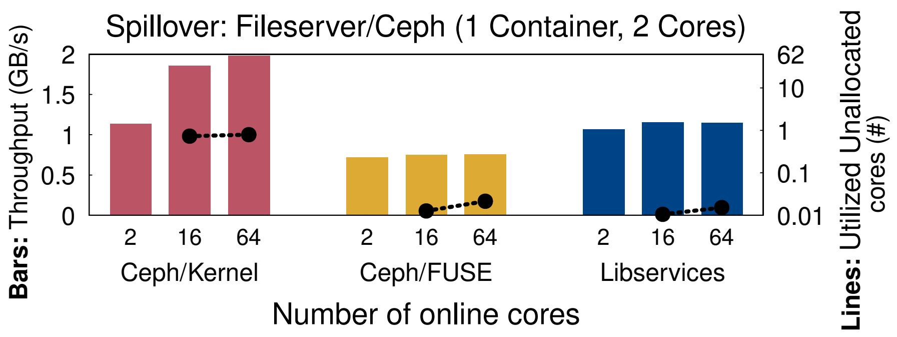


Figure 4.4: **Utilization of unallocated cores.** The kernel Ceph client increases the performance, but it utilizes the unallocated cores for that.

Instead, a user-level I/O system only utilizes the cores allocated to the served containers (Figure 4.4). We examine the Fileserver throughput over the Ceph client based on kernel, FUSE [48] or libservices. We use cgroups again to configure the Fileserver container with 2 cores and 8GB. We leave unallocated the remaining cores (up to 62), and use the system settings to enable a total of 2, 16 or 64 cores in the client host. The container runs on a root filesystem stored on the Ceph servers. Increasing the enabled cores of the client host from 2 to 64 raises the kernel I/O throughput by 75%, but only 5-8% the throughput of FUSE and libservices. The lines of the cpu utilization show that the kernel achieves higher Fileserver performance because it runs the flusher threads on cores that cgroups did not allocate to the Fileserver container.

We conclude that the tenant I/O performance can be isolated by running the filesystem functionality at user level on the reserved resources of the container pool. Although both FUSE and libservices run the filesystem at user level, libservices achieve higher performance because they bypass completely the kernel.

4.6 Summary

In this chapter we introduce libservices, a unified framework to provision per-tenant container storage systems and combine end-to-end isolation with elasticity, resource efficiency, and compatibility. Instead of relying on the shared kernel for storage services, we build at user-level storage functions derived from existing I/O libraries and create complex filesystem services for the client and server. With libservices we apply the same design pattern both at the client and server of a storage system in order to achieve end-to-end isolation. The libservices framework enables the dynamic provisioning of storage systems per tenant and honors resource limits by relying on user-level storage services.

With a prototype of the libservices framework we show that libservices achieve faster I/O response and stable performance in comparison to a kernel-based filesystem and a user-level fielsystem with a kernel-level access path. We conclude that the libservices framework is a promising I/O service abstraction to enable the dynamic provisioning of container storage systems that isolate the I/O performance among competing tenants.

Chapter 5

The Polytropon Toolkit

In this chapter, we introduce the Polytropon toolkit as a collection of user-level components configurable to build several types of filesystems. The toolkit provides an application library to invoke the standard I/O calls, a user-level path to isolate the tenant I/O traffic to private host resources, and user-level filesystem services distinct per tenant that rely on the libservice abstraction that we presented in the previous chapter.

Furthermore, we introduce the RCQB concurrent queue that relaxes operation ordering for improved communication throughput, and we provide the SMO pipelined memory copy that is faster than standard methods. RCQB achieves higher performance than the state-of-the-art queues by 4-52x, while SMO achieves 29-66% higher data transfer throughput than existing methods.

5.1 Overview

The Polytropon toolkit is a set of user-level components that can be used to implement several I/O systems per pool (§4) (e.g., filesystems, network stack, distributed storage) at user level and bypass the kernel during their execution. It consists of a POSIX-like API library linked to the cloud applications along with an efficient interprocess communication facility and a configurable stack of libservices (§4) all running at user level. The design of the Polytropon toolkit was guided by the following goals:

1. **Compatibility** Multiprocess applications access the host system services through a POSIX-like interface.
2. **Isolation** The tenant's containers access their isolated filesystems on a host over dedicated user-level paths.
3. **Flexibility** The filesystems of a tenant are flexibly configured to support cloning, caching or sharing of container images or application data.
4. **Efficiency** The containers use efficiently the datacenter resources to access their filesystems.

Polytropon achieves the above goals by relying on six design principles that we present below:

1. **Dual interface** Support common application I/O calls through the filesystem library, and only use the kernel for legacy software or system calls with implicit I/O.
2. **Filesystem integration** The libservices of a filesystem instance interact directly with each other through function calls for speed and efficiency.
3. **Multiprocess POSIX interface** We support a posix-like interface and we carefully handle process state to support multiprocess applications.
4. **User-level execution and communication** The filesystem service and the default communication path both run at user level for isolation, flexibility, and reduced kernel overhead.
5. **Path isolation** A filesystem service isolates the storage I/O of a container pool at the client side in order to reduce the interference among the colocated pools of the same or different tenants and improve their flexibility.

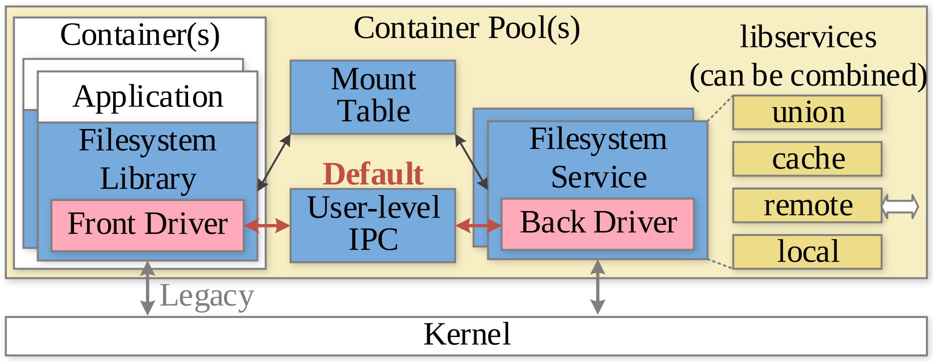


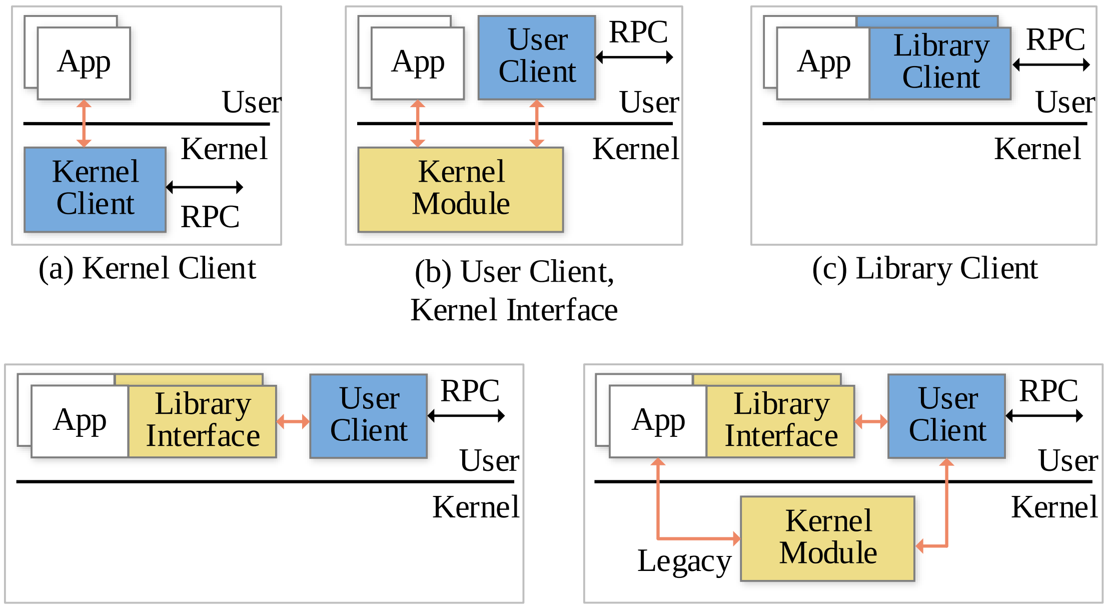
Figure 5.1: **The Polytropon toolkit.** In a container pool of Polytropon, the filesystem library communicates with the filesystem services through the mount table and the user-level IPC (default) or the kernel (legacy).

6. **Optimized queue and data copy** We can also optimize both the transport path (with RCQB), and the data path (with SMO) of the IPC for improved efficiency.

An orchestration system manages the applications in colocated containers on the host machines. Each pool operates several filesystem services according to the tenant's storage needs (Figure 5.1). The filesystem services provision both the private and shared filesystems of the containers. A container typically boots from a root filesystem and optionally mounts additional application filesystems. A filesystem consists of libservices in a filesystem service that the processes access through the *filesystem library*. The container I/O is routed to the filesystems through the *mount table*. The processes communicate with the filesystems through the *front driver* of the filesystem library and the *back driver* of the filesystem service (Figure 5.1). Although our description in the present work is influenced by the Linux containers used in our testbed, the Polytropon toolkit can be generally applied to systems that provide mechanisms for resource reservation and naming per process group.

5.2 System Design

The main Polytropon components are the dual interface, the filesystem service, the filesystem library, the mount table, and the interprocess communication. Below, we present their design and implementation along with interesting alternatives.



(a) Kernel Client (b) User Client, Kernel Interface (c) Library Client (d) User Client, User Interface (Danaus) (e) User Client, Dual Interface (Danaus)

Figure 5.2: **Filesystem types.** Three existing types of filesystems (a-c) and two that we introduce with Polytropon (d-e).

5.2.1 Interface

The interface between the application and the filesystem is a critical component that we had to specify early on in our design. We designed the interface of Polytropon-based filesystems after considering the following five options:

Full Kernel Run the filesystem inside the kernel (e.g., Ceph, NFS) for performance and access it through the kernel (e.g., VFS) for compatibility (Figure 5.2a).

User Execution Run the filesystem at user level for easy prototyping (e.g., FUSE) and access it through the kernel (Figure 5.2b).

Linked Library Link a library filesystem directly to each application (e.g., Ceph [24], Gluster [218], NFS, Hare [199]) for improved efficiency at the cost of extra complexity in multiprocess cache sharing (Figure 5.2c).

Full User Run the filesystem at user level as a standalone process and let the applications access it over shared memory through a library (Figure 5.2d). This approach combines isolation and efficiency with intra-tenant cache sharing.

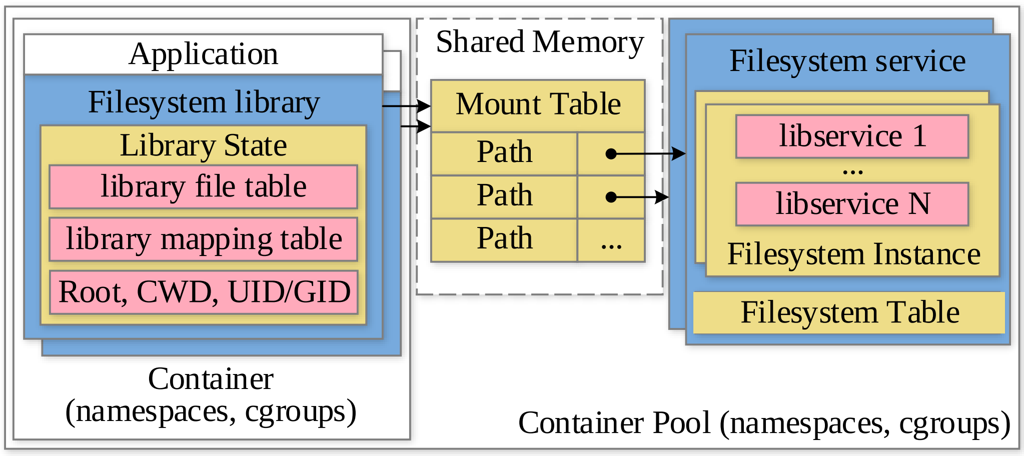


Figure 5.3: **The structure of the Polytropon components.** This figure depicts the structure of the filesystem library, filesystem service, and mount table of Polytropon.

Dual Interface For improved backward compatibility of legacy applications, we complement the full user (described above) with a second optional interface passing through the kernel (Figure 5.2e). Polytropon supports the dual interface. The filesystem service accommodates a dedicated FUSE thread to serve the I/O requests redirected from the kernel. Thus, any legacy executable can seamlessly pass to the filesystem service kernel-initiated I/O (e.g., *exec*).

5.2.2 Filesystem Service

A tenant requires the configuration flexibility to access a number of different filesystem types on a shared host. For isolation and efficiency, the filesystems of a tenant should run at user level and share functionalities in configurable ways. For example, a container typically mounts a private root filesystem sharing read-only branches with other containers, and optionally mounts several application filesystems that can be fully shared in read-write mode. Furthermore, cross-tenant filesystem sharing can be supported by establishing a common container pool among the collaborating tenants.

The *libservice* is an abstraction of a user-level filesystem implemented as a library with a POSIX-like interface (Figure 5.3). The libservices implement a range of functionalities, such as (i) a local filesystem over a direct-attached device or network block volume, (ii) a client of network-attached storage from a distributed filesystem, (iii) a union filesystem offering file-level deduplication, or (iv) a cache based on a local storage device or memory. A libservice may also support caching of the offered functionality with customized settings for a tenant application.

The *filesystem service* is a user-level process that handles the container storage I/O in a single pool (Figure 5.3). A *filesystem instance* is a mountable user-level filesystem implemented as a collection of one or multiple stackable libservices. Although the topmost libservice is unique (e.g., a union filesystem), the libservices underneath are shareable (e.g., client of distributed filesystem). A filesystem instance is specified by the libservice objects, and a *mount path* that is unique in the pool's mount namespace. When mounting a new filesystem, the filesystem service creates a new filesystem instance and inserts a new entry in the *filesystem table* indexed by a unique identifier (Figure 5.3).

The back driver of a filesystem service receives user-level I/O requests from the applications (Figure 5.1). An incoming request specifies the filesystem instance that will serve it starting from the topmost and moving to the lower libservices as needed. When the topmost libservice returns a file descriptor (e.g., serving an *open* call), the back driver instantiates a *service open file* in memory. The structure stores the open file state, which can be shared across multiple processes. The state includes the filesystem instance, the file descriptors of the libservices, the file path (e.g., *openat*), the reference count, and the call flags. Part of the state (e.g., file offset) is maintained directly in the libservices. The *service file descriptor* is the memory address of the service open file returned to the front driver as file handle. A *close* request decrements the reference count and closes the file when it reaches zero. We follow a similar approach for other types of descriptors, such as the directory streams.

The filesystem service also accommodates a dedicated FUSE thread to serve the I/O requests redirected from the kernel. The FUSE thread uses the high-level API of FUSE and operates on file paths. To handle a received I/O request, it first queries the filesystem table to retrieve a filesystem instance. Then, it forwards the I/O request to the top libservice found in the libservices stack of the filesystem instance. We use the direct I/O option of FUSE to bypass the page cache and prevent double caching inside the kernel.

5.2.3 Mount Table

The *mount table* is a hash table that translates a mount path to the serving pair of filesystem service and filesystem instance. The mount table is located in shared memory that is accessible by the applications and the filesystem services of the pool (Figure 5.3). A mount request carries the mount path, the filesystem type, and a list of options. At the mount table, it searches for the longest prefix match of the mount path:

1. **Full match** At full match with the mount path, the operation completes immediately because the specified filesystem instance is already mounted.
2. **Sharing** At partial match, if the *sharing* option is enabled in the mount request, the requested filesystem instance should share libservices with an existing instance. Accordingly, the matching filesystem service creates a new filesystem instance using the shared libservices specified in the option list. A new entry is added to the mount table for the mount path.
3. **No match** In all other cases, we initiate a new filesystem service with a new filesystem instance and add a new entry to the mount table for the mount path.

5.2.4 Filesystem Library

The filesystem library implements the I/O API which consists of synchronous and asynchronous file I/O calls, along with several management calls for processes, threads, sockets and pipes.

A major design challenge that we faced was the support of POSIX-like compatibility in the I/O system calls. In order to address it, we needed to determine the most convenient locations of the I/O path to place the file I/O state. Placing the state in the filesystem library favors the isolation but complicates the multiprocess sharing. We decided to follow a balanced approach that splits the file I/O state in a private part maintained in the filesystem library and a shared part located in the filesystem service. This decision provides the correct semantics to processes with shared state (e.g., parent and child) but requires to adjust the handling of I/O calls for correct state management.

Below we describe our steps to maintain the correct interface across different categories of system calls. Overall, we support over *160 library functions* covering *file I/O* (e.g., *mount*, *read*, *opendir*, *fopen*), *process management* (e.g., *fork*), *asynchronous I/O* (e.g., *aio_read*), *network I/O* (e.g., *send*), and *memory mappings* (e.g., *mmap*).

open We maintain the private file I/O state of a process in the *library state* structure of the filesystem library (Figure 5.3). It includes the open file descriptors, the user identifiers, the root and current directory. For each open file, the *library file table* maintains the *library open file* structure with the service file descriptor and the mount table entry. The index of the library open file is the *library file descriptor* returned to the application as file descriptor of a successful *open* call. We handle similarly an *opendir* call by storing the directory stream pointer as service file descriptor and returning the library file descriptor casted to a directory stream pointer.

sockets An I/O call (e.g., *read*) accepts a socket descriptor as file descriptor argument. Supporting this in the filesystem library required to override all the standard I/O functions that manipulate network sockets. The filesystem library stores the socket descriptor as service file descriptor, and returns to the application the library file descriptor as socket descriptor.

fork We modified the system calls to correctly handle the filesystem state during process management (e.g., *fork*, *clone*, *pthread_create*). We handle the *fork* system call in three steps. First, we instruct the filesystem service to increase the reference count of all the files that are currently opened. If this step succeeds, we call the native *fork*, otherwise we return an error value to the application. If the native *fork* succeeds, we update the process info structure of the new child process and return the result to the application. Otherwise, we instruct the filesystem service to decrease the reference count of the currently opened files and return an error value to the application. Note that the child process is an exact duplicate of the father process, and thus the state of its filesystem library matches the state of the parent's filesystem library. The handling of the *clone* system call is similar.

exec The *exec* overwrites the original library state when loading a different executable in the address space. During an *exec* call, the filesystem library first preserves the library state with a copy that it creates in a new shared memory region created with the process ID as key. Subsequently, the native *exec* loads the new executable and invokes the filesystem library constructor to recover the library state from the copy in shared memory.

mmap The *mmap* call requires from the system kernel to modify the page table and dynamically read the file contents of the mapped memory addresses. The filesystem library partially emulates this interface by mapping the specified address area to memory and synchronously reading the file contents. The *library mapping table* (Figure 5.3) is a hash table that pairs the memory address with the address area length, the file offset, the service file descriptor, the mount table entry, and the call flags. The library *mmap* call creates an anonymous mapping through the native *mmap* method. We transfer the requested file portion to the mapping area with the library *pread* call, increment the file reference count, and create a new entry in the library mapping table. We support analogously the *msync* and *munmap* calls with the library *pwrite* transferring the data to the backing file.

libc The filesystem library supports the libc API (e.g., *fopen*) by using the *fopencookie* mechanism to manage the custom I/O streams. We use the library *open* function to open a file and return the library file descriptor. We supply the filesystem library *read*, *write*, *close* and *seek* functions as hook functions to *fopencookie*.

aio We support the POSIX asynchronous I/O interface (AIO) [219] based on code from the *musl* library [220].

Dual Interface The filesystem library can be linked to a user-space application in two different ways. In the first way, an unmodified application transparently links with the filesystem library using the LD_PRELOAD environment variable. When the process image is loaded, the dynamic linker calls the constructor function of the filesystem library to create a new library instance. The filesystem library overrides common C functions (e.g., `open`, `read`, `fork`) to gain control over their control flow. However, the dynamic linker cannot override the symbols of statically linked applications, applications that use the "*syscall*" function, or the "asm" compiler intrinsic to perform system calls directly. Hence, a second way is to slightly modify the application source code in order to directly invoke the user-level I/O functions by using the common function names extended with the prefix "polytropon_".

To intercept application I/O, the filesystem library uses a set of wrapper functions around the native I/O functions. The name of each wrapper function is the name of its corresponding C library function extended with the prefix "polytropon_".

The filesystem library defines the wrappers of all the supported I/O functions in the "libpolytropon.h" header file. Thus, a developer may link an application with the filesystem library, by including the "libpolytropon.h", and calling the function wrappers for I/O operations, instead of the native C I/O functions. The GNU linker also supports intercepting standard I/O library functions dynamically at run time using the LD_PRELOAD, as discussed above. In this case, each overridden function calls the respective wrapper directly to perform an I/O operation. A special case is the handling of the overridden functions with varying number of arguments. We handle such functions by first retrieving all the arguments, and then calling the appropriate wrappers.

At the beginning of each wrapper function we retrieve a pointer to the corresponding native C function using the *dlsym* method and assign it to a static variable. The filesystem library maintains a separate static variable for each supported I/O function. We use the name of the native function extended with the "`__native_`" prefix to name each static variable. As a result, on subsequent calls of a function wrapper we don't call the costly *dlsym* method to retrieve the native I/O function but instead use the static variable.

As a backward-compatible alternative, Polytropon offers an optional legacy path that first directs the I/O call to the kernel VFS API and then to the filesystem service through the FUSE API (Figure 5.1). Thus, we skip dynamic linking or recompilation to automatically support all the system calls involving I/O (e.g., *read*, *exec*, *mmap*) at the cost of sacrificing the benefits of the default user-level path.

5.2.5 Interprocess Communication (IPC)

Below, we describe the structure of the Polytropon interprocess communication subsystem along with alternative data transfer methods that we considered.

Shared Memory The interprocess communication subsystem of Polytropon relies on shared memory to enable the communication between the applications and the filesystem services. We isolate the shared memory of each pool in a private IPC namespace [160].

We considered two different APIs that are supported by Linux: the System V IPC and the POSIX shared memory. In the former, a memory-based interface is used to

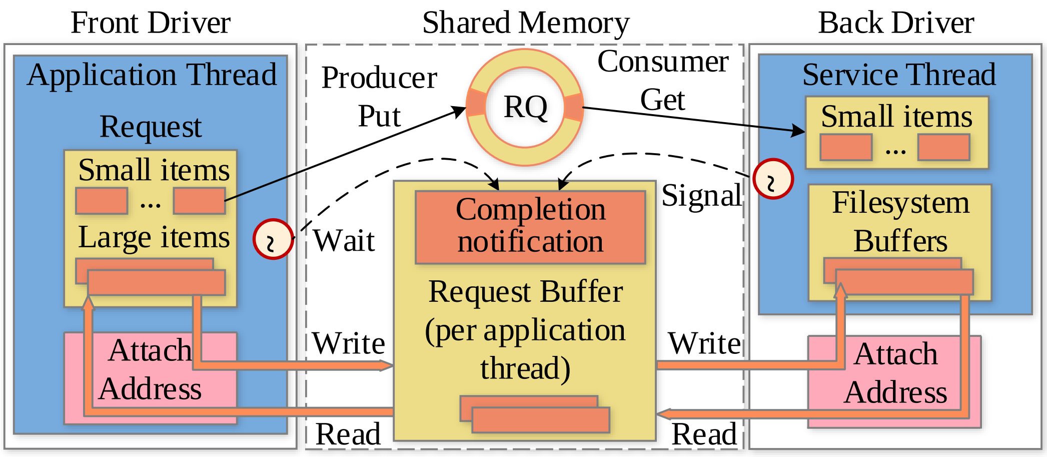


Figure 5.4: **The Polytropon IPC.** In the Polytropon IPC, the front driver transfers to the back driver small items over a request queue (RQ) and large items over a request buffer (per application thread).

create and manage shared memory segments. A shared memory segment is created with `shmget`. Then, a call to `shmat` attaches the segment to the address space of the calling process. In contrast, the POSIX shared memory mechanism uses a filesystem-based interface to create and manage shared memory segments. A call to `shm_open` creates a shared memory segment and returns a VFS file descriptor which can be used as a parameter to a `mmap` call. The `mmap` attaches the shared memory segment to the address space of the calling process. In order to avoid kernel crossing during a filesystem call we rely on the System V IPC API.

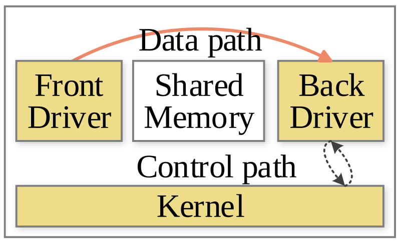
Request queue For the IPC isolation and efficiency, we apply user-level data transfer and synchronization over multiple queues in shared memory. Implementing the IPC at user-level is attractive for several reasons: (i) it reduces the number of system calls to request kernel services, (ii) it avoids copying data between the user and kernel space, (iii) it allows the usage of custom communication protocols, data structures, and copying functions for optimized communication, (iv) it is portable to different systems, because it utilizes widely available mechanisms, such as shared memory.

The front driver communicates the I/O requests to the back driver through the *request queue* data structure (Figure 5.4). We assume a typical processor that organizes the cores into groups (e.g., pairs) with shared cache (e.g., L2). We use a distinct request queue per core group to take advantage of the core parallelism and cache locality. The threads of the applications and filesystem services communicate over the request queue of the core group on which we pin them to run. The *enqueue notification* is a synchronization signal (e.g., futex [221]) that an application thread sends to the back driver for a valid request inserted at a specific queue entry. Correspondingly, a service thread sends a *completion notification* to the front driver for a specific completed request.

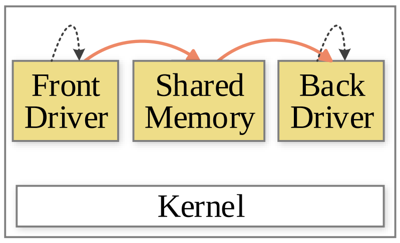
Each entry of the request queue consists of a *state* field, the *enqreq* condition variable for the enqueue notification, and a *request descriptor* structure of the I/O request. We consider an I/O argument *large* if it is a memory address (e.g., buffer pointer) or data of size exceeding a limit (e.g., 8 Bytes), and otherwise *small*. Each front-driver thread shares a dedicated *request buffer* with the back driver for the communication of completion notifications and large data items. The request buffer is located in shared memory and includes two fields: the *complreq* condition variable for the completion notification, and the *backadd* memory address of the request buffer at the back driver. The request descriptor contains the call identifier, the small arguments, the shared-memory identifier (e.g., System V key) of the request buffer, and the backadd memory address.

The first time that a front-driver thread performs an I/O request (Figure 5.4), it undergoes the one-time cost to create a new request buffer and map it to an application attach address. It also caches the local address (*frontadd*) of the request buffer at thread-local storage for reuse in subsequent requests. The first time that a back driver receives an I/O request from the application thread, it retrieves the shared-memory identifier from the request descriptor to map the request buffer to a local attach address (*backadd*). When responding back, the back driver copies the backadd address to the request buffer from which the receiving thread copies it to the next request descriptor. In the following communication between the two threads, the back driver skips the costly local mapping to retrieve the backadd address.

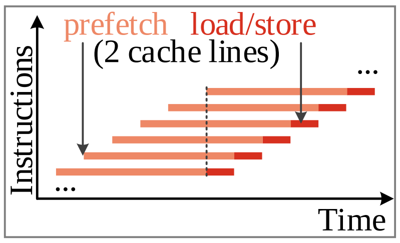
An application thread invokes an I/O call at the front driver by inserting an entry to the request queue. At a synchronous call, the thread waits by spinning up to configurable max tries (1000 in our prototype) and sleeping on the complreq field of the request buffer. The back driver notifies the waiting thread when the request completes. Thus the system preserves the program order of the synchronous I/O calls, submitted one after the other (e.g., *write* followed by *fsync* in a thread). When a thread terminates at the front driver, the request buffer is marked as deleted and unmapped from the application address space. The back driver maintains and



(a) CMA (1 Copy)



(b) SMC / SMO (2 Copies)



(c) SMO (Copy pipeline)

Figure 5.5: **Data transfer in the Polytropon IPC.** The control and data path of CMA (a), and SMC, SMO (b) transfer methods. (c) The SMO pipeline with the prefetch and copy stages per two cache lines.

periodically traverses a linked list with all the request buffers mapped to the address space of the filesystem service. Any request buffers marked as deleted by the front driver, are unmapped by the filesystem service and deleted by the kernel. We serve an I/O request with the following steps:

1. The front driver fills the request buffer with the large I/O arguments,
2. prepares a new request descriptor with the system call id, the small arguments, and the shared-memory id of the request buffer,
3. inserts the request descriptor to the request queue,
4. waits on complreq of request buffer for completion.
5. The back driver retrieves the request descriptor and a reference to the request buffer,
6. processes the request and copies the response to the request buffer,
7. sends the completion notification through the complreq of the request buffer.
8. The front driver wakes up and copies the response from the request buffer to the application buffer.

Data transfer From our extensive profiling across different systems, the memory copy operation turned out to play a critical role in the measured I/O performance. Below we describe two alternatives that we examined before developing our own user-level optimized method.

First, in the *cross-memory attach* (CMA) method, the front driver uses the request descriptor to pass the local addresses of the large I/O arguments to the back driver. The back driver uses an existing system call (e.g., Linux *process_vm_readv*) to directly copy the data between the address spaces of the application and the filesystem service with zero kernel buffering (Figure 5.5a). After processing the request, the back driver uses the opposite system call (e.g., Linux *process_vm_writev*) to directly write the large response items to the application address space. Second, the *shared-memory copy* (SMC) method similarly passes an I/O request as a request descriptor through the queue (Figure 5.5b). Instead of sending the pointers of the I/O arguments to the back driver, the front driver copies the I/O arguments to the request buffer with the libc *memcpy*. The back driver retrieves the request buffer address, reads the request buffer, processes the request, and writes back the response.

Finally, in the *shared-memory optimized* (SMO) method we follow the steps of SMC. However, we apply an improved memory copy sequence for arguments of size $\geq$1KB (Figure 5.5b). The memory copy optimization targets our testbed processor (AMD Opteron 6378 [222]). In our memory copy, we create a pipeline of two stages (Figure 5.5c). The first stage prefetches two cache lines into a non-temporal cache structure of the processor [223, 224]. The second stage transfers the two prefetched cache lines to the destination memory address through eight 128-bit registers (xmm0-xmm7 of x86-64 SSE [225]). We initiate five (128-byte) outstanding prefetches before starting the first memory store in parallel with the sixth prefetch. We use the `PREFETCHNTA` instruction to prefetch the data, and the `SFENCE` instruction to make the stored data globally visible given the weak ordering of the x86 non-temporal data store [225]. In Polytropon, we use the SMO method because it achieves higher performance, according to our experiments (§5.5.4).

5.2.6 The Relaxed Concurrent Queue Blocking (RCQB)

Polytropon targets multicore machines with network and storage devices of high speed and low latency. The interaction between an application and a filesystem is the classical producer-consumer communication between a client and a server over shared memory. The producer and the consumer in our system are different processes whose communication should achieve high throughput and low latency at high concurrency. These conditions make a preallocated data structure (e.g., fixed-size array)

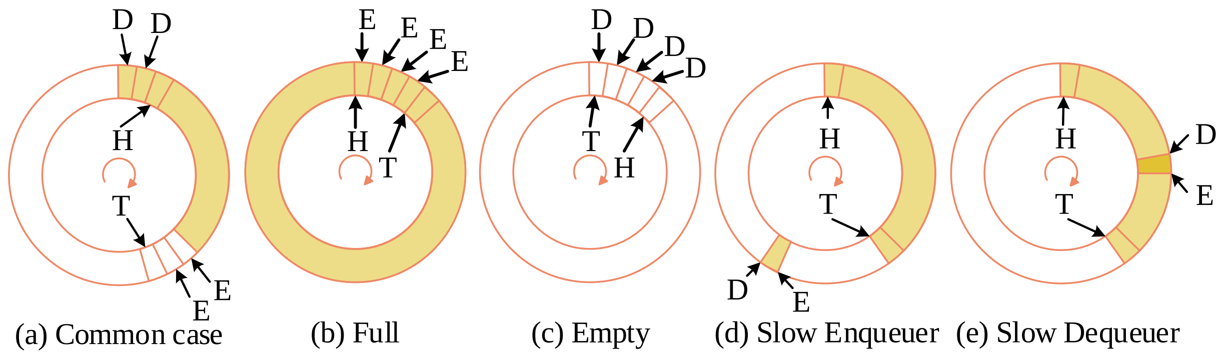


Figure 5.6: **The Relaxed Concurrent Queue Blocking.** (a) Dequeuers (D) following the enqueuers (E). (b) Enqueuers waiting in full queue. (c) Dequeuers waiting in empty queue. (d) A dequeuer waits on an entry locked by a slow enqueuer. (e) An enqueuer waits on an entry locked by a slow dequeuer We depict the free slots as white and the locked or occupied slots as colored. The interior of each queue includes the head (H) and tail (T). The enqueuers and dequeuers are assigned clockwise to entries around the circular queue.

preferable to a dynamically allocated one (e.g., linked list) for several reasons. First, a preallocated data structure supports shared-memory accesses from different address spaces without complex pointer conversions. Second, it avoids the system overhead of frequent dynamic memory allocation. Third, it skips the processing overhead of update versioning that memory pools require for consistent object recycling (e.g., ABA problem [59]). Finally, a preallocated data structure keeps constant and predictable the memory occupancy over time.

The existing systems often use concurrent priority queues for communication [226, 227, 59, 228]. A typical choice is a first-in-first-out (FIFO) queue that avoids starvation by giving priority to the item with the longest wait time. Although the FIFO ordering is fair in the above sense, it is an inherently sequential policy for two reasons. First, only one item at a time can be inserted or deleted respectively at each end of the queue. This suggests a limitation of at most two concurrent operations. Second, the correct implementation of the queue operations requires tracking the queue empty or full condition with state updates (e.g., item counting) that reduce concurrency [59]. Recent data structures aim to increase concurrency through multiple queues that are inherently sequential due to their FIFO ordering [229, 230].

Arguably, the actual order of request service in a system depends on several factors beyond the queue priority [230]. First, the time period of request preparation or service execution is determined by the thread scheduling policy. Second, the implementation of the filesystem as consumer introduces delays arising from the locking structure and the data access pattern. Third, the underlying devices also generate delays according to the workload requirements and their scheduling policy. Essentially, the unpredictable system behavior in the service path weakens the relevance of the strict queue ordering. *Indeed, the delay increase due to limited queue concurrency may exceed the delay variation due to violation of the strict FIFO order.*

Definition For increased performance in the communication between the application and the filesystem, we introduce a new queue algorithm, called *Relaxed Concurrent Queue Blocking (RCQB)*. Our key idea is to split each of the enqueue and dequeue operations into two stages. The first stage assigns the operation sequentially into a queue slot, and the second stage completes the operation. Thus, we distribute the operations across the slots and let them complete in parallel and potentially out of FIFO order. We aim to achieve high throughput of the enqueue and dequeue operations but also low *wait latency* referring to time period between the arrival and departure of an item from the queue.

RCQB is a blocking queue implemented as a circular buffer over a fixed-size array. More specifically, we call *slot* each array element consisting of the state field, the enqreq variable, and a request descriptor (item). The *tail* and *head* indexes are two integer variables that track the next slot to insert an item at and the next slot to remove an item from, respectively. We set the array size to a power of two (e.g., 256) and treat the overflow of an index variable as an atomic increment with modulo arithmetic. The slot state can take four values: *free* while a slot is ready to receive a new item; *enqpend* while an enqueuer writes a new item to the slot; *deqpend* while a dequeuer removes an item from the slot; and *occupied* while the slot contains a valid item ready to be removed.

Enqueue The enqueue operation consists of four steps: *slot allocation*, *enqpend locking*, *item insertion*, and *slot release*. We allocate the slots sequentially to the enqueuers by applying the fetch-and-add (FAA) instruction to the tail index. Thus we allocate the slot specified by the tail index before we atomically increment it (Figure 5.6a). The enqueuer atomically reads the state value of the allocated slot. If the state is enqpend or occupied, then the slot is currently being updated by another enqueuer, or it has already received an item. If the state is deqpend, then a dequeuer is currently removing a valid item from the slot. These three cases indicate that the queue is probably full (Figure 5.6b). In anticipation of the slot soon becoming free, the enqueuer *pause-spins* by staying idle for a brief time period (e.g., through the `PAUSE` instruction of x86 SSE2 [225]) before reading the state value again. If the slot state is free, then the enqueuer attempts to atomically switch the state to enqpend with the compare-and-swap (CAS) instruction. If the CAS fails, then the slot is already occupied or locked by another enqueuer or dequeuer, and the enqueuer reads the state again. When the CAS succeeds, the enqueuer has locked the slot to enqpend and inserts the item. Subsequently, it atomically sets the state to occupied and signals the dequeuer threads (if any) waiting at the slot through the enqreq variable.

Dequeue We also split the dequeue operation in four steps: *slot allocation*, *deqpend locking*, *item removal*, and *slot release*. We allocate the array slots sequentially to the dequeuers by executing the FAA instruction on the head index (Figure 5.6a). Then, the dequeuer atomically reads the state of the allocated slot. If the state is free, the dequeuer retries up to a limit of times before it will sleep on the enqreq variable of the slot to be woken up by an enqueuer (Figure 5.6c). If the state is enqpend or deqpend, then the slot is locked by an enqueuer or another dequeuer that soon will set it to occupied or free, respectively. In both these cases, the dequeuer pause-spins and reads the state again. If the state is occupied, then the dequeuer attempts to switch the slot state to deqpend through CAS. If the CAS succeeds, then the dequeuer removes the item and atomically sets the state to free. If the CAS fails, then the dequeuer pause-spins and reads the state again.

Analysis The enqueuers and dequeuers are assigned to the slots sequentially and update them concurrently. When multiple enqueuers compete at the same time, they are delayed by the speed of incrementing the tail rather than the cost of slot update. Normally an enqueuer is assigned to a free slot with FAA, and proceeds immediately to insert an item with CAS. Finding a slot non-free is possible when the number of enqueuers approaches the queue size, or the enqueuers insert items faster than the dequeuers removing them. Then, the enqueuers retry until the slot becomes free, instead of being redirected to use another slot.

Correspondingly, the dequeuers are assigned to slots at the speed of incrementing the head. Normally, the head remains behind the tail, and the dequeuers immediately remove items from the occupied slots. If the item arrival is slower than the item departure, or the number of dequeuers approaches the queue size, then the head catches up with or moves ahead of the tail. Then, the dequeuers are assigned to non-occupied slots and retry or sleep instead of being relocated to a different slot. From our experience across different workloads and thread counts or balances, the retry with pause-spin by the enqueuers and dequeuers leads to stable behavior and high performance. On the contrary, the relocation to a different slot [229, 231] can lead to increased delays and substantial performance variation in several cases.

In the unlikely case that a thread fails while it occupies a slot in the enqpend or deqpend state (Figure 5.6c,d), then any other threads assigned to the same slot are blocked. The threads that operate at the remaining slots continue their normal progress. In the worst case, all the enqueuers and dequeuers will end up blocked in the locked slot of the failed thread. This is a consequence of the blocking operation that typical blocking queues do not address. RCQB handles it gracefully in the sense that only the threads assigned to the locked slot are affected. Additionally, each tenant of Polytropon uses separate filesystem services, and each filesystem service operates separate queues across the different core groups.

In Chapter 8 we design and implement a family of Relaxed Concurrent Queues (RCQ) that consists of the blocking RCQB and a number of lock-free variations. For specific RCQ variations, we provide formal algorithms and reason about their correctness and progress properties.

5.3 Pool Management

The pool of a tenant mounts the filesystems and specifies the namespaces and resource usage limits of the containers running on a host. We manage the pools and their containers according to the parameters of their configuration files through an API of *create*, *start* and *stop* commands.

The *container engine* is a standalone process that manages the container pools of a host. Our Linux-based prototype isolates the resource identifiers of a pool with namespaces [160], specifies the processor cores and memory nodes with the cpuset controller (cgroup v1), and limits the memory usage with the memory controller (cgroup v2). We use the *clone* call to create the namespaces and launch the first process of a new pool. The container processes are forked from the first process and inherit the namespaces of the pool or create new namespaces nested underneath (e.g., PID, UID). The cgroup hierarchy provides a dedicated subtree for each pool to monitor and limit the resource usage. The controllers and processes of a container are attached to a leaf node of the subtree if the controllers are distinct from those of the pool; otherwise, they are attached to the root of the subtree.

The *storage driver* mounts a root or application filesystem to a subset of the pool processes. The filesystems are categorized by the user or kernel level of their execution or interprocess communication (four combinations). The default filesystems of Polytropon are filesystem instances constructed from stacks of user-level libservices accessible through both the user and kernel level. For the experimental comparison with existing systems, the container engine can also mount traditional filesystems based on kernel modules. These filesystems are accessed through the kernel and run at user or kernel level (e.g., VFS). More specifically, our prototype apart from the Polytropon user-level filesystem, optionally supports: (a) as backend client, the kernel-based CephFS client and the FUSE-based ceph-fuse, (b) as union filesystem, the kernel-based AUFS and the FUSE-based unionfs-fuse (fairly comparable as both are derived from Unionfs). We do not include the *OverlayFS* because the available kernel-based version is not working over remote filesystems [232]. The kernel-based filesystems use the system page cache by default with optional disabling through direct I/O (e.g., avoid double caching in FUSE).

5.4 Example Filesystems

A root or application filesystem can be constructed from a combination of basic filesystems, such as: (i) A Polytropon filesystem instance consisting of a union libservice over the client libservice of a distributed filesystem. The filesystem instance runs at user level inside a filesystem service accessed from the user level for isolation, or through the kernel for compatibility. (ii) A legacy filesystem constructed from a union filesystem (e.g., AUFS) over a local filesystem (e.g., ext4), with the path and execution of both at kernel level. (iii) A legacy filesystem composed of a FUSE-based union over a kernel-level client of a distributed filesystem. Although the FUSE union runs at user level and the client at kernel, the IPC paths of both pass through the kernel. A filesystem instance is mounted through the Polytropon filesystem service, unlike a traditional filesystem mounted through the system kernel.

5.4.1 Implementation Effort

We developed a prototype of the Polytropon toolkit in the C language with 14360 lines of code (LoCs). Table 5.1 summarizes our development effort and reports the size of various components in our prototype.

Implementation effort of the Polytropon prototype

Component	**Modified**	**New**
Filesystem library	-	7877
Filesystem service	22	2265
IPC	-	1174
Mount table	-	343
Common	-	2334
Tools	-	729
Container Engine	-	3826
Total	**22**	**14722**

Table 5.1: Implementation size measured in uncommented lines of code, not including white spaces. We measured the lines of code using the cloc [233] utility.

The Polytropon prototype is comprised of five major components: The filesystem library, the filesystem service, the mount table, and the IPC.

We implemented the filesystem service in 1903 commented lines of C. We created two libservices. The first implements a trivial local filesystem with memory buffering, and the second is based on the library-based Ceph client (libcephfs) which required the addition of 362 LoCs and the modification of 22 LoCs.

We implemented the filesystem library in 7877 commented lines of C. The filesystem library replaces the common filesystem interface that is provided by the VFS, in order to let an unmodified application to directly interact with the Polytropon user-level filesystems. By default, it is compiled as a shared library that links with user-space applications. Each application process uses its own instance of the filesystem library.

We implemented the IPC driver in 1174 lines of C. The IPC driver permits the filesystem library and the filesystem service to communicate at user-level. The filesystem library uses the front driver of the IPC driver to place application I/O requests, while the filesystem service uses its back driver to receive the requests and communicate the I/O responses back to the front driver.

We implemented the mount table in 343 lines of C and the necessary tools that facilitate the mounting and unmounting of Polytropon user-level filesystems in 729 lines of C.

Finally, we implemented the container engine in 3826 lines of C. The multitenant container engine is responsible for deploying and managing container pools on a host.

5.5 Experimental Evaluation

In this section we present a series of experiments to evaluate the performance of the key components that make up the Polytropon toolkit and experimentally answer the questions that we list below. In the next chapter, we use Polytropon to build and evaluate a client architecture to serve the storage needs of tenant containers.

1. How does the Polytropon IPC compares with a state-of the art RPC framework from the literature (§5.5.2)?
2. To what extent the IPC queue algorithm affects the application performance over network storage (§5.5.3)?
3. What is the enqueue latency and end-to-end throughput of RCQB in comparison to the best known queues (§5.5.3)?
4. How SMO benefits Polytropon over SMC, CMA (§5.5.4)?
5. How the IPC of Polytropon and FUSE compare (§5.5.4)?

5.5.1 Experimentation Environment

In our experiments we use two identical x86-64 machines of 4 sockets with the 16-core AMD 6378 processor clocked at 2.4GHz. Each machine has 64 cores, 256GB RAM and 20Gps bonded connection to 24-port 10GbE switch. The local storage includes 2 SAS disks in RAID1 for the host root filesystem. The machines run Debian 9 Linux (kernel v4.9). We repeated several of our experiments on machines with more recent processor model (Intel Cascade Lake) and kernel version (v5.4) with similar results.

The client machine uses the Linux cgroup and namespaces to configure the containers running directly on the host. The server machine uses 8 cores and 32GB RAM to run the host system, and 7 VMs (Xen v4.8) of 32GB RAM and 8 cores to run 6 OSDs (object storage devices) and 1 MDS (metadata server) of Ceph (v10.2.7). On each OSD, we use 8GB for main memory and 24GB as ramdisk with XFS to store the OSD data and journal at high performance.

In our experiments we use the Filebench [206] (v1.5-alpha3) and several custom-developed tools. Unless otherwise specified, we repeated the experiments as needed to get the half-length of the 95% confidence interval of the primary metric (e.g., throughput) within 5% of the measured average.

5.5.2 Comparison with Mercury IPC

The Mercury RPC interface is a state-of-the-art framework from ANL that supports RPC with small or large data transfers over native transport mechanisms [234, 235]. The shared-memory plugin for local node communication transfers small messages over shared memory, and large data items over the cross-memory attach (CMA) service of Linux. Mercury provides the test_perf and test_server tools that run as client and server over the same or different nodes. The test_perf fills in a buffer with integers and repeatedly (1000 times) sends it to the server and waits for the response. The reported throughput is the average rate of data transfer from the client to the server at request completion. For the comparison with Mercury, we ported the Polytropon IPC to the test_perf and test_server tools.

In Figure 5.7, we find Polytropon to be faster than Mercury up to 369.3x at 1KB (32 cores) and 22.7x at 1MB (64 cores). The comparison is based on the data throughput reported by test_perf for 1KB and 1MB buffer size on a machine with 2, 8, 16, 32 or 64 enabled cores. We set both the client and server threads equal to the

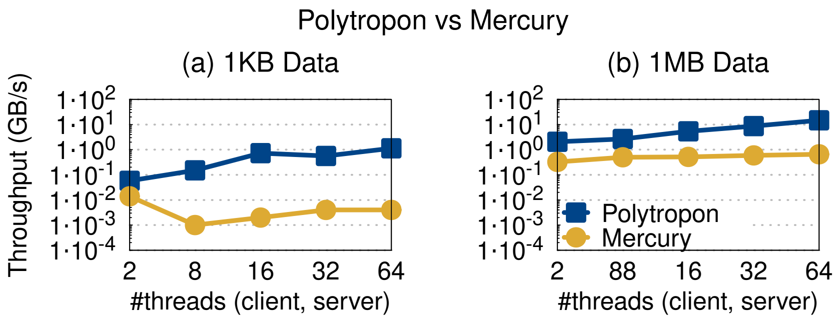


Figure 5.7: **Polytropon vs Mercury.** This figure compares the performance of the Polytropon IPC with Mercury RPC framework. On a machine with 2, 8,16, 32 or 64 enabled cores, we set both the client and server threads equal to the number of cores.

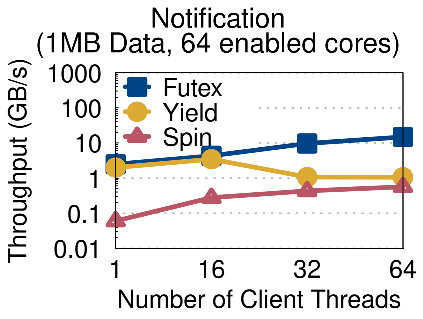


Figure 5.8: **Notification mechanism.** This figure compares the performance of various notification mechanisms in the Polytropon IPC, including the Linux futex, yield, and spin.

number of enabled cores. At increased message size up to 128MB our results were similar to those of 1MB.

We also examined the notification mechanism in the Polytropon IPC. We compared the performance of the Linux futex (fast user-space locking [221]) with an initial period of spinning (1000 iterations) used in Polytropon to that of yield, and plain spin for the request completion notification. At 64 cores with Mercury test_perf, we found futex to be up to 14.1x and 26x faster than yield and spin, respectively, as we depict in Figure 5.8.

The results highlight the improved performance of carrying out the communication entirely at user level for short and large messages, instead of relying on the kernel for large messages as the Mercury framework does. We also found that the combination of an initial spinning period with a lightweight blocking mechanism for synchronization outperforms cpu yield and plain spinning, when it is used as a notification mechanism.

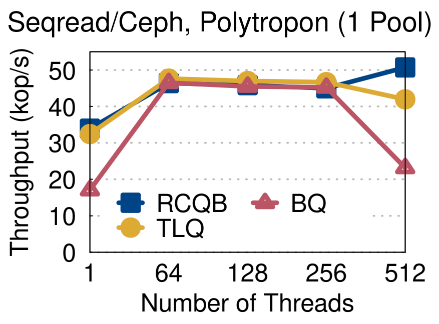


Figure 5.9: **Comparison of RCQB, BQ, and TLQ in Polytropon.** Application performance of Seqread/Ceph across RCQB, BQ and TLQ.

5.5.3 Concurrent Queues

The IPC queue of the toolkit critically affects the performance of the applications running over Polytropon. The Broker Queue [228] (BQ) is a state-of-the-art array-based blocking algorithm optimized for massively parallel systems (e.g., GPUs). On the contrary, the list-based Two-Lock Queue [226, 59] (TLQ) is a baseline algorithm with one enqueuer and one dequeuer allowed to modify the queue in parallel. We alternatively implement the Polytropon IPC with the RCQB, BQ or TLQ queue. In a pool with 64 cores and 200GB RAM, we run the Filebench [206] Singlestreamread (Seqread) with 1-512 threads for 120s on 1GB file. We store the file in the container root filesystem mounted from Ceph through Polytropon.

The enqueuer and dequeuer threads of Polytropon belong to distinct application and service processes, respectively. This is a normal requirement that prevented our implementation from using queue algorithms designed for one process (e.g., LCRQ [229], WF [231]). For high concurrency, we set the benchmark I/O size to 1KB and use a single queue to connect the benchmark application with the filesystem service. From Figure 5.9, all three algorithms perform well at 64-256 enqueuer or dequeuer threads (128-512 total). However, at 1 or 512 threads per operation type, the throughput of BQ drops to half of RCQB or less. This is expected because BQ requires a long execution path to handle an empty or full queue [228]. At 512 threads, the TLQ coarse-grain locking leads to 21% lower throughput than RCQB.

We examine additional state-of-the-art algorithms with a number of dedicated enqueue and dequeue threads running in one process over a single queue. We consider the RCQB, the lock-free LCRQ [229], the wait-free WFQ [231] and the blocking BQ [228]. The workload runs 10M enqueue and dequeue operations in a closed

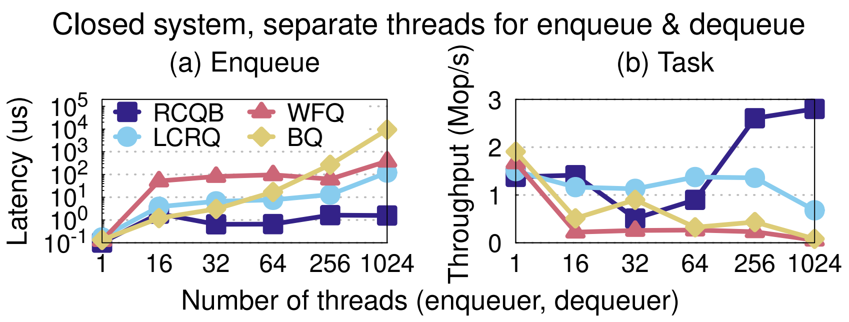


Figure 5.10: **Comparison of RCQB, LCRQ, WFQ, and BQ.** Average enqueue latency (a) and throughput (b) of synchronous tasks over RCQB, LCRQ, WFQ and BQ.

system. An enqueuer sends a synchronous request item and the dequeuer responds with a completion notification immediately after extracting the item from the queue (Figure 5.4).

In Figure 5.10a, the average *enqueue latency* of RCQB gets up to 1.6μs at 1024 threads, while that of LCRQ, WFQ and BQ is 77x, 246x and 5881x higher, respectively. In Figure 5.10b, the *task throughput* is the rate at which the enqueuers receive completions from the dequeuers. This metric encompasses the item communication between the enqueuers and dequeuers including the wait latency through the queue. At 32 threads, LCRQ is up to 2.2x faster than RCQB, but at 1024 threads, RCQB is faster 4x over LCRQ, 34x over BQ and 52x over WF. The higher task throughput of RCQB follows from the parallel completion of the enqueue and dequeue operations, which results into improved concurrency over the strict FIFO ordering of LCRQ, WFQ and BQ. We also confirmed the RCQB advantages with other metrics (e.g., tail latency) and unbalanced thread counts.

We conclude that RCQB combines low operation latency with high communication throughput making it a preferred choice, especially for multicore systems with high concurrency.

5.5.4 Sensitivity Analysis of IPC

Data transfer We compare the performance impact of the CMA, SMC and SMO data transfer methods by running a data-intensive application over network storage mounted through Polytropon. On a host with 8 enabled cores and 32GB of RAM, we generate sequential reads over Ceph with 4GB of client object cache by running the Filebench [206] Seqread with 64 threads on 1GB file. As the workload fits within the

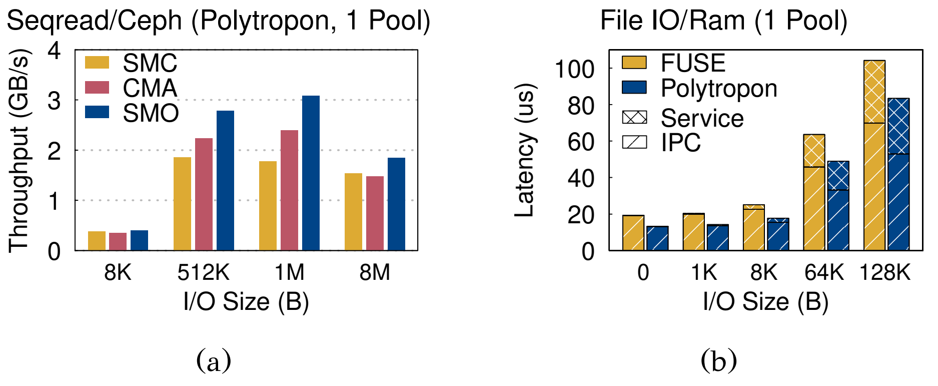


Figure 5.11: **Sensitivity analysis of Polytropon IPC.** (a) Throughput comparison of cached sequential read with SMO, CMA, and SMC, (b) Cost breakdown of Polytropon and FUSE.

object cache we mainly examine the memory performance of the client.

From Figure 5.11a, SMO has consistently higher throughput up to 3.1GB/s unlike CMA that gets up to 2.4GB/s and SMC only up to 1.9GB/s.

SMO performs 2 memory copies instead of 1 with CMA, and runs at user level rather than inside the kernel. Nevertheless, the benefit from the pipelined operation makes SMO 66% faster than SMC and 29% faster than CMA.

Polytropon vs FUSE We evaluate the performance difference between the user-level path of Polytropon and the kernel-level path of FUSE by placing the same filesystem functionality at the service side. In a pool of 8 cores and 32GB RAM, we implement a trivial local filesystem with memory buffering. The benchmark simply opens a file, invokes a read 1M times, and closes it.

For read I/O size in the range 0-128KB, we examine the average read latency (lower is better) broken down into IPC and service time (Figure 5.11b). In comparison to Polytropon, FUSE takes 25-46% longer to serve the read due to 32-46% higher IPC time.

We conclude that handling the IPC through SMO at user level makes Polytropon faster than FUSE.

5.6 Summary

The Polytropon toolkit achieves scalable storage I/O over a stock kernel in order to serve the data-intensive containers of multitenant hosts. It consists of reusable user-level components that provision composable filesystems per tenant and connect them to multiple container processes. We optimize the IPC with a relaxed concurrent queue (RCQB) and a pipelined memory copy (SMO). Our results highlight highlight the improved performance of carrying out the communication with the filesystem at user-level. RCQB achieves higher performance than the state-of-the-art queues by 4-52x, while SMO achieves 29-66% higher data transfer throughput than existing methods.

Chapter 6

The Danaus Client Architecture

In the previous chapter we introduced the Polytropon toolkit that can be used to implement several I/O systems at user level to skip the shared kernel hotspots. In this chapter, we use Polytropon to build the Danaus client architecture in order to let each tenant access the container root and application filesystems from network storage through a private host path. We developed a Danaus prototype that integrates per tenant a union filesystem with a Ceph distributed filesystem client and a configurable shared cache. Across different host configurations, workloads and systems, Danaus achieves improved performance stability because it handles I/O with the reserved container resources of a tenant and avoids the intensive kernel locking. It offers offers up to 14.4x higher throughput than a popular kernel-based client under conditions of I/O contention, and in comparison to a FUSE-based user-level client reduces by 14.2x the time to start 256 high-performance webservers. Danaus also serves 32 tenants with RocksDB over network storage with 7.2-14x lower latency than kernel systems, and reduces up to 2.9x the timespan to execute source-code processing in the containers of 32 tenants.

6.1 Overview

Below, we describe the critical parts of Danaus and justify our choices over alternatives. Our presentation includes the goals and principles, the basic structure, the cache, the consistency and the interprocess communication.

In the design of our system, we set the following goals:

1. **Compatibility** Provide native container access to scalable persistent storage through a backward-compatible POSIX-like interface (e.g., open, fork, exec).

2. **Isolation** Improve the performance isolation and the fault containment of data-intensive tenants colocated on the same client machine.

3. **Efficiency** Serve the root and application filesystems of containers through effective utilization of the processor, memory and storage resources.

4. **Flexibility** Enable flexible tenant configuration of the sharing and caching policies (e.g., files, consistency).

A *container pool* (or *pool*) comprises the application and service containers of a tenant on a host. We manage the colocated container pools with a container engine (§ 5.3) per host deployed by the cloud provider. We store the container *root* and *application* filesystems directly on the shared distributed filesystem. We focus on the file-based network storage for the following two reasons: first it allows flexible data sharing among the hosts, and second it avoids the multilayer virtualization overheads of block-based storage [236, 237]. The *storage backend* consists of the server nodes that store the data and metadata of the distributed filesystem. Our architecture can also support clients of block-based network storage but we leave for future work the exploration of this direction.

The Danaus architecture integrates the collection of filesystem functionalities that serve the container pool of a tenant. The *backend client* is the distributed filesystem client that the containers use to access the storage backend. The *union filesystem*

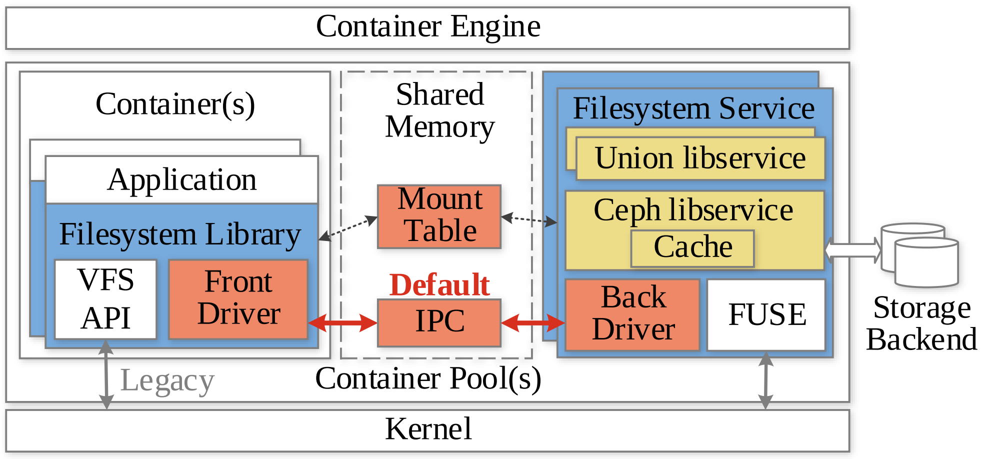


Figure 6.1: **Architecture of the Danaus user-level client.** The color shades illustrate the new components of Danaus and the white background the components of existing systems.

offers I/O deduplication across the container clones or dataset replicas. The *cache* provides caching over the root and application filesystems with configurable sharing and consistency semantics.

The Danaus architecture consists of the *filesystem library* linked to each application, the *filesystem services* that handle the storage I/O, and the interprocess communication that connects the applications with the services (Figure 6.1). The *front driver* of the filesystem library passes the incoming requests at user level to the *back driver* of a filesystem service over shared memory. A *filesystem service* is a standalone user-level process that runs the *filesystem instances* mounted to the containers. A filesystem instance is implemented as a stack of libservices. A libservice is a user-level storage subsystem accessed through a POSIX-like interface. Our prototype implementation currently supports the *union libservice* of the union filesystem and the *client libservice* of the backend client. The union libservice invokes directly the client libservice to serve the branch requests without extra context switching or data copying.

6.2 Client Cache

We faced the dilemma of placing the client cache at either the filesystem library or the filesystem service. We noticed that the library would complicate the coordination of the cache sharing across the container processes of the tenant. In contrast, the filesystem service would run a natively shared cache inside the union filesystem or the backend client. A third possibility could be to run the cache as a distinct libservice above the backend client. We observed that the backend client already operates its own local cache with consistency to the storage backend. Thus, we decided to operate the client cache inside the backend client at user level and run the union deduplication without cache on top of the backend client (Figure 6.1). The union filesystem does not create separate file inodes because it interacts with the backend client through function calls at the file level. The backend client recognizes a shared file from its pathname and holds in the client cache a single copy of each shared branch.

6.3 Consistency

The client cache is the first stateful part in the I/O path from the application to the storage backend. When an application write returns, it has certainly reached the client cache and probably the storage backend. Thus, it will be visible from a subsequent read to the same backend client. The filesystem consistency policy propagates the write to other backend clients at the same or different host. The synchronous requests are served in thread program order by the filesystem service. Instead, the concurrent or asynchronous requests can meet specific ordering requirements through application-level synchronization. At a client crash, the loss of cached state depends on the consistency policy of the distributed filesystem. The application will need to repeat the request if is not notified for completion, or the request is lost during the crash.

6.3.1 Interprocess Communication

The filesystem library communicates with the filesystem service using the Polytropon IPC mechanism that relies on request queues and request buffers. The processor hardware typically organizes the cores in groups with shared same-level cache (e.g., pair over L2). Accordingly, we maintain a separate fixed-size request queue per core group to facilitate the efficient communication between the application container and the filesystem service running on the same group. Each queue is concurrently accessed by multiple producer and consumer threads respectively of the front and back driver. Each queue entry contains a request descriptor and a state field tracking whether the entry is currently empty, under modification, or carrying a valid request. The request descriptor contains a call identifier, a fixed vector of small arguments, and a pointer to a request buffer. This is a data structure per application thread that the front and back driver use to exchange large data items over shared memory.

6.4 Prototype Implementation

We use the filesystem library, the filesystem service, and the IPC components of the Polytropon toolkit to construct a Danaus prototype. We created two libservices in C code based in part on the libcephfs client of Ceph (v10.2.7) [24] and the unionfs-fuse filesystem [238]. The total development effort required the addition of 1295 new lines of code, modification of 2115, and removal of 1245. Table 6.1 summarizes our implementation effort.

Implementation effort of the Danaus prototype

Component	Modified	New	Removed
Cephfs libservice	5	362	17
Unionfs libservice	2110	933	1228
Total	**2115**	**1295**	**1245**

Table 6.1: Implementation size measured in uncommented lines of code, not including white spaces. We measured the lines of code using the cloc [233] utility.

6.4.1 Client libservice

The client libservice is based on the library-based Ceph client (libcephfs). The libcephfs provides direct access to the Ceph filesystem without passing though the VFS.

The client has an integrated object cache to cache recently used data. In most cases, the client refers to the object cache when reading from, or writing to a file. The object cache is organized as a least recently used list of buffers. Each buffer contains the cached data and its state (e.g., clean or dirty). A threshold parameter (client_oc_size) controls the total amount of data that can reside in the object cache.

To handle a read request, the client first looks in the object cache. If the data is not already in the cache, a new entry is added and filled with the data read from the storage servers. Otherwise, the client uses the cached data to satisfy the read request. Before replying to the read request, the client checks the current size of the object cache. If its size has exceeded the client_oc_size threshold, it trims the cache by removing the data that has been expired according to the least recently used policy.

Danaus Object Cache Parameters

Linux		User-level ceph client	
Parameter	**Default**	**Parameter**	**Default**
dirty_background_bytes	0	client_oc_target_dirty	8MB
dirty_background_ratio	10%	-	-
dirty_bytes	0	client_oc_max_dirty	100MB
dirty_ratio	20%	-	-
dirty_expire_centicecs	30s	client_oc_max_dirty_age	5s
dirty_writeback_centicecs	5s	(hard-coded timeout)	1s
-	-	client_oc_size	200MB
-	-	client_oc_max_objects	1000

Table 6.2: The parameters of the object cache [239] and their counterparts in Linux page cache [240].

Similarly, to handle a write request, the client verifies whether the corresponding data is already included in the object cache. If not, a new entry is added to the cache and filled with the data to be written on the storage servers. Otherwise, the cached data is modified in place. In both cases the cached data is marked as dirty. The I/O data transfer does not start immediately, but it is delayed for a while in order to give the chance to application processes to further modify the data to be written. Before replying to the write request, the client checks three thresholds. First, it checks whether the amount of dirty data in the object cache has exceeded the maximum number of dirty bytes (client_oc_max_dirty) and the maximum number of cached objects (client_oc_max_objects). If one or both of these thresholds has been exceeded, the client triggers the write-back of dirty data, and blocks until the amount of dirty data and the number of cached objects drop below their maximum thresholds. Second, it checks whether the amount of dirty data has exceeded the client_oc_target_dirty threshold. In this case, the client triggers the write-back of dirty data and returns the write result to the application process that performed the request without blocking. Third, it checks the current size of the object cache and if it has exceeded the client_oc_size threshold, it trims the cache like in the read case.

A single flusher thread checks the object cache for dirty data and writes it to the storage servers. The flusher thread continuously checks whether the amount of dirty data has exceeded the client_oc_target_dirty threshold. In this case, the flusher thread transfers the excess amount of data to the storage servers. Otherwise, it checks the object cache for dirty data whose age has been exceeded a maximum threshold in seconds (client_oc_max_dirty_age) and transfers it to the storage servers. Finally, it sleeps for a hard-coded timeout period (1s). It wakes up for the next iteration either if the timeout has expired or if the client triggered the write-back.

In table 6.2 we summarize the parameters of the object cache and compare them with the parameters of the Linux page cache. Note that the dirty_bytes and the dirty_background_bytes parameters of the Linux page cache are the counterparts of dirty_ratio and dirty_background_ratio. The former are expressed in bytes, while the later are expressed in percentiles.

6.4.2 Union libservice

The union libservice is responsible to provide a union view of the filesystem that is exported by the client libservice. It is based on the unionfs-fuse [238]. The unionfs-fuse is a user-level implementation of the in-kernel unionfs filesystem [28]. Its implementation is based on the high-level API of FUSE.

The union combines two or more directories together to produce a single merged directory. For each directory, the union maintains a *branch* in-memory data structure that contains the filesystem path of the directory, its access mode (read-only, or read write) and its file descriptor. Note that the union keeps the directory of each branch opened to prevent accidental unmounts of the branch path. The union maintains its branches in an in-memory table, called the *branch table*. It also maintains another in-memory data structure, the *union instance*. The union instance maintains the branch table along with the number of branches and characterizes the union view that the union exports.

The branch table is always processed in ascending order based on its index. The top branch in the table is used in writable mode, while the other branches below it are used in read-only mode. The directory entries of a branch hide the entries with the same name of all the other branches bellow it. In fact, when the union searches for a file, it traverses the branch table from top to bottom. Thus, the writable branch is always looked up first. When a file on a read-only branch is modified, it is copied up to the top writable branch and modified there. The copy up operation is triggered when the file is opened with write permissions. However, when a file is copied up to the writable branch in a directory that exists only on its origin branch, the entire directory path to that file is created on the writable branch, before the copy up operation. Another feature is the deletion of an object from a branch. This is implemented with whiteout directory entries. A whiteout is a directory entry that covers up all entries of a particular name from lower branches.

We implemented the union libservice by transforming the unionfs-fuse standalone filesystem to a library whose I/O functions take a filesystem instance as parameter and invoke a libcephfs client libservice rather than a local filesystem. We cleared the codebase of unionfs-fuse from global variable dependencies that maintain filesystem state (e.g., the uopt variable) and replaced the FUSE API with the libservices API.

6.5 Analysis

In the present section, we qualitatively analyze how the design and implementation of Danaus realize our goals (§ 6.1).

Compatibility The containers mount the root and application filesystems of Danaus from a shared distributed filesystem. Container applications can access the Danaus filesystems through a POSIX-like interface without source-code modifications or re-linkage, by taking advantage of the dual interface.

Isolation A container pool obtains a set of namespaces potentially distinct from those of the host [157, 241]. Correspondingly, a filesystem service runs on the hardware resources of the container pool rather than those of the shared kernel. Through the user-level execution, Danaus improves the multitenant I/O isolation and reduces the contention at processor cores, kernel locks and memory caches. The host kernel is a single point of failure with attack surface inherited to all the colocated containers. Yet, the execution of the filesystem and cache at user level involves thinner kernel interface and results into smaller trusted computing base [106]. For fault containment, Danaus decomposes the filesystem into separate user-level services with possibly distinct implementation per pool rather than fixed by the host kernel. A failed filesystem service affects the processes of a single pool but not the system kernel or the host root filesystem.

Efficiency At the server side, the *shared* network filesystem naturally prevents the storage space duplication of shared application and system files. At the host side, the client cache serves the shared files without duplication of memory space and network bandwidth. The integration of the union filesystem with the backend client prevents resource duplication and page-cache crossing for their interaction. The union filesystem deduplicates the memory space and bandwidth when serving common I/O requests across cloned containers. The user-level interaction of the application with the filesystem service avoids costly memory copies and mode switches through the kernel. In kernel-based filesystems, a typical call enters the kernel once or multiple times. On the contrary, Danaus only involves the kernel in network processing and thread scheduling in the common case.

Flexibility A tenant uses a Danaus backend client on a host to let the containers share files or branches in read-only or writable mode. A tenant can use *multiple* filesystem services with distinct settings in resource naming, memory reservation, page replacement (e.g., swappiness) or data consistency (e.g., writeback) [51]. Multiple tenants can collaborate through a shared pool. Casual administration tasks (e.g., software updates, malware scanning, container migration) can conveniently run centrally through the backend storage rather than separately inside the individual containers.

6.6 Experimental Evaluation

In this section we experimentally quantify the improved isolation, high performance and low resource consumption of Danaus in comparison with representative systems using production applications and microbenchmarks. More specifically, we quantify the ability of different filesystem clients to:

1. Prevent I/O contention between homogeneous and heterogeneous resource-demanding workloads (§6.6.2).
2. Serve the multitenant host I/O over network storage at high performance (§6.6.3).
3. Achieve high performance at low resource consumption across several microbenchmarks and real-world applications (§6.6.4).
4. Restrain a failure inside a pool from affecting the operation of the other pools on the same host (§6.6.5).

6.6.1 Experimentation Environment

Machines The client and server are two identical x86-64 machines of 4 sockets clocked at 2.4GHz. Each machine has four 16-core processors (AMD Opteron 6378), 256GB RAM, and a 20Gbps bonded connection to a 24-port 10GbE switch. A core pair shares 64KB L1 data cache and 2MB L2 cache. Each machine has 6 local disks (125-204MB/s) with the root system installed on 2 of them (in RAID1). Both machines run Debian 9 Linux (kernel v4.9). We repeated several of our experiments on a more recent Linux kernel (v5.4.0) but obtained similar results. The client machine runs containers configured with the Linux cgroup (v1,v2) and namespaces. The server machine is organized into 8 nodes, each of 8 cores and 32GB RAM. We use one node to run the host system. The rest of the nodes are configured as 7 VMs (Xen v4.8) running a Ceph (v10.2.7) cluster of 6 OSDs (object storage devices) and 1 MDS (metadata server). On each OSD we use 8GB RAM for main memory and the remaining 24GB as ramdisk with XFS for the fast storage of the OSD data and journal.

Filesystems We compare Danaus (D) to several combinations of union filesystems and Ceph clients. As representative mature system, we examine the kernel-based Ceph client (CephFS). We run it either standalone (K) or below the kernel-based AUFS (K/K), with the page cache used by both the union and client. As user-level baseline, we use the FUSE-based Ceph client (ceph-fuse) using the page cache (FP) or bypassing it (F). Over the FUSE-based client, we optionally run the FUSE-based Unionfs (unionfs-fuse), with the page cache used by both the union and client (FP/FP) or none (F/F). F/F occupies the least memory space because the FUSE-based Ceph

Summary of configurations

Symbol	Union Filesystem		Backend Client	
	System	*Cache*	*System*	*Cache*
D	Danaus (opt.)		Danaus	ULcC
K			CephFS	PagC
F			ceph-fuse	ULcC
FP			ceph-fuse	ULcC+PagC
K/K	AUFS	PagC	CephFS	PagC
F/K	unionfs-fuse		CephFS	PagC
F/F	unionfs-fuse		ceph-fuse	UlcC
FP/FP	unionfs-fuse		ceph-fuse	UlcC+PagC

Table 6.3: System components of the examined clients. UlcC: user-level client cache; PagC: system kernel page cache.

client only uses the user-level object cache, and neither the union nor the client uses the kernel page cache. Finally, we examine without cache the FUSE-based Unionfs running over CephFS with page cache (F/K). In FUSE, we enable several known optimizations from prior research [48]; in order to bypass the page cache (in F, F/K or F/F), we use the direct I/O mount option. Table 6.3 summarizes these configurations.

Workloads We use the Filebench [206] Singlestreamwrite and Singlestreamread (briefly Seqwrite and Seqread) micro-workloads to generate sequential write and read; the Filebench Fileserver (FLS) to emulate simple fileserver read/write I/O; and the Filebench Webserver (WBS) for local read-intensive I/O [206]. We use the RandomIO of Stress-ng (RND) to generate local random read/write I/O [207]; the CPU benchmark of Sysbench (SSB) for cpu-intensive processing [242]; the NOP application to measure the startup time of containers that perform a no-op operation; the Lighttpd [243] standards-compliant high-performance web server for read-intensive I/O at container startup or under the wrk HTTP benchmark [244]; the RocksDB persistent key-value store for writes and out-of-core reads [208]; the GAP Benchmark Suite [245] (v1.2) for graph processing; the diff command to compare the difference between two source code trees; the git revision control system to perform source code management tasks; our own file append (Fileappend) and read (Fileread) for sequential write and read of a single file with low metadata activity. We run each workload instance on a separate container with basic Debian 9 Linux.

Methodology The comparison of different systems on the same platform provides crucial intuition about the relative benefits and potentials of Danaus. The kernel-based Ceph, the user-level libcephfs and the FUSE-based ceph-fuse are functionally similar but considerably different implementations of the Ceph client. AUFS is a rewrite of the kernel-based Unionfs for improved reliability and performance, while the unionfs-fuse is a FUSE-based version of Unionfs [246, 28, 238]. We keep the default settings of 5s for the expire interval and 1s for the writeback interval in dirty page flushing. To prevent out-of-memory termination, the size of user-level client cache is set to 50% of the pool memory, unless specified differently. We set the max dirty bytes to 50% of the pool RAM in kernel-based Ceph, and to 50% of the client cache in Danaus and FUSE-based Ceph. We repeat the experiments as needed (up to 10 times) to get the half-length of the 95% confidence interval of the primary metric (e.g., throughput) within 5% of the measured average. Although we mostly show the latency, throughput, or timespan, we reached similar conclusions from other gathered metrics, including tail statistics.

6.6.2 Isolation

We examine the performance interference among 1 or 7 Fileserver (FLS) instances and another workload that is either I/O or cpu-bound. Each workload instance runs in a dedicated *container pool* of 2 cores and 8GB RAM. The workloads run at the client machine with the number of enabled cores (4-16) twice the number of running instances (2-8). We run FLS with 5MB mean file size, 1000 files and 120s duration. The dataset is stored on Ceph (without union) through Danaus (D) or the kernel CephFS client (K). Danaus mounts the root filesystem of a container using a private filesystem service with 5GB client cache to hold the entire dataset. Table 6.4 summarizes the workloads that we use in this section.

Figure 6.2a expands Figure 3.4a with results from FLS running over Danaus to prevent the performance variability of the kernel. RandomIO (RND) performs random 512B read/write with readahead using 2 threads. It accesses a 1GB file stored through ext4 on 4 local disks (in RAID0). We run 1 or 7 FLS instances on D or K alone or in parallel to 1 RND instance. From the bar chart, 1FLS/D has 41%

Summary of Workload Symbols

Symbol	Description
FLS	Fileserver (Filebench) on Ceph
RND	Random I/O with readahead (stress-ng) on ext4/RAID0
SSB	CPU benchmark (Sysbench)
WBS	Webserver (Filebench) on ext4/RAID0
1FLS/D	1 x Fileserver on user-level Danaus/Ceph cluster
7FLS/D	7 x Fileserver on user-level Danaus/Ceph cluster
1FLS/K	1 x Fileserver on kernel CephFS/Ceph cluster
7FLS/K	7 x Fileserver on kernel CephFS/Ceph cluster
X+Y	X next to Y, X=(1|7)FLS/(D|K), Y=(RND|SSB|WBS)

Table 6.4: Workloads of isolation experiments.

lower throughput than 1FLS/K. We remind that the execution of RND causes the FLS throughput to drop 7.4x in 1FLS/K+1RND and 16.5x in 7FLS/K+1RND. On the contrary, it only drops 16% in 1FLS/D+1RND and it is even slightly faster in 7FLS/D+1RND. Put differently, D is faster than K by 3.7x-14.4x when FLS is colocated with RND. Danaus avoids the pressure from RND by handling the FLS I/O with the cores of the pool running the respective FLS instance. This is confirmed by the line chart showing that the cores of the RND pool are utilized less than 2.5% when 1FLS/D or 7FLS/D run alone. Additionally, Danaus reduces the kernel lock contention by running a separate user-level filesystem service per container. Moreover, the core and lock contention together cause the 7FLS/K+1RND to flush 32% fewer dirty pages than 7FLS/K running alone.

In particular, our kernel profiling showed that 1FLS/K+1RND and 7FLS/K+1RND are remarkably delayed at the *i_mutex_key* kernel lock of the superblock struct and the *rlock* kernel lock of the Ceph Inode struct. Table 6.5 shows the highest contended kernel locks, while running the fileserver workload on kernel-level Ceph and Danaus. The highest contended kernel lock while using the kernel-level Ceph client is the *i_mutex_key* of the superblock object. Its total waiting time increases up to 7.8x while running 7FLS/K+1RND in comparison to 1FLS/K. In contrast, the top four contended locks while using Danaus relate to memory management operations (e.g., *mmap_sem*) and networking (e.g., *slock-AF_INET*).

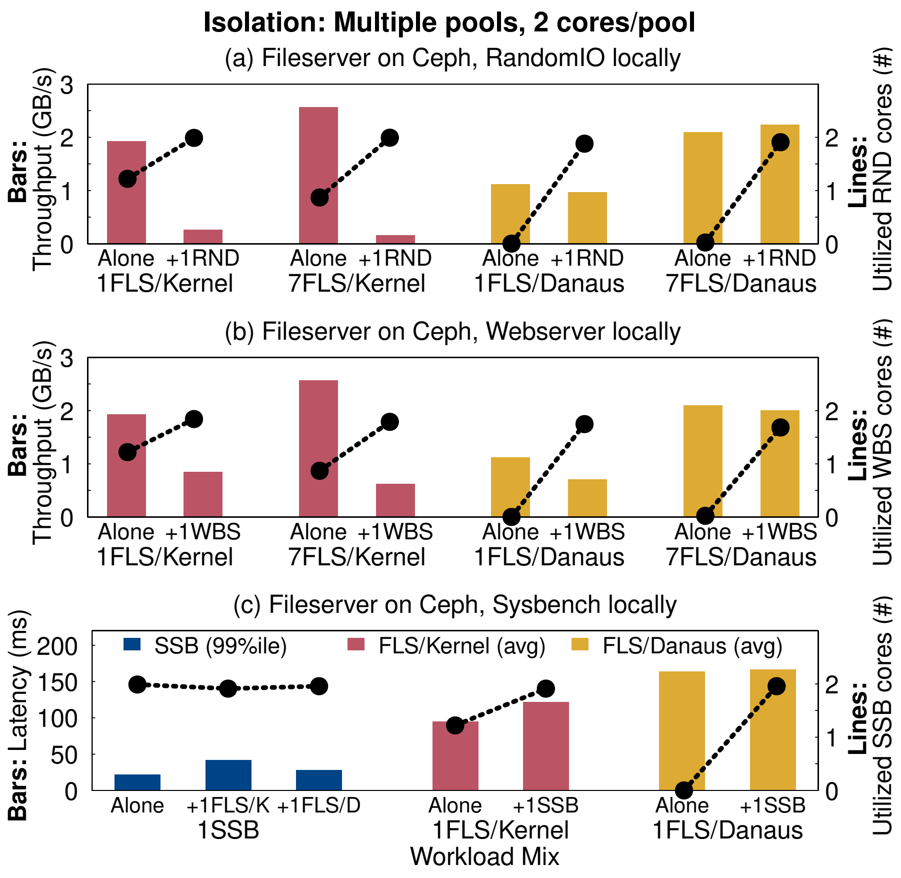


Figure 6.2: **Host contention.** (a) Fileserver throughput (higher is better) and RandomIO core utilization. (b) Fileserver throughput and Webserver core utilization. (c) Sysbench or Fileserver latency (lower is better), and Sysbench core utilization.

We obtained similar results by running FLS next to the Webserver benchmark (WBS). We configure WBS with 50 threads and 200K files of 16KB mean size on ext4 over 4 local disks (in RAID0). We run 1 or 7 instances of FLS over K or D alone or in parallel to an instance of WBS. From the bar chart of Figure 6.2b, the concurrent execution of WBS reduces the kernel FLS throughput by 2.3x in 1FLS/K+1WBS and 4.2x in 7FLS/K+1WBS. Although 7FLS/D alone is 18% slower than 7FLS/K, FLS in 7FLS/D+1WBS is 3.2x faster than 7FLS/K+1WBS. With the WBS inactive, 1FLS/K and 7FLS/K utilize 87-122% the cores of WBS, while 1FLS/D and 7FLS/D only 0.5-2.5% (lines of Figure 6.2b). As with RND, we conclude that the kernel FLS performance drops when WBS is running because the kernel cannot utilize any more the cores of WBS to serve FLS.

Danaus also limits the latency increase (lower is better) across colocated heterogeneous workloads. We consider the Sysbench cpu benchmark (SSB) with 2 threads

Kernel lock contention

Kernel Lock	Wait Time Total (us) x1000			
	1FLS/K	1FLS/K+RND	7FLS/K	7FLS/K+RND
&sb->s_type->i_mutex_key#2-W	651061.9	627599.5	3688702.8	4141121.0
&sb->s_type->i_mutex_key#2/1	935234.3	982964.9	4144350.2	5187051.2
&con->mutex	17190.0	17705.7	49990.9	57928.3
&(&ci->i_ceph_lock)->rlock	4.5	2.1	7.6	6.7
	1FLS/D	1FLS/D+RND	7FLS/D	7FLS/D+RND
&mm->mmap_sem-R	18347.7	16478.7	105019.4	16478.7
slock-AF_INET	22.2	59.6	153.6	59.6
&rq->lock	153.9	199.2	781.1	199.2
&(&sch->q.lock)->rlock	1.0	0.7	8.7	0.7

Table 6.5: Top four kernel locks with highest total wait time.

in prime number calculation using 64-bit integers. In the bar chart of Figure 6.2c, we show the 99%ile of the SSB latency and the average FLS latency over Ceph. We run one FLS instance over K or D, alone or in parallel to one SSB instance. The workload interference raises the respective SSB and FLS latency by 93% and 28% in 1FLS/K+1SSB but only 27% and 2% in 1FLS/D+1SSB (3.4x and 14x less). When FLS runs alone, K achieves lower latency than K because it also utilizes the reserved SSB cores (Figure 6.2c line chart). In comparison to RND and WBS, SSB affects less intensely the 1FLS/K because it only runs user-level calculations.

In summary, Danaus achieves throughput and latency stability across competing workloads because it serves I/O with less hardware and kernel contention. On the contrary, the kernel-based client causes performance drop up to 16.5x.

6.6.3 Performance and Efficiency with Applications

We consider I/O-bound applications over Danaus or different combinations of FUSE and the kernel. We run a workload per container and report the measured performance and consumed resources. Each container runs either independently on a distinct system image (2.7GB), or cloned over a shared read-only lower branch and a private writable upper branch.

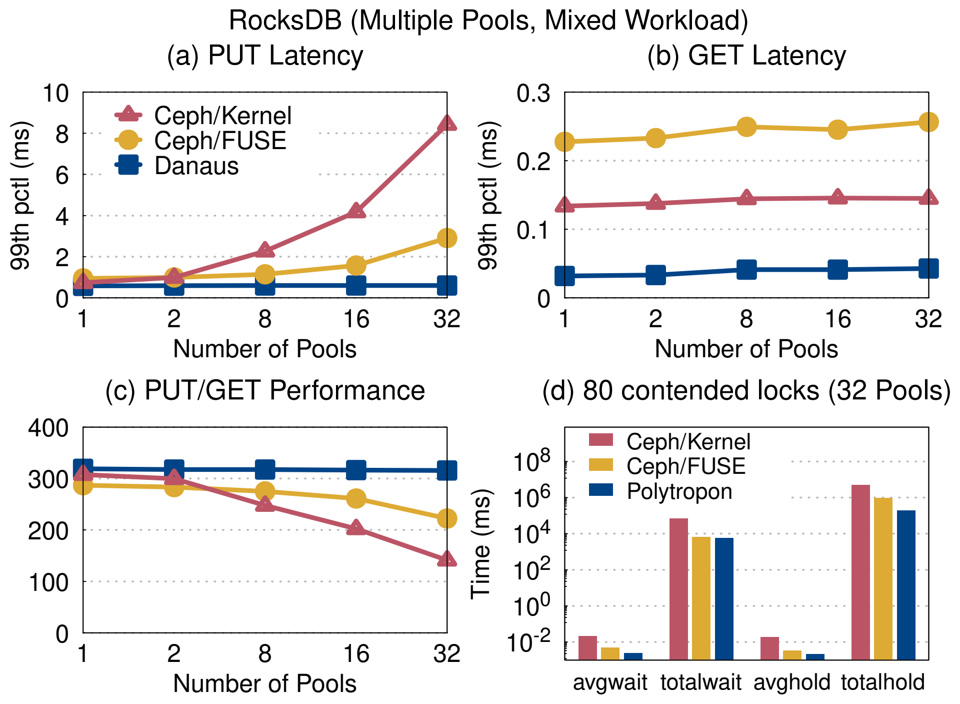


Figure 6.3: **RocksDB in scaleout setting.** The RocksDB storage engine running over three alternative Ceph clients. We measure the 99%ile of put (a) and get (b) latency, the average put/get throughput per pool (c), and the average/total lock wait/hold time (d).

RocksDB We first examine the scaleout performance of serving multiple tenants with separate pools. We revisit the RocksDB [208] experiment of Figure 3.5 by expanding the plots with a third line for comparison with Danaus (Figure 6.3). Danaus allows each pool to access a container root filesystem from Ceph through a separate instance.

We run the RocksDB storage engine independently in 32 container pools of the same machine. Each pool is configured with one container over 2 cores and 8GB RAM. The container runs RocksDB with 64MB memory buffer and 2 compaction threads. The pool uses a dedicated Ceph client to mount a private root filesystem holding the files of both the container and RocksDB. The put workload uses 1 thread to insert 1GB random pairs of 9B key and 128KB value.

We start with the 99%ile of the put and get latencies across the three clients (the results from average statistics were similar). At 1-32 pools, the put latency of Danaus remains almost flat within 3% of 580μs (Figure 6.3a). Instead, the put latency of FUSE and the kernel is higher up to 1.6-4.8x and 1.3-14x over Danaus. From

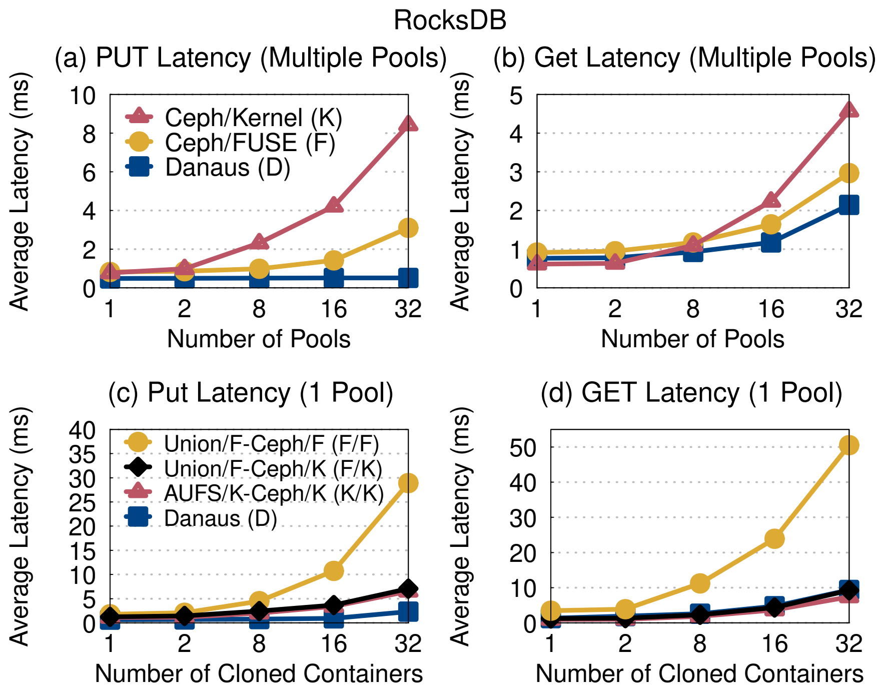


Figure 6.4: **RocksDB in both scaleout and scaleup settings.** Average latency of RocksDB put and get in scaleout (a,b) and scaleup (c,d) settings of container pools.

Figure 6.3b, the get latency of Danaus is consistently lower than the kernel and FUSE up to 4.2x and 7.2x, respectively. In Figure 6.3c, the flat throughput per pool confirms the linear scalability of Danaus. In contrast, the throughput of FUSE and the kernel at 32 pools is up to 23% and 54% lower than that of 1 pool.

Figure 6.3d explains the above results through the evidence that the FUSE and kernel clients face kernel lock wait or hold time up to an order of magnitude higher than Danaus.

We also measured the average put latency of RocksDB over the D, F, or K client for 1-32 pools. In Figure 6.4a, D is faster than F and K up to 5.9x and 16.2x (32 pools). A main reason is the intense kernel contention of F and K demonstrated by total kernel lock wait time up to 5x and 152x higher over D (1 pool). In another experiment, we run an out-of-core read-intensive workload per pool. Each container populates RocksDB with 8GB using random 128KB puts before reading back 8GB with random gets. In Figure 6.4b, D is faster than F and K up to 1.4x and 2.2x (32 pools). Correspondingly, the total kernel lock wait time of F and K is up to 2.8x and 17.8x that of D (32 pools). Moreover, D reduces the cpu activity (incl. IO wait) and memory consumption of K up to 9.5x and 2.6x. Danaus reduces the lock contention

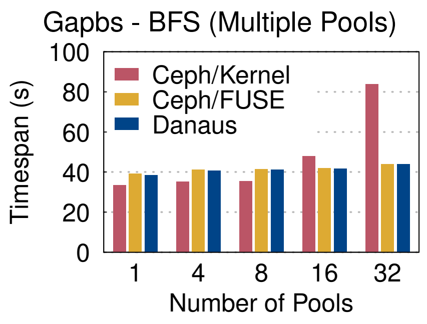


Figure 6.5: **Gapbs BFS workload.** Timespan to start and run containers with the Gapbs workload.

and resource consumption by running and accessing at user level a distinct client per container.

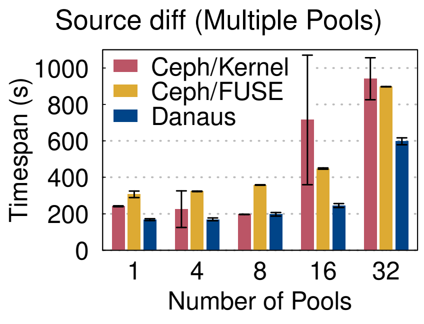
In summary, both the FUSE and kernel clients involve the shared data structures of the system kernel in the I/O transfer or handling even though each pool is served by a separate FUSE process or kernel mount.

We also explored a scaleup setting running multiple cloned containers in a single pool. Each container runs a private RocksDB instance. A container mounts its root filesystem through a private filesystem instance consisting of a distinct union filesystem and a shared Ceph client. The Ceph client lets the containers share the lower read-only layer of their root filesystem. We measured the RocksDB latency across D, F/F, F/K and K/K (Table 6.3). With respect to put latency, D is faster than F/F, F/K and K/K up to 12.6x, 3.9x and 3.6x, respectively (Figure 6.4c). In contrast, the get workload (Figure 6.4d) leads to mixed results with D up to 5.4x faster than F/F at 32 clones but up to 2x slower than K/K at 2 clones.

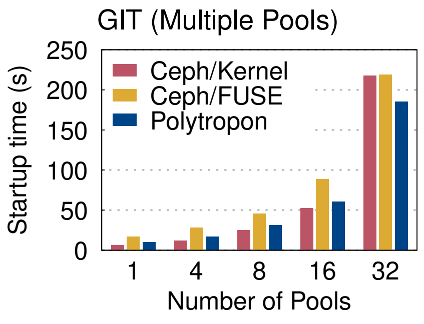
Serving all the clones with a single client in scaleup cancels the scaleout decentralization and concurrency. Still, D achieves substantial latency benefit over the other systems in put, and over F/F in get.

Gapbs We use the GAP benchmark suite (Gapbs [245]) as an application of graph processing. We run the read-intensive Breadth-First Search (BFS) algorithm on a directed graph labeled with the distances of all the roads in the US (1.3GB). Each pool mounts a separate root filesystem from Ceph and executes one container. The BFS algorithm performs calculations on a graph built in memory with the edge lists retrieved from the container root filesystem.

In Figure 6.5, we measure the timespan to initiate several independent pools, run



(a)



(b)

Figure 6.6: **Software development workloads.** Timespan to start and run containers with the (a) diff workload and (b) the git workload.

BFS in each pool container, and wait until they all finish. We observe that FUSE and Danaus keep the timespan in the range 38-44s almost independently of the pool count. Instead, the kernel timespan grows from 33s to 84s (1.9x of 44s) because of the longer I/O time to read the edge lists. From profiling the system kernel, we found the kernel-based client to be slowed down by the wait time on the spin lock of the LRU page lists.

Although several ongoing projects target to improve the system kernel concurrency, Danaus provides separate filesystems per container pool at user level as a more straightforward solution.

Software Development As a software development application, we use the diff command to generate the difference between the Linux kernel v5.4.1 and v5.4.2. The command invokes a mix of the *read*, *stat*, *opendent*, *open*, *close* and *munmap* calls. Each pool consists of one container running the diff over a private repository stored in its root filesystem. Figure 6.6a is a bar chart of the workload timespan with the standard deviation as error bar. At increasing number of pools, FUSE and the kernel take up to 1.9x and 2.9x longer than Danaus. Moreover, the kernel-based client has up to 32.6x higher standard deviation than Danaus.

As a version control application, we use the git system to perform common version control operations on a source code tree. We use up to 32 pools. Each pool consists of one container running a version control workflow that consists of the following commands: 1) *git init*, 2) *git add* 3) *git commit* over a private repository of the rocksDB source code stored in its root filesystem. In Figure 6.6b we show a bar chart of the version control workload timespan. At increasing number of pools, FUSE and kernel

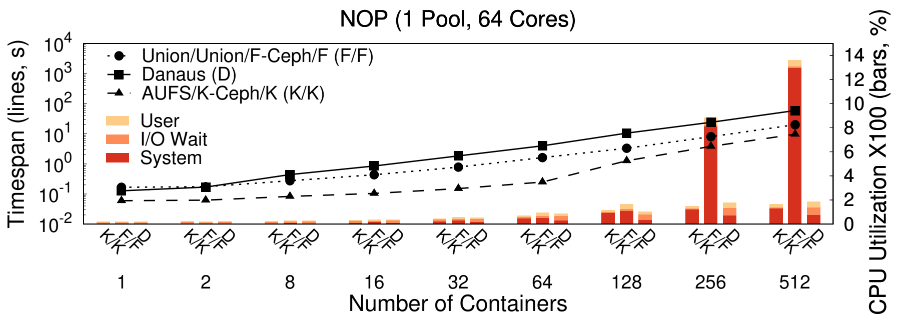


Figure 6.7: **NOP container startup scaleup.** Time and cpu utilization to initiate up to 512 containers in one pool. The containers are cloned from the same system image over union with Ceph.

take up to 1.7x (with 1 pool) and 1.2x (with 32 pools) longer than Danaus respectively.

Our results show that the I/O kernel path of the FUSE-based and kernel-based clients causes higher delays, while the kernel I/O handling causes substantial performance variability.

NOP This experiment explores the scaleup I/O performance in a pool that serves all the containers from a single Ceph client. The embedded barchart in Figure 6.7 shows the cpu utilization in units of fully utilized cores. Both F/F and D implement the union and Ceph filesystems from the same codebase, but D incurs substantially less kernel involvement. Instead, K/K is faster because both AUFS and CephFS are more mature than the user-level libservices of D. In particular, the libcephfs is documented to use coarse-grain locking that limits concurrency.

Therefore, Danaus provides a clear improvement with respect to FUSE in multitenant environments that require user-level flexibility and isolation.

Lighttpd In a single pool, we measure the time to start a number of cloned containers with Lighttpd waiting for requests. The I/O traffic is generated by the *exec* call to start the initial command, the *mmap* calls to fetch the dynamic libraries, and the user-level calls to prepare the application files. In Danaus, this scaleup workload exercises mostly the (legacy) kernel path to read data and less the (default) user-level path to write data. The kernel-based AUFS (K/K) with CephFS (K/K and F/K) lead to higher performance in comparison to the user-level unionfs-fuse with cephfs-fuse (F/F) or libcephfs (D). With respect to the real time to start 1-256 instances, D is up to 8.8x slower than K/K and 2.9x than F/K (Figure 6.8a). From kernel profiling we confirmed that the workload is read-intensive and D uses the (legacy) FUSE path to

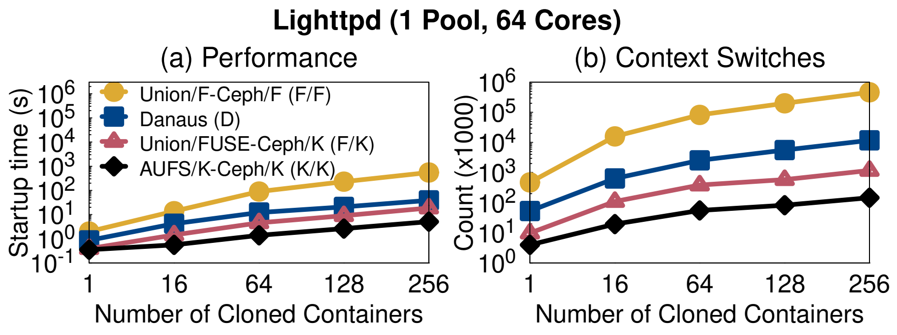


Figure 6.8: **Lighttpd container startup scaleup.** Real time to start 1-256 Lighttpd cloned containers in a single pool over shared client.

fetch the dynamic libraries. In comparison to F/F that uses similar Unionfs and Ceph client code, D reduces considerably the startup time (2.3-14.2x) and cpu activity (up to 10.5x). The improved performance and efficiency of D over F/F is explained in part by the 9-39x fewer context switches (top two lines of Figure 6.8b).

In the examined real-world applications, Danaus achieves lower latency and resource consumption than the other systems in scaleout or put scaleup, and over F/F in get scaleup. In legacy-dominated I/O, the more mature K shortens the startup time, while D is up to 14.2x faster than F/F.

6.6.4 Performance and Efficiency with Microbenchmarks

We measure the performance and resource consumption of two Filebench workloads and a custom-developed one. We consider scaleout and scaleup settings of root filesystems mounted from Ceph network storage through components based on Danaus, FUSE or the kernel.

Scaleout Seqwrite/Seqread We generate sequential writes with Filebench [206] Seqwrite in 1-32 pools. Each pool of 2 cores and 8GB RAM mounts a private root filesystem from Ceph through D, F or K. We configure Seqwrite with 1GB file size, 16 threads, and 120s duration. Seqwrite generates I/O activity in the entire path from the application to the backend servers. In Figure 6.9 (top), the throughput of D and F is up to 2.8x higher than K, while the I/O wait cpu time of K is up to 90x that of F. One reason is that K spends three orders of magnitude more time for kernel lock waiting (most notably *i_mutex_dir_key* and *i_mutex_key*). We also found that K handles I/O with unallocated cores whose number decreases at more pools.

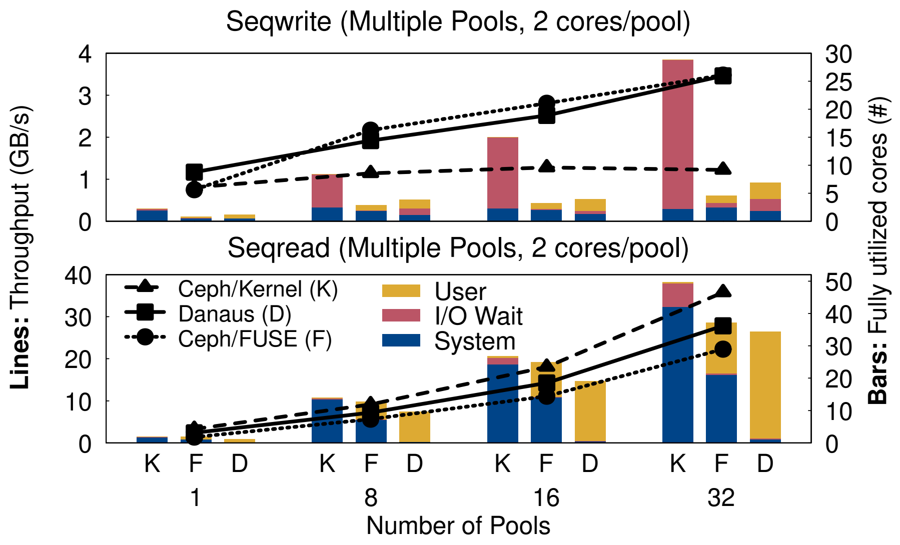


Figure 6.9: **Sequential I/O scaleout.** Performance and core utilization of the Filebench Seqwrite/Seqread at multiple pools.

In a different experiment, we generate sequential reads with Seqread configured similarly to Seqwrite. The workload stresses the local path to the client cache fully holding the accessed file. From Figure 6.9 (bottom), K is faster than D up to 37%, and D faster than F up to 75% (1 pool). Unlike F and K, D predominantly runs at user level and negligibly uses the kernel (bar chart). With user-level profiling, we found that the concurrency of D is limited by *client_lock* [247]. This is a global lock of libcephfs used by D, but not by the kernel locking of K. From preliminary experiments, removing the global lock improves the Danaus concurrency but requires refactoring libcephfs, which is beyond our current scope.

We conclude that at pool scaleout, D is faster than F and K in sequential write, but slower than K in cached sequential read.

Scaleup Seqwrite/Seqread Running each of the K, F, D clients on a single pool of increasing size shows similar trends as above. We configure the pool with 1 to 64 cores, memory size 4GB times the cores, and client cache 4GB. We run Seqwrite with threads twice the pool cores, and duration 90s (from 120s above) to keep the written data within the OSD storage capacity. In Figure 6.10 (top), D has higher data throughput than K up to 12.7x and F up to 2.2x. Also, K at 1 core has low system and usercpu utilization but high I/O wait. At larger pool sizes, D has much higher user cpu utilization over F, and K has much higher system utilization over D and F. From the kernel profiling that we did, we attribute the lower K performance to the

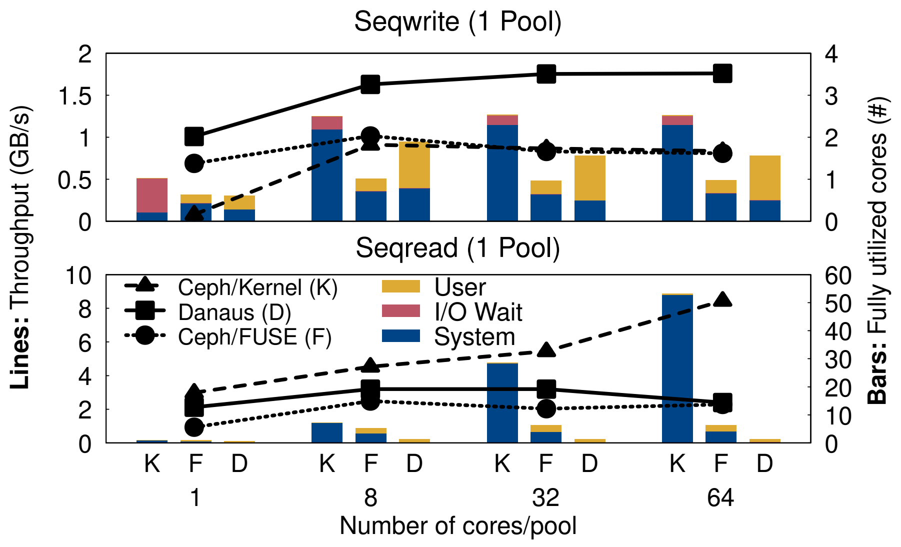


Figure 6.10: **Sequential I/O scaleup.** Performance and core utilization of the Filebench Seqwrite/Seqread at one pool of increasing size.

substantial lock waiting time of K on the i_mutex_dir_key kernel lock used in inode access [248].

In the sequential read workload with cache-resident dataset, K achieves higher performance than D and F in scaleup, as shown in Figure 6.10 (bottom). In Seqread, we set the threads 8 times the pool cores, and the duration 120s. At pool size 1-64 cores, K has 1.4-3.5x higher throughput than D, and D has 5-127% higher throughput than F. The total cpu utilization of K gets up to 45x that of D. T

The performance advantage of K over D in cached read is wider at scaleup (relatively to scaleout) due to the higher lock contention of libcephfs in Danaus on pool size up to 64 cores (instead of pool size 2 in scaleout).

Scaleout Fileserver We experiment with Filebench [206] Fileserver in scaleout of multiple pools. Each pool of 2 cores and 8GB RAM mounts a root filesystem through a private D, F or K client. Fileserver running on Ceph generates a random read/write workload that stresses the entire path to backend storage. We run 1-16 pools on the client host and measure the total throughput of the pools.

From Figure 6.11 (line chart), D achieves 2.7GB/s at 16 pools with an advantage of 1.7x over F at 1 pool and 2.3x over K at 8 pools. The amount of resources consumed by the three clients at the server side is comparable (not shown), but K generates up to 22x higher I/O wait cpu time at the client side (Figure 6.11, bar chart).

Thus, in a random read/write workload at pool scaleout, D achieves substantial perfor-

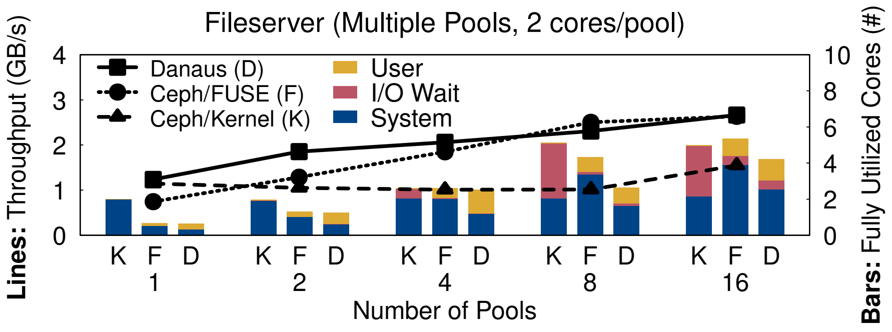


Figure 6.11: **Random I/O scaleout.** Performance and core utilization of the Filebench Fileserver at multiple pools.

mance improvement over K at increasing number of pools, and over F at smaller scales.

Scaleup Fileappend/Fileread We run multiple cloned containers inside a single pool of 64 cores and 200GB memory, leaving 64GB RAM to the host system. The root filesystem of each container is mounted through a private union filesystem and a shared Ceph client to provide a common read-only lower branch. We measure the time to launch the containers, run a high-data/low-metadata workload and wait until they all finish.

Fileappend opens a single 2GB file in `O_WRONLY|O_APPEND` mode, writes 1MB and closes it. The generated I/O is approximately 50/50 read/write because the file is copied to the upper layer by the copy-on-write union. In Figure 6.12a (top), D tends to achieve shorter timespan (lower is better) in comparison to the other systems and especially K/K by up to 46% in 32 containers. In Figure 6.12a (bottom), the maximum memory required by K/K, F/F and D increases linearly with the number of containers, while the page-cache usage by the FUSE-based Ceph client almost doubles it in FP/FP.

In a different experiment, Fileread opens a 2GB file in `O_RDONLY` mode, reads the entire file in 1MB blocks, and closes it. In Figure 6.12b (top), K/K achieves 1.2-4.9x shorter timespan than D but up to 7.4x higher client cpu activity (not shown). In Figure 6.12b (bottom), F/F requires the same memory size as D but with 11-23% longer timespan. Although FP/FP achieves shorter timespan than D, it occupies up to 30x more memory. The excessive memory usage of FP/FP is caused by caching of the union and Ceph client in page cache, and of the Ceph client at user level. Bypassing the page cache at either the FUSE union or the FUSE client keeps the memory usage at least twice that of F/F.

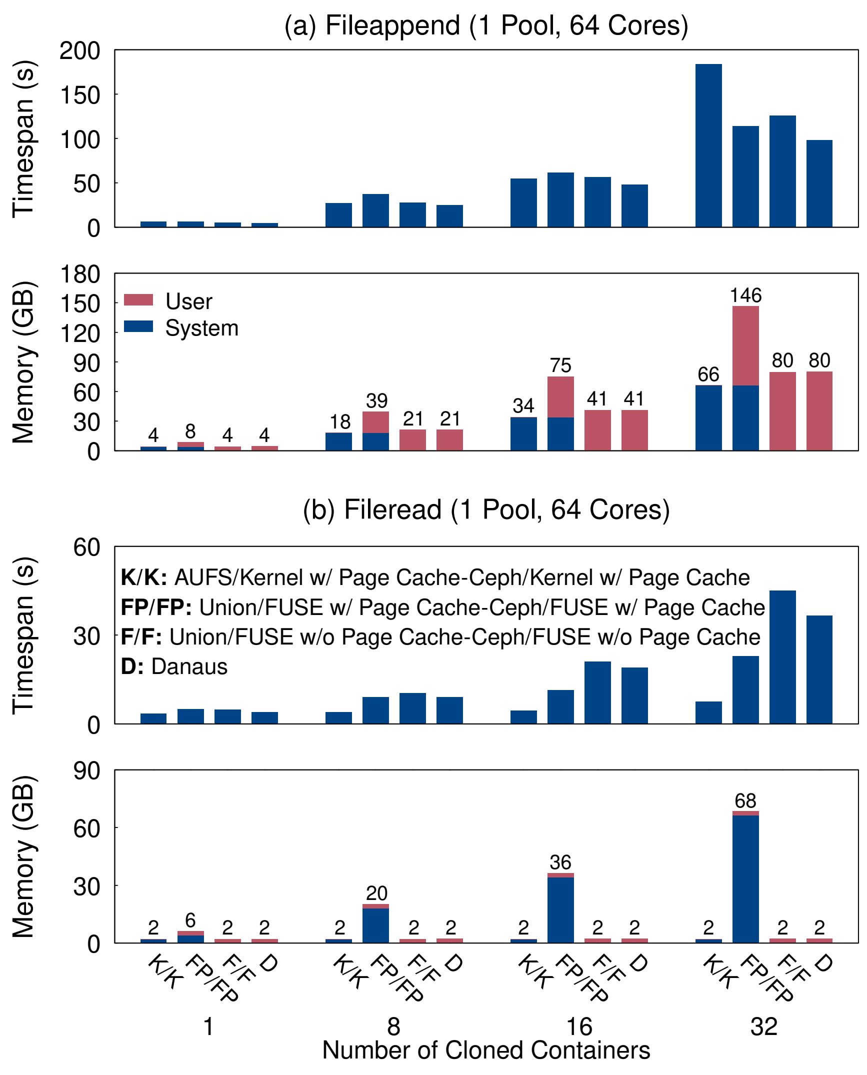


Figure 6.12: **Sequential I/O scaleup with cloned containers.** In a single pool, we measure the timespan (lower is better) and maximum memory to start a number of concurrent containers running the Fileappend (a) or Fileread (b) benchmark, and wait until they all finish.

Therefore, in pool scaleup with cloned containers over union and shared client, Danaus and FUSE are faster than the kernel in mixed read/write, but slower in sequential read.

6.6.5 Fault Tolerance

We use a host with two pools to experimentally demonstrate the ability of the user-level systems to tolerate failures. Each pool runs a Ceph client and a container in a separate network namespace. The container runs the lighttpd web server over Ceph and the wrk benchmark with two threads sending requests to lighttpd. The two pools

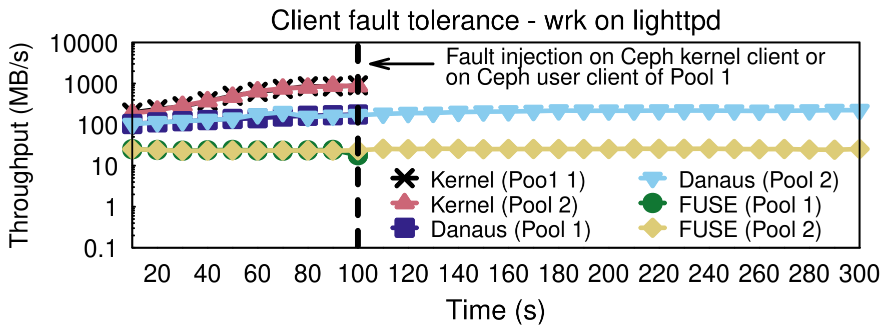


Figure 6.13: **Fault injection.** Lighttpd running in two separate pools with one or both terminated by an injected fault.

run the same type of Ceph client, either D, F or K.

We start the two pools simultaneously and inject a fault to pool 1 at time 100s. Depending on the client, the injected fault causes termination of the filesystem service of Danaus, the Ceph process of FUSE, or the CephFS kernel module. As shown in Figure 6.13, the performance of the surviving pool remains unaffected in the case of F or D. Instead, the fault injection of K crashes the entire host system and terminates the operation of both pools. This behavior is reasonable because D runs a separate Ceph filesystem service and F a separate Ceph process per pool, while K serves both pools with the same kernel module. Although FUSE also runs a kernel module, we only terminate the user-level process serving the pool. Danaus uses the kernel futex synchronization and network stack, but also a FUSE module only for the legacy kernel path with the default IPC handled at user level.

6.7 Summary

This chapter introduces Danaus, a user-level client architecture that provides isolation and efficiency for the container root and application filesystems in a single framework. Danaus serves separately each tenant with a user-level filesystem service that integrates a union filesystem with a distributed filesystem client and local caching. We handle I/O with the private resources of a container pool and avoid the costly locking of the shared kernel.

We experimentally demonstrate that Danaus improves isolation at reduced memory and cpu utilization and even achieves (up to 14.6x) higher performance than a mature kernel-based client, most notably in write-intensive benchmarks and applications.

We conclude that our user-level client architecture improves the isolation and efficiency of container I/O at multitenant hosts over an unmodified stock system kernel.

Chapter 7

Fast Interprocess Communication

Modern software systems (e.g., a microkernel, or a distributed application) comprise a complex collection of independent software components. Such systems can be constructed using two methods: i) implementing each software component as a library and host all the libraries in a single process, or ii) implementing each component as a separate process. The first method provides the best possible performance, as the software components interact with each other using synchronous function calls, data references, and intra-domain data copying. However, it complicates the sharing of software components (e.g., a caching layer) to multiple application processes. It also lacks isolation, because an error on a software component can propagate across components and compromise the whole system. The second method enables multiple processes to share software components, provides strong isolation, but imposes interprocess communication (IPC) overheads as it involves cross-domain copying of requests and data.

In this chapter, we delve into the details of user-level IPC and examine two of its key components, the data structure that is used by producers and consumers for communication and the memory copy operation that is used for data transfer.

7.1 Background

The interprocess communication is a fundamental component of a distributed software system and its performance can determine the overall performance of the application. As a result, it has to be fast and combine low latency with high throughput.

An operating system kernel provides a plethora of mechanisms for interprocess communication, including pipes, sockets, and files. However, kernel-level IPC faces performance issues, because its performance is limited by the cost of invoking the kernel through system calls, address space switching, and inter-domain data copying.

The above issues can be solved by moving the IPC mechanism out of the kernel and implementing it at user level by relying on shared memory in order to enable two different address spaces to communicate.. Communication performance is improved since the kernel invocation can be avoided during the communication between address spaces on the same machine.

Typically, application processes that communicate through IPC are categorized as clients (or producers) and servers (or consumers), where the client requests data and the server responds to client requests. In the following sections we introduce the producer-consumer problem and provide background material related to memory operations.

7.1.1 The Producer-Consumer Problem

The producer-consumer problem commonly appears in a multicore machine when a server process receives requests from application threads and returns back responses. As the Figure 7.1 illustrates, the producers communicate with the consumers through a shared data structure often implemented as a concurrent FIFO queue [227]. The concurrency allows multiple producer and consumer threads to simultaneously access the queue, while the FIFO property guarantees that the consumer removes from the queue the producer request that has waited the longest. The safety of the concurrent queue is ensured by the linearizability property that each queue operation appears as completed at a single time point and the equivalent operation sequence meets the

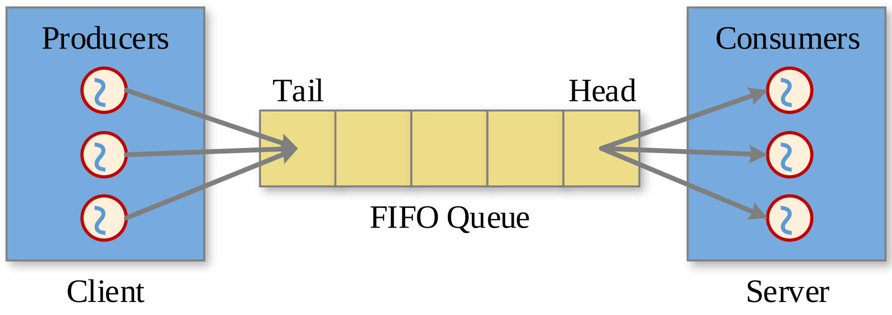


Figure 7.1: **Producer-Consumer communication.** A number of producer threads of a client process place requests on the tail of a FIFO queue. Each consumer thread of a server process extracts a request from the head of the queue and serves it.

properties of an abstract FIFO queue. An enqueuer cannot insert a node to a full queue, a dequeuer cannot remove a node from an empty queue, and each request is removed in FIFO order after it is inserted.

The *enqueue latency* of an item is the time period between the item arrival and the item insertion at the queue. This includes the potential delay of a full queue to free a slot plus the synchronization overhead of the insertion. The *dequeue latency* is the time period between the dequeuer arrival and the item removal. It includes the potential delay of an empty queue to receive an item plus the synchronization overhead of the removal. We collectively call *operation latency* either one of the enqueue and dequeue latencies. The *queueing latency* of an item is the time period between the item insertion and item removal from the queue. The *wait latency* of an item is the time period from the item arrival until the item removal from the queue.

The fast operation at high concurrency is typically achieved by fine-granularity locking or non-blocking synchronization. The respective data structures consist of one or more queues often delayed by frequent locking, memory allocation or livelock. The LCRQ [229] lock-free FIFO algorithm dynamically creates a new queue when the current becomes full, and removes items from the older queues. The wait-free queue [231] connects the enqueuer and dequeuer threads in a linked list and lets the dequeuer threads complete the pending requests. On a data structure consisting of multiple queues, several techniques maintain a relaxation bound on the deviation from the strict FIFO ordering of the item insertion or removal [249, 191, 250, 251, 252].

We aim to minimize the wait latency by balancing the decrease of operation latency and the increase of queueing latency resulting from the relaxed ordering.

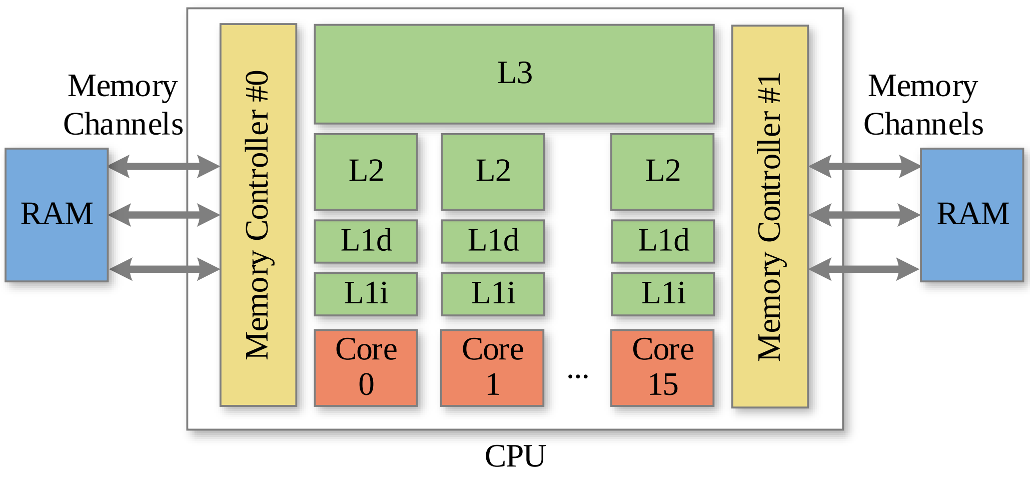


Figure 7.2: **The memory subsystem.** A memory controller is typically integrated on the CPU and communicates with the RAM modules through multiple memory channels. To access data fast, the CPU maintains a cache hierarchy that contains least-recently-used data.

7.1.2 Memory Management

Supporting efficient memory copying operations is also critical for the performance of the interprocess communication mechanism. In this section, we examine the memory subsystem of modern systems which is a key component of a system's architecture. The memory retains system state, holds machine instructions for execution, and data for computation. In modern systems, the memory bandwidth is considered the primary bottleneck that limits the performance of applications [78].

Memory organization An important component of the memory subsystem (Figure 7.2) is the *memory controller* which manages the flow of data going to and from the main memory modules. The memory controller is connected with the memory modules through buses, known as *memory channels*. In modern system architectures the memory controller is integrated into the CPU. The portion of memory that is attached directly to a CPU is known as *local memory* and is accessible by the CPU at full speed. A system interconnect (e.g., crossbars, point-to-point links) connects multiple CPUs together and allows each CPU to have access to *remote memory*, a portion of memory that is attached on another CPU. As a result, all the available system memory is accessible by all the CPUs that are installed on the system. However, memory access time and effective memory bandwidth varies depending on how far away the

CPU that makes the memory access is from the CPU that connects directly with the target memory. Because the memory access times are not uniform, this architecture is known as Non-Uniform memory Architecture (NUMA).

Caching To speed up application memory access, a faster memory is added between a CPU core and the main memory which is known as *cache* (Figure 7.2). The cache is commonly organized into a hierarchy of multiple levels that differ with each other in terms of size and speed. The first level (L1) is the fastest and is divided into a data cache (L1d) which holds least-recently-used data and an instruction cache (L1i) which holds instructions that the CPU must execute. This cache level is typically private to each processor core. The second level (L2) of the cache hierarchy is larger and slower than the first level. It can be private to a processor core or shared across a number of cores. The last level of the cache hierarchy (L3) is the largest and slowest cache and is typically shared among all the cores of the CPU.

By default, all data read or written by a CPU core is stored in the cache. There are two exceptions to this rule: memory regions that cannot be cached, and instructions that allow the developer to deliberately instruct the CPU core to bypass certain caches. When a CPU core needs a data word, it searches the caches first. However, the entries stored in the cache are not single words, but, *lines* of several contiguous words. On modern CPUs the typical size of a cache line (CL) is 64 bytes. As a result, when a CPU core needs a data word, it loads an entire cache line into the L1d. When an instruction modifies memory, the CPU still has to load a cache line first and modify its contents.

Data reference patterns A data reference pattern is classified as *temporal* if the data is used again soon and *non-temporal* if the data is referenced once and not reused in the immediate future [253, 254]. Additionally, it is classified as *spatial* if the data is accessed at adjacent locations and *streaming* if accessed at contiguous addresses. A store is *write-combining* if successive writes to the same cache line (CL) are combined, and *weakly-ordered* if no ordering is preserved with respect to other loads and stores. An operation is *uncacheable* if some aspects of it bypass the core cache. Most recent systems follow a *write-allocate* policy that applies a store to a line already residing in cache, or fetched to it on miss (*read-modify-write*). A store is *not write-allocating* if it updates a CL without prior load (*read-for-ownership* bus request) [255].

Load and store types Following the above definitions, we call *temporal* a load or store executed through the normal cache path, and *non-temporal* otherwise. A non-temporal load transfers an aligned CL through a temporary internal buffer often implemented with *Line Fill Buffers (LFBs)* [254, 256]. A non-temporal store transfers the data from a register to memory with an operation that is write-combining, weakly-ordered, uncacheable, and not write-allocating. The LFBs are located between the L1 Data Cache and L2 to serve write-allocating stores with cache miss and non-temporal loads or stores (e.g., Intel x86). There is a different temporary internal buffer called *Super Queue (SQ)* between L2 and L3 to handle the L2 misses [257].

Prefetching Prefetching moves data from memory to processor in advance of using it [258, 253, 259, 254, 260]. Software prefetching hints through special instructions the data transfer to either the processor cache or a temporary internal buffer without register involvement. Instead, hardware prefetching is activated automatically by the processor. The hardware prefetchers at L1 either fetch the next line when they detect ascending access pattern at recently loaded data (*streaming prefetcher*), or prefetch forward/backward when they detect regular load strides (*instruction pointer-based stride prefetcher*). The hardware prefetchers at L3 (and probably L2) extend a requested CL to a CL pair (*spatial prefetcher*), or to multiple CLs by detection of ascending/descending pattern at L2 (*streamer*).

Memory moves Data moves handle units of 1 byte up to 64 bytes according to the supported instructions and registers. Larger registers do not necessarily lead to higher transfer throughput because the processor dynamically adjusts the operating frequency according to the types of instructions executed across the available cores [259]. Depending on the inclusion policy, the core cache capacity varies between the size of the last-level cache and the total size of the caches at different levels. The memory access pattern affects critically the memory latency. Relevant factors include the contiguity of the accessed addresses over a page, the access concurrency across different pages, the switching between reading and writing (*read-to-write/write-to-read turnaround*), or the reading switching (*read-to-read turnaround*) across different memory ranks [258].

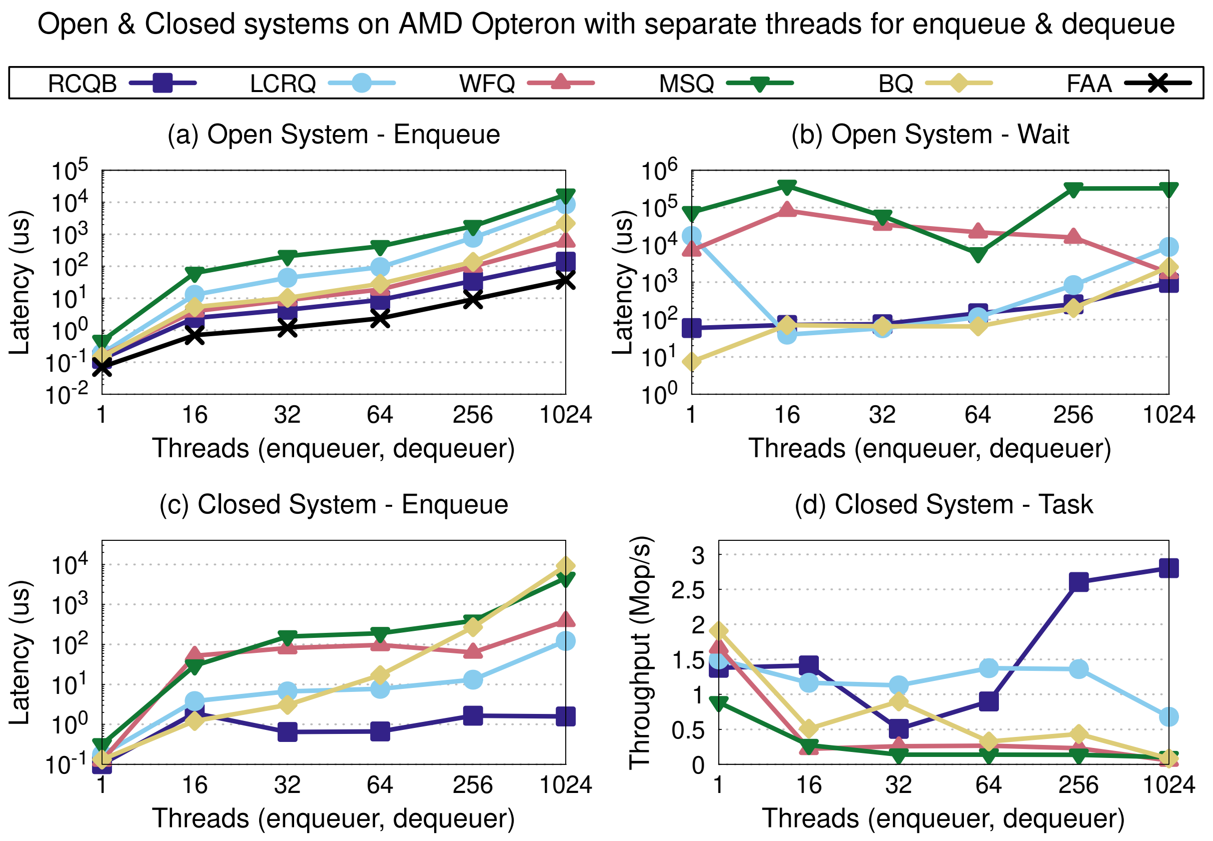


Figure 7.3: **RCQB vs strict FIFO queue algorithms.** This figure compares RCQB with state-of-the-art strict FIFO queue algorithms (#enqrs=#deqrs).

7.2 Problem Definition and Motivation

In this section, we focus on two key components of the interprocess communication mechanism, the queue data structure that is used for request communication, and the memory copy routine that is used for data transfer.

7.2.1 Concurrent Queues

The design and implementation of a practical concurrent queue is challenging because it involves complex tradeoffs between conflicting goals and abstract reasoning about the claimed properties. The enqueue and dequeue operations should be fast for low latency. A queueing discipline (e.g., first-in-first-out, or FIFO) typically removes first the item with the longest time in the queue to minimize waiting and avoid starvation. *However, the time-based ordering of items is an inherently sequential approach that limits concurrency and delays the queue operations.* Indeed, the comparative benefit of the FIFO fairness over the resulting sequentiality delay remains unclear from prior research.

In chapter §5 we introduced the Relaxed Concurrent Queue Blocking algorithm that relaxes operation ordering by splitting queue operations at two stages, a first stage that distributes the operations sequentially across the slots of the queue, and a second stage that lets them complete in parallel and potentially out of FIFO order.

We compare RCQB to previously known algorithms over a 64-core AMD Opteron machine (setup details in §8.5.1). In both the open and closed systems, RCQB is several factors to orders of magnitude faster with respect to enqueue latency, wait latency or task throughput (Figure 7.3a-d). For instance, in the open system WFQ has up to 1159x higher item wait latency than RCQB and in the closed system it has up to 32.8x higher task latency than RCQB. RCQB has enqueue latency 1.7-23.1x slower than FAA (Figure 7.3a). We conclude that despite RCQB is a lock-based blocking algorithm, it has a substantial latency and throughput advantage over state-of-the-art lock free algorithms. However, in comparison to FAA its enqueue latency is still high due to its locking nature. These results motivate the need to overcome the potential disadvantages that originate from the blocking semantics of RCQB.

7.2.2 Memory Copy

The memory copy (memcpy) between private and shared buffers remains in the critical path of the application-kernel interaction, interprocess communication, and device access. It is not surprising that the memcpy alone is reported responsible for processor cycle percentage as high as 38% in production cloud benchmarks [78]. The related measurements show high sensitivity to the hardware configuration, the system settings, and the mix of colocated workloads. Thus, the memory access performance has been constantly optimized with prefetching [79, 80, 81], reduced cache pollution [78, 82], non-temporal [83] and other [84, 85] instructions.

Existing systems already include multiple memcpy versions to choose from, e.g., according to the SIMD features supported by the processor [86], but they lack adaptation at a finer grain.

To motivate this problem, we examine the performance sensitivity of a distributed filesystem to the memcpy routine for data copy between the application buffers and the client cache. We link the libcephfs library to an application to read 100 times an entire 1GB file stored on a Ceph storage cluster through the specified I/O size with 64 threads over 8 cores on a Xeon or Opteron machine with libcephfs object cache set to

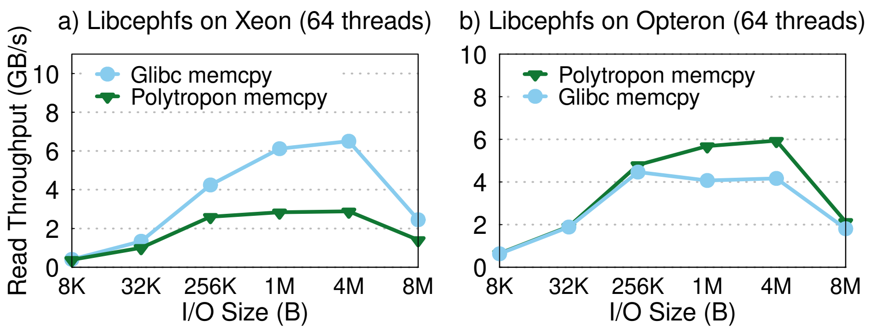


Figure 7.4: **Polytropon vs GNU libc memory copy.** This figure compares the Polytropon memory copy method with the default GNU libc memory copy regarding read I/O performance of libcephfs over Ceph.

32GB. For setup details see §9.4.1. We modified libcephfs to either use the Polytropon memory copy algorithm that we introduced in Section 5.2.5, or the default memory copy method of GNU libc. Figure 7.4 shows the total read throughput achieved by the client over a warmed object cache. As the Polytropon memcpy is optimized for the Opteron machine, it outperforms the Glibc memcpy on this machine by up to 1.42x. Interestingly, the Glibc memcpy achieves up to 2.25x higher throughput than the Polytropon memcpy on the Xeon machine suggesting the need for an automatic adaption of the memcpy routine to each system architecture.

7.3 Solution: Fast Concurrent Queues and Cross-Platform Optimization of Memory Copy

Fast Concurrent Queues Building on the RCQB queue, we introduce a family of Relaxed Concurrent Queues to combine fast enqueue and dequeue operations with short item waiting in the queue. Our algorithms use a single queue with a preallocated fixed-size array to avoid the overhead of dynamic memory management during the queue operations. We relax the FIFO ordering by splitting each of the enqueue and dequeue operations into two stages. The first stage assigns the operations in sequential order to the queue slots and the second stage completes the operations in order that depends on the slot location, the thread scheduling and the memory access arbitration. The RCQ familty contains several lock-free algorithms that can be derived from the base RCQB with minor modifications. In particular, the RCQS is a provable lock-free algorithm that uses a single-word CAS to modify both the state and data fields with a single CAS operation. In RCQD we use the CAS2 hardware instruction (instead of CAS) in order to atomically modify both the value and the state of a cell respectively stored in two different variables (instead of one). Additionally, it is possible to modify the array size from 256 slots to other multiples of 2, smaller or larger (e.g., RD2K). Alternatively, the RDEF can count the current number of enqueue and dequeue operations (or difference thereof) that arrived to the data structure in order to support the empty and full boolean operations. Finally, when an operation is unsuccessful at the current slot, we can retry it at the next slot until it succeeds with RDHP.

Cross-Platform Optimization of Memory Copy The memcpy performance depends on multiple hardware components with features that vary across different architectures, vendors, and models.

We design a copy optimization algorithm that identifies the maximum performance per transfer size by applying an exhaustive search over the load, store and register type, and the prefetch size, distance and type. We focus on the x86 architecture, which is popular and supports a wide range of instructions, register sizes and prefetch parameters. The search space includes temporal or non-temporal move instructions that support either (i) string data copy with the RSI/RDI address registers, or (ii) SIMD data move through the MMX, XMM, YMM, ZMM data registers of respective size 64, 128, 256 and 512 bits. The load or store core unit implementing a move instruction remains the same across the different data types supported by the hardware (e.g, int, float). Therefore, we only include instructions of one data type to maximize the performance for specific locality or register settings. The hardware prefetching is automatically initiated when the processor detects a predictable access pattern at a cache level. Instead, the software prefetching flexibly uses special instructions to hint the prefetch of data cache lines. Temporal prefetching is preferred for data that will be used soon and can fit at the targeted cache line. In contrast, the non-temporal prefetching limits the cache pollution by bringing the data through temporary internal buffers to L1 and avoiding the replacement activity at higher cache levels (e.g., L2/L3). The prefetch size is the amount of data fetched by a sequence of prefetching instructions. The prefetch distance is the length in bytes by which the data is requested ahead of its actual load during the memcpy.

7.4 Summary

The interprocess mechanism that is used for request and data communication among the software components of modern software stacks has become a particular important for the performance of the applications. In this chapter, we focus on two key components of an IPC mechanism, namely the queue data structure that is used for request communication, and the memory copy method that is used for data transfer.

At the queue side, we experimentally show that a strict priority order, like FIFO, limits its concurrency, while a relaxed queue has the potential to achieve stable and higher performance, even at high concurrency levels. As a solution to this problem we introduce a family of bounded relaxed lock-based and lock-free queues that relax FIFO ordering.

At the copy side, we demonstrate that the performance of critical functions like memory copy depends on the low-level characteristics of the target hardware, but there is a lack of an automated optimization method. To this end, we design a copy optimization algorithm that experimentally generates optimal memcpy routines with parameters optimized for the target system.

Chapter 8

Relaxed Concurrent Queues

The producer-consumer communication over shared memory is a critical function of current scalable systems. Queues that provide low latency and high throughput on highly-utilized systems can improve the overall responsiveness perceived by the end users. In order to address this demand, we set as priority to achieve both high operation performance and item transfer speed.

The Relaxed Concurrent Queues (RCQ) are a family of queues that we have designed and implemented for that purpose. Our key idea is a relaxed ordering model that splits the enqueue and dequeue operations into a stage of sequential assignment to a queue slot and a stage of concurrent execution across the slots.

In chapter 5 we introduced the Relaxed Concurrent Queue Blocking (RCQB) algorithm, as a component of the Polytropon toolkit. In this chapter, we provide a more formal implementation of RCQB. We also define several other variants of the RCQ algorithms with respect to offered concurrency, required hardware instructions, supported operations, occupied memory space, and precondition handling. For specific RCQ algorithms, we provide pseudo-code definition and reason about their correctness and progress properties. We developed prototype implementations of the RCQ algorithms and experimentally compare them with several representative strict FIFO and relaxed data structures over a range of workload and system settings.

The RCQS algorithm is a provably linearizable lock-free member of the RCQ family. We experimentally show that RCQS achieves factors to orders of magnitude advantage over the state-of-the-art strict or relaxed queue algorithms across several latency and throughput statistics of the queue operations and item transfers.

8.1 Definitions and Assumptions

We assume that multiple producer and consumers threads of the same or different processes exchange data items over shared memory [227]. Two operations conflict if one writes and the other reads or modifies the same memory location [261]. A data race arises in a program with two conflicting operations, unless the two operations are atomic, or one of them happens before the other. In the RCQ implementation, we prevent data races with atomic operations or locking.

An object is a container of data with a set of methods to manipulate it [59]. Program order is the order of method calls issued by a single thread. Sequential consistency is a correctness (safety) property requiring that method calls act as if they occurred in a sequential order consistent with program order. Linearizability is the correctness property that requires that each method call should occur instantaneously in the real-time interval between its invocation and response. With respect to progress (liveness), an object implementation is blocking if the delay of any one thread can delay others, and non-blocking otherwise. Unexpected thread delays are common in modern systems, for example, due to a cache miss, page fault, or scheduling preemption. A non-blocking object is lock-free if it guarantees that some method call always finishes its execution in a finite number of steps, and wait-free if it guarantees that every method call finishes its execution in a finite number of steps.

A method is *total* if it is defined for any object state, otherwise it is *partial*. The non-blocking property does not rule out the explicitly intended blocking, for example of a dequeue until the queue becomes non-empty [59]. Formally this is acceptable through a *partial* specification that leaves an operation undefined when an object does not satisfy the required precondition. A partial method can simply be totalized by returning error when not admitted by the current object state. Alternatively, pending requests can be explicitly included in the linearizable object semantics by being registered in the object state as reservations [262, 227, 263].

Recent C/C++ standards guarantee the sequential consistency of the atomic operations [264]. To avoid unnecessary overheads in RCQ, we use atomic load with acquire at a memory location to prevent reordering reads or writes before the load and also make visible to the current thread the writes from threads that release this memory location. Similarly, we use atomic store with release at a memory location to prevent reordering reads or writes past the store and also make the writes of the current thread visible to threads that load this memory location. We use atomic read-modify-write with acquire-release to perform the load operation with acquire and the store with release. Finally, we reduce the cost of spinning (*spin-pause*) with a cpu hint (e.g., x86 PAUSE [265]) [266].

We assume a 64-bit architecture with memory represented as an array of bytes. The access of the object starting at address *a* is denoted as *m[a]*. The architecture supports regular load and store operations with atomicity depending on the memory location alignment. Additionally, it supports the following atomic instructions: (i) *atomicLoad(a)*, returns the object from location *a*, (ii) *atomicStore(a, v)*, writes the object *v* to location *a*, (iii) *fetch-and-add* denoted as *FAA(a,v)*, returns the object loaded from *a* and writes back *m[a]+v* (*atomicInc(a)* is *FAA(a,1)*), (iv) *compare-and-swap* denoted as *CAS(a, o, v)* if *m[a] = o*, it stores *v* into *a* and returns true; otherwise, it returns false, (v) *compare-and-swap2* denoted as *CAS2(a,(*o_0*,* o_1*), (*v_0*,* v_1*))*, if $m[a] = o_0$ and $m[a+8] = o_1$, it stores v_0 to a, v_1 to *a+8* and returns true; otherwise it returns false. These instructions are implemented by the x86-64 hardware (e.g., FAA with LOCK XADD, CAS with LOCK CMPXHGQ, CAS2 with LOCK CMPXHG16B [267, 265]).

8.2 The Blocking RCQB

In this section, we provide the formal definition and implementation of the RCQ Blocking algorithm (RCQB) that we introduced in section 5.2.6. We also present and reason about its correctness and progress properties.

Recall from section 5.2.6 that the key insight of RCQB is to assign the enqueue and dequeue operations to the slots of a bounded queue in round-robin order but let the system scheduling policy potentially complete the operations out of order. As a result, two enqueue or dequeue operations may complete in order different from that of their assignment to the same or different slots. An interesting question is the possibility of a deadlock with the enqueuers waiting indefinitely for their slots to become free and the dequeuers waiting indefinitely at different slots for new items. The round-robin assignment of the enqueuers and dequeuers over the slots organizes them respectively into contiguous layers of consecutive enqueuers and dequeuers. As a result, it is impossible for an enqueuer to indefinitely wait at an occupied slot isolated from the dequeuers waiting at free slots. We further justify this claim in the proof of Theorem 1 (§8.2.3).

8.2.1 Implementation

Below, we describe the implementation of the RCQ data structure that is common to all algorithms of the RCQ family, and provide formal algorithms for the enqueue and dequeue operations of RCQB.

Listing 8.1: Data structures of the RCQ algorithms

```
struct slot {
  // fields in distinct cache lines
  state: int (32 bits) // state/condition variable, initially FREE
  data: int (64 bits) // value or pointer (in RCQS it includes the 1-bit state)
  waiters: uint (32 bits) // initially 0
}
struct rcq {
  // fields in distinct cache lines
  slots[N]: struct slot // N = 2^n (e.g., n = 8)
  head: uint (16 bits) // uint of byte-multiple size, initially 0
  tail: uint (16 bits) // uint of byte-multiple size, initially 0
}
```

Queue structure The queue is organized as a circular buffer implemented with a fixed-size array. The array size is limited to powers of two ($N = 2^8$ by default) so that the queue indexes can be atomically incremented and overflow to the start of the array. The data structure consists of the array, called *slots*, and two unsigned integers, called *head* and *tail* (lines 9-11 of Listing 8.1). A *slot* element consists of the *state*, the *waiters* and the *data* fields (lines 3-5). The state field is both an integer identifier of the current state and a condition variable that notifies a waiting dequeuer for a new item enqueue. The waiters field counts the waiting dequeuers; if equal to zero, an enqueuer skips the wake call for efficiency (lines 54-55 of Listing 8.4). The data field can be a 64-bit value or pointer. If necessary, the locking enqpend and deqpend states of RCQB allow the data field to have longer size.

Enqueue operation An enqueuer thread is assigned to the slot location specified by the tail index before the atomic increment (line 17 of Listing 8.2). We use atomic load with acquire and atomic store with release for efficiency (§8.1). The enqueuer reads the state of the slot (line 19). If the slot is free, the thread attempts to switch the state to enqpend with CAS. If the CAS succeeds, the enqueuer updates the data field and atomically sets the state to occupied (lines 21-23). If the CAS fails or the state is not free, then the enqueuer pauses for a short time (*spin-pause*) (§8.1) before it will read again the slot state and repeat as above (lines 28-29). If there are waiting dequeuers, the enqueuer wakes them up (lines 54-55) to attempt item removal.

Dequeue operation A dequeuer thread is assigned to the slot specified by the head index before the atomic increment (line 36 of Listing 8.3). If the thread finds occupied the state field, then it attempts to switch the state to deqpend with CAS. If the CAS succeeds, then the dequeuer copies the data field to a local variable and atomically sets the state to free (lines 40-42). If the CAS fails or the state is either enqpend or deqpend, then the dequeuer spin-pauses for a short time before it will read the slot state and repeat the above steps (lines 50-51). If the state is free, then the dequeuer rechecks the state up to *MAXSPINS* times before it will increment the waiters field and sleep on the state field (lines 61-68). We use the Linux futex synchronization facility for putting a thread to sleep [221].

8.2.2 Correctness

Here, we justify the RCQB blocking, ordering and concurrency, and reason about linearizability under relaxed ordering.

Listing 8.2: Enqueue of blocking RCQB

```
int enqueue(q: pointer to rcq, d: int){
  locTail: uint (16 bits)
  locState: int (32 bits)

  locTail := atomicInc(&q→tail) & (N-1)
  while (true) {
    locState := atomicLoad(&q→slots[locTail].state)
    if (locState = FREE) {
      if (CAS(&q→slots[locTail].state, FREE, ENQPND) = true) {
        q→slots[ltail].data := d
        atomicStore(&q→slots[locTail].state, OCCUPIED)
        wakeDeq(&q→slots[locTail])
        return(0) //successful enqueue
      }
    }
    spinPause
  }
}
```

Blocking The RCQB is a blocking algorithm that allows an enqueuer or dequeuer to lock a slot in the enqpend or deqpend state. When a blocking thread delays unexpectedly or fails, the blocked threads assigned to the same slot wait and potentially starve. We overcome this limitation with the lock-free RCQS/RCQD algorithms (§8.3). Independently of the blocking operations, we let a dequeuer sleep on a free slot until the slot will receive a new item from an enqueuer [59]. This is acceptable under the assumption that the dequeue operation is partial until the precondition of non-free slot state is satisfied.

Ordering The allocation of a slot to an enqueuer does not mean that the enqueuer will be the next to insert an item. Similarly, the oldest item in the queue will be the first to be allocated to a dequeuer, but not necessarily the first to be actually removed. Multiple enqueuers or dequeuers may be concurrently active on the same slot, if the tail or head complete full circles over the array sufficiently fast. The completion order of same-type concurrent operations on a slot is determined by the order of successful CAS completion. The sequential thread assignment to slots favors the operations that arrived earlier to complete first. A larger array size trades memory space for less CAS

Listing 8.3: Dequeue of blocking RCQB

```
int dequeue(q: pointer to rcq){
  locHead: uint (16 bits)
  locState: int (32 bits)
  locData: int (64 bits)

  locHead := atomicInc(&q→head) & (N-1)
  while (true) {
    locState := atomicLoad(&q→slots[locHead].state)
    if (locState = OCCUPIED) {
      if (CAS(&q→slots[locHead].state, OCCUPIED, DEQPND) = true) {
        locData := q→slots[locHead].data
        atomicStore(&q→slots[locTail].state, FREE)
        return(locData) //successful dequeue
      }
    }
    else if (locState = FREE) {
      waitEnq(&q→slots[locTail], FREE)
      continue
    }
    spinPause
  }
}
```

contention and allows the operations to complete independently provided that an item is dequeued exactly once after it is inserted. A smaller array size makes multiple threads more likely to compete at the same slot for the same operation type. Setting the array size equal to one diminishes the queue to an unordered bag.

Concurrency A successful enqueuer atomically locks a free slot in the enqpend state and inserts an item. An unsuccessful enqueuer retries at the same slot until it will succeed. A successful dequeuer atomically locks an occupied slot in the deqpend state and removes an item. An unsuccessful dequeuer retries at the same slot until it will succeed, or sleeps until the slot becomes non-free. The operation atomicity through the FAA and CAS instructions prevents the ABA problem showing up in queues that recycle nodes [59, 268, 269]. We update sequentially the queue head, the queue tail, and the state of each slot. We serve in alternate order the enqueuers and dequeuers of each slot to ensure that an item is dequeued strictly after it is enqueued. Multiple concurrent enqueuers and dequeuers at a slot are served sequentially. The threads assigned to different slots are served independently of each other favoring concurrency over strict ordering.

Linearizability The linearizability property requires that the operations of RCQB keep valid the queue structure and complete in the time interval between their invocation and response [270]. From the relaxed ordering of RCQB, a valid implementation should satisfy two invariants. (i) *The queue slots are sequentially allocated round-robin to the enqueuer and dequeuer threads.* It is enforced by the atomic increment of the head and tail indexes that specify the slot allocated to each thread. (ii) *An item is safely removed from a slot strictly after it is inserted.* It is guaranteed by the slot locking in the enqpend and deqpend state for the respective item insertion and removal.

An operation includes two stages, the allocation and the locking of a slot. We presume that an operation is undefined while it is waiting for a precondition (e.g., dequeue for non-free state) or inside the locking stage [270]. From the out-of-order relaxation, we presume that an operation is implemented with two distinct methods, the assignment to a slot and the locking modification of it [262, 227]. Their linearization points are respectively the completion of the FAA instruction (line 17 or 36) and the atomic store instruction (line 23 or 42) that releases the slot.

We can optionally support the *empty* and *full* boolean methods that report an empty or full queue. The queue size can be tracked through an integer atomically updated by the enqueuers and dequeuers. The update contention is reduced if the size is calculated from the difference between two separate (64-bit) counters of the enqueuers and dequeuers [59].

8.2.3 Progress

The enqueuers and dequeuers repeatedly try to lock their allocated slot. If a slot ends up allocated to multiple threads of a single type, the enqueuers retry waiting for a dequeuer to arrive, or the dequeuers sleep waiting for an enqueuer. In that case, we are concerned about a potential deadlock arising from the enqueuers and dequeuers waiting at occupied and free slots respectively without any progress. Next we show that the enqueuers cannot wait indefinitely at a slot, as long as there are active dequeuers in the queue.

We first make the simplifying but *unrealistic* assumption that the queue has infinite size. The array slots are initially free. The variables head index I_h and tail index I_t uniquely count the dequeue and enqueue operations that have arrived to the queue. Each slot is allocated *exactly one* enqueue operation (enqueuer thread) and *exactly one* dequeue operation (dequeuer thread) in arbitrary order before it will become free and never be used again. The count of each enqueue or dequeue operation is equal to the location of the allocated slot. An arriving enqueuer inserts an item to a slot without waiting. If a dequeuer arrives to a free slot early, it waits for the corresponding enqueuer to arrive and insert an item. The new operations that arrive at the queue always move to the next available slots of the array to insert or remove items.

Listing 8.4: WakeDeq and WaitEnq of RCQ

```
wakeDeq(s: pointer to slot){
  if (s→waiters >0) {
    wake(&s→slots[locTail].state) // wake up all dequeuers of s
  }
}
waitEnq(s: pointer to slot, v: value){
  i: int (64 bits)

  for (i :=0 to MAXSPINS) {
    if (s→state = v) { spinPause }
    else { return }
  }
  atomicInc(&s→waiters)
  while (s→state = v) {
    //atomically load, check, and sleep if state = v
    wait(&s→state, v) // wait on futex variable for enqueuer
  }
  atomicDec(&s→waiters)
}
```

Next we consider a *realistic* array of finite size N. We count the arriving enqueue (enq) and dequeue (deq) operations with the current value of the tail index I_t and head index I_h, respectively. The function $T_a(op)$ returns the arrival time of the op enqueue or dequeue operation. An enqueuer or dequeuer remains at a slot until it will successfully complete the assigned operation. We count the total number of enqueuers or dequeuers that have been assigned to a slot until a time instant regardless of whether the slot actually holds an item at that time or not.

Definition 1. *The set $E_a(s,t) = \{enq_i \,|\, i \,(mod\, N) = s,\, T_a(enq_i) \leq t\}$ of size $e_a(s,t) = |E_a(s,t)|$ consists of the enqueuers assigned to slot s by time t.*

Definition 2. *The set $D_a(s,t) = \{deq_i \,|\, i \,(mod\, N) = s,\, T_a(deq_i) \leq t\}$ of size $d_a(s,t) = |D_a(s,t)|$ consists of the dequeuers assigned to slot s by time t.*

Definition 3. *The function $f_a(s,t) = e_a(s,t) - d_a(s,t)$ returns the difference between the number of enqueuers and dequeuers assigned to slot s by time t.*

Lemma 1. *If $e_a(s,t)$ and $e_a(k,t)$ count the enqueuers respectively assigned to slots s and k by time t, then $e_a(s,t) - e_a(k,t) \in \{0,\, 1\}$ if $s < k$, and $\in \{-1,\, 0\}$ if $s > k$.*

Proof. From the round-robin assignment of enqueuers to slots, the number of enqueuers at slot s will be the same or one more than those at slot k if $s < k$, and the same or one less if $s > k$. $\square$

Lemma 2. *If $d_a(s,t)$ and $d_a(k,t)$ count the dequeuers assigned to slots s and k by time t, respectively, then $d_a(s,t) - d_a(k,t) \in \{0,\, 1\}$ if $s < k$, and $\in \{-1,\, 0\}$ if $s > k$.*

Proof. Follows from the dequeuer round-robin assignment. $\square$

Lemma 3. *If $f_a(s,t) > 0$ at time t, the slot s has a surplus of $f_a(s,t)$ enqueuers whose items will not be dequeued unless at least as many dequeuers are assigned to the slot.*

Proof. The dequeuers have to complete $f_a(s,t)$ circles over the slot s to remove all the remaining items of the enqueuers assigned to the slot. $\square$

Lemma 4. *If $f_a(s,t) < 0$ at time t, the surplus of $-f_a(s,t)$ dequeuers will not depart from the slot s unless at least as many enqueuers arrive to insert new items at the slot.*

Proof. The enqueuers have to complete $-f_a(s,t)$ circles over the slot s and insert an equal number of items for the remaining dequeuers of the slot to depart. $\square$

Theorem 1. *If an item is pending to be inserted at a slot or removed from it, then a dequeuer cannot indefinitely wait at a different slot for a new enqueuer to arrive.*

Proof. We assume that at time t there is at least one dequeuer active in the queue and at least one item waiting to be inserted or removed. Let s_e a slot with $f_a(s_e,t) > 0$, and s_d ($\neq s_e$) a slot with $f_a(s_d,t) < 0$. From Lemma 3, at time t there is at least one item at slot s_e with no corresponding dequeuer to remove it and from Lemma 4 there is at least one dequeuer at slot s_d waiting for a new enqueuer to arrive. The new enqueuer would simply be a thread with a new item to insert. From the above assumptions, we get:

$$\begin{aligned} f_a(s_e,t) \geq 1 \,\text{and}\, f_a(s_d,t) \leq -1 &\Rightarrow \\ f_a(s_e,t) - f_a(s_d,t) \geq 2 &\Rightarrow \\ [e_a(s_e,t) - d_a(s_e,t)] - [e_a(s_d,t) - d_a(s_d,t)] \geq 2 &\Rightarrow \\ [e_a(s_e,t) - e_a(s_d,t)] - [d_a(s_e,t) - d_a(s_d,t)] \geq 2 & \end{aligned}$$

From Lemmas 1 and 2, both the amounts $e_a(s_e,t) - e_a(s_d,t)$ and $d_a(s_e,t) - d_a(s_d,t)$ will take values from the same set $\{0, -1\}$, if $s_d < s_e$ or the set $\{0, 1\}$, if $s_d > s_e$. By considering the possible value combinations of the two amounts, their difference will belong to the set $\{-1, 0, 1\}$ and it cannot be ≥ 2. Therefore, it is impossible for a slot to have surplus of enqueuers and another to have surplus of dequeuers indefinitely. □

Intuitively, the slots preceding s_e may have one more enqueuer or dequeuer than s_e, while those following it may have one less enqueuer or dequeuer. It follows that the difference between enqueuers and dequeuers at all slots $s_d \neq s_e$ deviates at most by one with respect to s_e. Therefore, from $f_a(s_e,t) > 0$, $f_a(s_d,t)$ should satisfy $f_a(s_d,t) \geq 0$ and cannot drop below zero. Consequently, the dequeuer of s_d will consume a pending item from the current and following slots before it will reach s_e (potentially through slot 0). If necessary, the active dequeuer(s) will complete $f_a(s_e,t)$ circles around the queue to "peel off" all the remaining items of s_e.

8.3 The Lock-free RCQS and RCQD

In order to overcome the potential disadvantages of the blocking semantics, we introduce two lock-free RCQ algorithms, which we describe In the next section.

8.3.1 Implementation

The RCQD (D for double) refers to a lock-free RCQ algorithm that uses the CAS2 instruction to atomically set both the slot state and data fields, stored in two distinct

Listing 8.5: Enqueue and Dequeue of lock-free RCQD

```
72 int enqueue(q: pointer to rcq, d: int){
73   locTail: uint (16 bits)
74   locState: int (32 bits)
75   locData: int (64 bits)
76
77   locTail := atomicInc(&q→tail) & (N-1)
78   while (true) {
79     locState := atomicLoad(&q→slots[locTail].state)
80     locData := atomicLoad(&q→slots[locTail].data)
81     if (locState = FREE) {
82       if (CAS2(&q→slots[locTail], &locState, &locData, OCCUPIED, d) = true) {
83         wakeDeq(&q→slots[locTail])
84         return(0) //successful enqueue
85       }
86     }
87     spinPause
88   }
89 }
```

64-bit variables. The data field is limited to 64 bits that can store a pointer to a buffer, or the buffer itself of size up to 64 bits. Instead, the RCQS (S for single) uses the CAS instruction to set both the state and data fields in a single 64-bit variable. The buffer pointer should be aligned to an address that is a multiple of two to leave the least significant bits for the slot state. RCQS only requires the CAS hardware instruction and a memory allocator with flexible alignment, while RCQD requires the CAS2 instruction instead. In the rest of this section we only explain RCQD given the similarity with RCQS.

The slot state can only take the values *free* or *occupied*. An enqueuer (Listing 8.5) attempts to switch the slot state from free to occupied and simultaneously update the slot data. If the CAS2 succeeds (line 82 of Listing 8.5), the slot update completes and the operation returns 0. Otherwise, the slot is occupied or another thread already completed CAS2, in which case the enqueuer spin-pauses and retries (lines 87-88). A dequeuer (Listing 8.6) attempts to switch the state from occupied to free and simultaneously set the value to 0. The CAS2 operation completes if the initial state is occupied and the data is equal to the last retrieved value of the slot (lines 97-100 of Listing 8.6). If the CAS2 succeeds, the operation returns the retrieved data (line 101). If the slot state is already free, then the dequeuer invokes the waitEnq function to sleep after spinning (line 106). If the state is occupied but the CAS2 failed, then another dequeuer successfully executed CAS2 and the current dequeuer spin-pauses and retries (lines 103-104).

Listing 8.6: Dequeue of lock-free RCQD

```
90  int dequeue(q: pointer to rcq){
91    locHead: uint (16 bits)
92    locState: int (32 bits)
93    locData: int (64 bits)
94
95    locHead := atomicInc(&q→head) & (N-1)
96    while (true) {
97      locState := atomicLoad(&q→slots[locHead].state)
98      locData := atomicLoad(&q→slots[locHead].data)
99      if (locState = OCCUPIED) {
100       if (CAS2(&q→slots[locHead], &locState, &locData, FREE, 0) = true) {
101         return(locData) //successful dequeue
102       }
103       spinPause
104     }
105     else{
106       waitEnq(&q→slots[locTail], FREE)
107     }
108   }
109 }
```

8.3.2 Correctness

A slot can only hold a single item at a time, and an item is removed strictly after it is inserted. The free and occupied states are handled independently across the slots, thus relaxing the restrictions from strict FIFO ordering or empty and full states of a traditional queue. An enqueuer can insert an item as soon as the allocated slot becomes free, and a dequeuer can remove an item as soon the allocated slot becomes occupied. Additionally, an enqueuer is assigned to a slot even with all the slots occupied and a dequeuer is assigned to a slot even with all the slots free.

Linearizability The validity invariants are the round-robin sequential assignment of the operations and the safe removal of an item after it is inserted to a slot. Both these invariants are guaranteed by the atomic instructions that increment the queue indexes and update the slot contents. As with RCQB, we presume that an operation is implemented as two methods, the slot allocation and the state modification. The linearization points of the two methods respectively are the successful completion of the FAA (lines 77 or 95) and the CAS2 instructions (line 82 or 100).

8.3.3 Progress

Lock-free The definition of a lock-free algorithm requires that infinitely often some operation finishes in a finite number of steps. Assuming the operation implementation with two methods, the dequeuers or enqueuers are allowed to retry on a free or occupied slot [262, 227]. The enqueuer retries to insert an item if the slot is occupied without assigned dequeuer, or a competing enqueuer succeeds first. The successful enqueuer completes the operation in a finite number of steps and satisfies the lock-free requirement. With an argument similar to Theorem 1, the dequeuers cannot indefinitely wait on free slots while the enqueuers retry at occupied slots. A dequeuer eventually removes the inserted item, and an enqueuer subsequently succeeds at this slot.

A dequeuer retries to remove an item from a slot if the slot is occupied and other competing dequeuers complete the CAS2 operation first. Thus, the lock-free requirement is satisfied by the completion of another dequeue operation at the same slot in a finite number of steps. Adopting a partial definition, we put the dequeuer to sleep after a maximum number of retries at a *free* slot because new enqueuers may not arrive for an arbitrary time period. While there are active enqueuers in the queue, one of them will eventually reach the slot and insert an item before waking up the waiting dequeuers (similarly to Theorem 1). The waiting dequeuers will eventually complete their operation in a queue with occupied slots.

8.4 Other Variants

Below, we describe additional algorithms of the RCQ family with larger queue arrays, support for global empty and full operations, slot hoping, and bag semantics.

Listing 8.7: The rcq data structure of the RDEF algorithm

```
110 struct qsize {
111   // fields in distinct cache lines
112   esize: uint (64 bits) // incremented by a successful enqueue(), initially 0
113   dsize: uint (64 bits) // incremented by a successful dequeue(), initially 0
114 }
115 struct rcq {
116   // fields in distinct cache lines
117   slots[N]: struct slot // N = 2^n (e.g., n = 8)
118   qs: struct qsize // queue size
119   head: uint (16 bits) // uint of byte-multiple size, initially 0
120   tail: uint (16 bits) // uint of byte-multiple size, initially 0
121 }
```

RS2K and RD2K These algorithms are similar to RCQS and RCQD respectively, but the array of the queue contains 2048 slots instead of 512 by setting n=11 in Listing 8.1. Because we use 16 bit unsigned integers for the head and tail indices of the queue, the maximum size of the array can be 65536 slots.

RDEF This algorithm extends RCQD in order to support the boolean empty and full operations (§8.2.2). We extend the rcq data structure with the *qs* field (line 118 of Listing 8.7) which contains two 64-bit unsigned integer counters called *esize*, and *dsize* (lines 112-113 of Listing 8.7). Both esize and dsize are initialized with zero. The esize is incremented by a successful enqueue operation and similarly the dsize is incremented by a successful dequeue operation.

The RDEF queue provides two methods that check if the queue is empty or full. The *isEmpty()* method (lines 122-134 of Listing 8.8) reads the esize and dsize fields of qs using atomic load into two local variables. If the two local variables are equal it attempts to zero out both esize and dsize with CAS2. If the CAS succeeds, the method returns true, otherwise it returns false. Similarly the *isFull()* method (lines 135-147 of Listing 8.8) reads the esize and dsize fields of qs into two local variables. If esize is equal to $dsize + N$, it attempts to overwrite the values of esize and dsize with the values read into the respective the local variables with CAS2. If the CAS succeeds, the method returns true to indicate that the queue is full, otherwise, it returns false.

RSHP and RDHP RSHP and RDHP (HP for hopping) are RCQS and RCQD with enqueue or dequeue retry at next slot when unsuccessful at current slot. More specifically, if an enqueuer finds the state of a slot occupied or if its CAS operation on an empty slot fails, it atomically proceeds on the next slot by atomically incrementing the queue tail index. In a similar fashion, if a dequeuer finds the state of a slot empty or if its CAS operation on a full slot fails, it proceeds on the next slot by atomically incrementing the queue head index.

Bag We implement a simple bag structure using RCQD or RCQS with all the operations starting from array location zero with atomic retry at subsequent slots until they succeed.

Listing 8.8: isEmpty and isFull of RDEF

```
isEmpty(q: pointer to rcq){
  locEsize: uint (64 bits)
  locDsize: uint (64 bits)

  locEsize := atomicLoad(&q→qs.Esize)
  locDsize := atomicLoad(&q→qs.Dsize)
  if (locEsize = locDsize) {
    if (CAS2(&q→qsize, &locEsize, &locDsize, 0, 0) = true) {
      return(True) //Empty
    }
  }
  return(False) //Not Empty
}
isFull(q: pointer to rcq){
  locEsize: uint (64 bits)
  locDsize: uint (64 bits)

  locEsize := atomicLoad(&q→qs.Esize)
  locDsize := atomicLoad(&q→qs.Dsize)
  if (locEsize = locDsize + N) {
    if (CAS2(&q→qsize, &locEsize, &locDsize, locEsize, locDsize) = true) {
      return(True) //Full
    }
  }
  return(False) //Not Full
}
```

8.5 Experimental Evaluation

We compare the RCQ algorithms in open system with existing relaxed structures under balanced load and threads, strict FIFO queues under operation-pair or dedicated threads, in closed system with strict FIFO queues under load at dequeuers or imbalance between enqueuers and dequeuers, and across different (RCQ) variants including a BAG. Given the operation ordering control by the bounded relaxation and strict FIFO queues, we quantify their fit to producer-consumer communication by measuring the wait latency along with the operation and task latency or throughput.

8.5.1 Experimentation Environment

Machines We provide experimental measurements from a server with two 16-core (32 hardware threads) Intel Xeon Gold 5218 (Cascade Lake) processors (2.3GHz, 22M L3 cache) and 128GB RAM running Debian 11. We also obtained similar results from a server with four 16-core AMD Opteron 6378 (Abu Dhabi) processors (2.4GHz, 16MB L3 cache) and 256GB RAM running Debian 9. We use the Linux kernel v5.4.0, and the tcmalloc memory allocator from Google [271].

Queues We consider several representative relaxed data structures: the bounded-size k-FIFO [191] (BS-KFIFO) with 100K k-segments of $k = 64$; the distributed queue with random load balancing [250] (DQ-1RA) over $d = 1$ from $p = 64$ partial queues; the distributed queue with round-robin load balancing [250] (DQ-64RR) using $b = 64$ counter pairs (thread-local round robin) and $p = 64$; and the locally linearizable static distributed queue (LLD-DQS) [251]. The implementation of the above structures is based on the open-source Scal framework [272]. The relaxation bound of BS-KFIFO is $k = 64$ (Prop.1. [191]), and of DQ-64RR is $k = 3 * (2b + n) * (p - 1) = 36,288$ from $b = 64$, $p = 64$ and $n = 64$ threads (Prop.3.2. [250]).

Additionally, we examine several representative strict FIFO queues: the lock-free LCRQ; the wait-free queue (WFQ); the blocking Broker Queue (BQ); and the lock-free list-based Michael-Scott queue (MSQ). We implemented BQ based on literature [228], and used implementations of LCRQ, WFQ and MSQ based on public source code by Yang [273]. As a baseline of low cost we include the execution of an FAA instruction without any queue.

We consider several RCQ algorithms: RCQB is blocking RCQ with CAS, RCQS is lock-free RCQ with CAS, RCQD is lock-free RCQ with CAS2, RS2K is RCQS with array size 2048, RD2K is RCQD with array size 2048, RDEF is RCQD supporting the boolean empty and full operations, RDHP (HP for hopping) is RCQD with enqueue or dequeue retry at next slot when unsuccessful at current slot, and BAG is a simple RCQD-based bag structure..

In the array-based algorithms (including CRQ of LCRQ, WFQ), we use default queue size equal to 256, except for 2048 in RS2K and RD2K. We also examined larger queue sizes (e.g., 16384) for LCRQ and WFQ but obtained lower performance. Unlike the RCQ algorithms, MSQ can add nodes and LCRQ, WFQ can add CRQs dynamically, therefore their occupied memory space is not fixed. Similarly, the bounded relaxation queues are treated *favorably* with respect to memory because BS-KFIFO uses 100K segments, and DQ-1RA, DQ-64RR, LLD-DQS consist of unbounded list-based partial queues over *preallocated* memory by Scal [272, 251].

Frameworks In our experimentation framework we include the RCQ algorithms and the strict FIFO queues. We consider an *open system* in which the enqueuers send items to dequeuers without waiting for response, or a *closed system* in which an enqueuer inserts an item to the queue and waits for response from a dequeuer before sending the next item. We consider threads that invoke a pair of enqueue and dequeue operations with *think time* a random number (up to 100) NOP instructions per operation. Alternatively, we consider enqueue and dequeue threads that invoke operations of a single type, either back-to-back or with (up to 100K) NOP load in-between. We also examine equal or unbalanced numbers of enqueuers and dequeuers. Over Scal [272] with c=250 or 2000 cycles *load* between successive requests, we compare RCQS/RS2K to existing relaxed structures. We always pin the threads round-robin to the hardware threads, but our results were similar without pinning.

Experiment configuration Each experiment generates either 10M separate enqueue and dequeue operations in our framework, or 1M ops/thread in Scal. We measure the latency and throughput of the enqueue and dequeue operations, the wait latency of the items in the queue, the latency and throughput of the tasks submitted by the enqueuers in a closed system and served by the dequeuers. We illustrate the average value, standard deviation, or 99.9^{th}percentile. We repeat each experiment (up to 10 times) until the execution time obtains half length of the 95% confidence

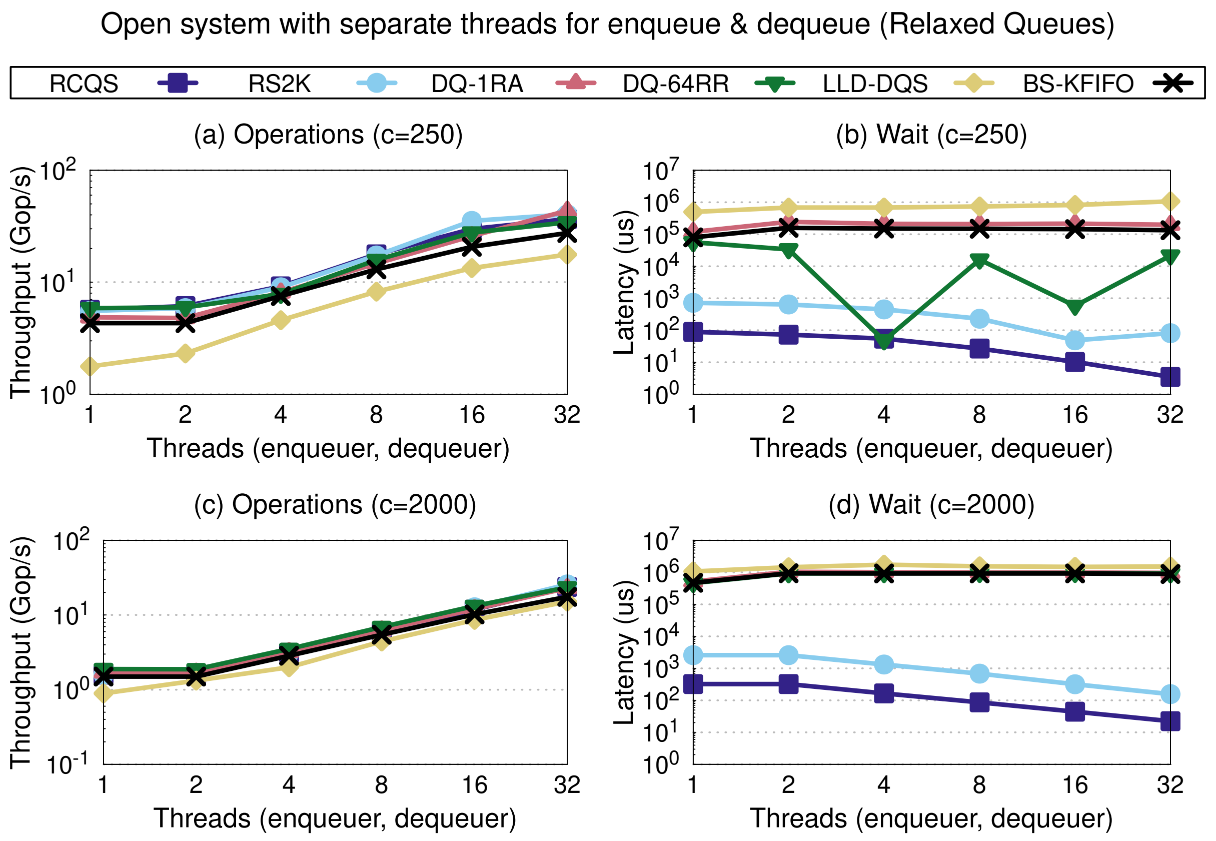


Figure 8.1: **Comparative evaluation of relaxed queues in open system.** RCQS/RS2K and existing relaxed structures with 64 total threads and c=250 or 2000 cycles load (#enqrs=#deqrs).

interval within 5% of the measured average. The reported latency of the enqueue operation spans the time period from the *first* attempt of the thread to perform the operation until the successful insertion of the item. Similarly, the latency of the dequeue operation covers the entire time period from the *first* attempt by the thread until the successful item extraction. The operation may involve multiple retries as needed by the algorithm.

8.5.2 Relaxed Queues on Open System with Dedicated Threads

In an open system based on Scal, we generate enqueue and dequeue requests using 1-32 dedicated threads (2-64 total). We consider computational load of c=250 or c=2000 cycles between the successive requests of a thread. We compare the operation throughput and the wait latency among RCQS, RS2K, DQ-1RA, DQ-64RR, BS-KFIFO and LLD-DQS. From Figure 8.1a for c=250, the algorithms have compa-

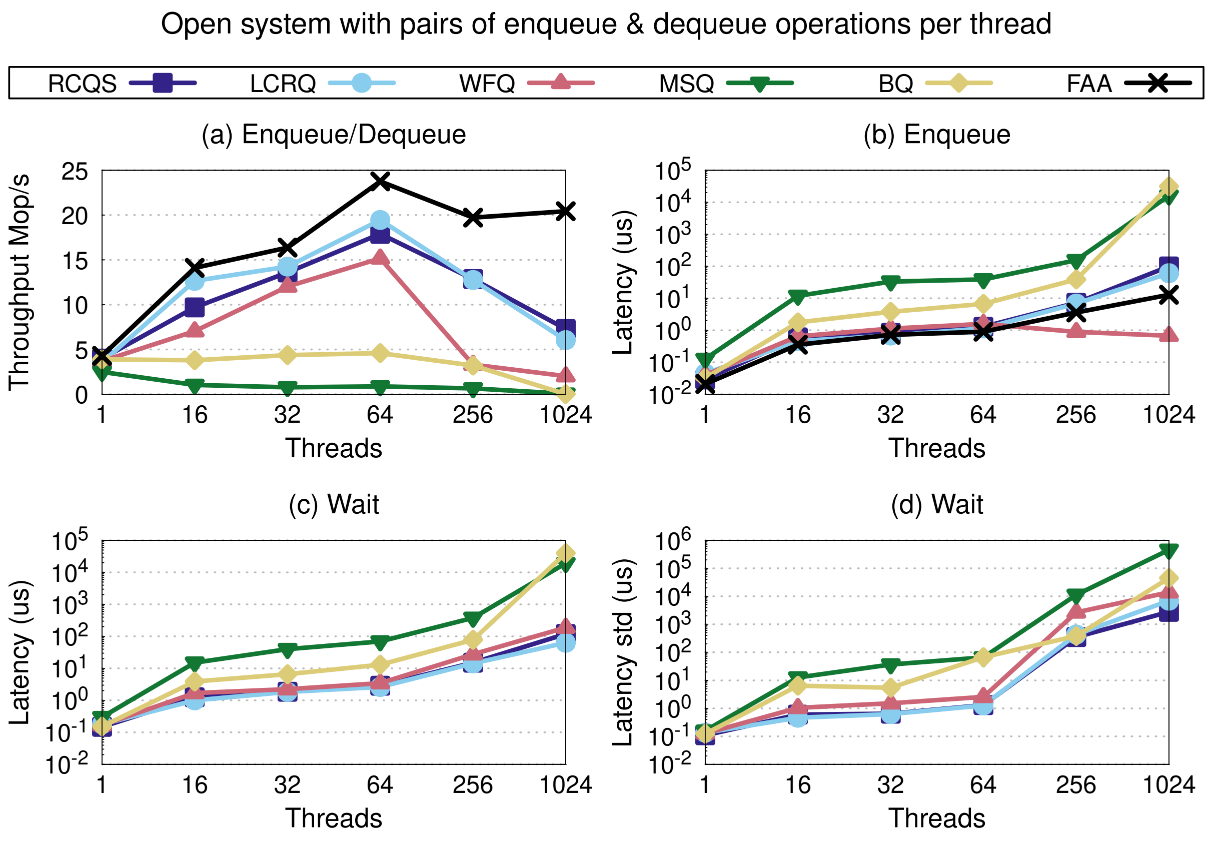


Figure 8.2: **Comparative queue evaluation in open system with operation pairs per thread.** Enqueue/dequeue throughput, enqueue latency, average queue wait latency and std. dev. across 1-1024 threads.

rable throughput (higher is better) with RCQS being faster in most cases (e.g., up to 3.2x over LLD-DQS) except for RS2K (at 16 threads). However, from Figure 8.1b, RCQS keeps the wait latency (lower is better) up to 89μs and RS2K up to 722μs, while the other four algorithms take up to five orders of magnitude ($\sim 10^5$x) longer. The comparative results remain similar for c=2000 (Figure 8.1c,d).

Therefore, RCQS/RS2K serve the requests concurrently at similar or higher operation throughput than the existing relaxed algorithms, but they forward the items substantially faster through the queue.

8.5.3 Open System with Operation Pairs

Next we use our framework to compare RCQS with strict FIFO structures, which by design remove the item that waited the longest in the queue. We evaluate an open system of threads that generate enqueue/dequeue pairs with think time (up to 100

NOPs). From Figure 8.2a, the pair throughput (higher is better) of LCRQ and RCQS approximately overlap, while that of WFQ, BQ and MSQ is substantially lower. In Figure 8.2b at ≥ 256 threads, the enqueue latency (lower is better) of WFQ drops below RCQS, unlike the respective dequeue latency (not shown) that is up to 10x higher. We also measure the queue wait latency to quantify the benefit of RCQ relaxed ordering over strict FIFO. From Figure 8.2c, LCRQ, WFQ and RCQS approximately overlap, while MSQ and BQ are substantially slower. From Figure 8.2d, the latency standard deviation of LCRQ and RCQS is substantially lower than that of the other algorithms. We attribute the variability of WFQ and BQ to the operation retries at multiple slots for strict FIFO ordering.

We conclude that RCQS has comparative performance with state-of-the-art FIFO queues in an open system with operation pairs. This experiment is common in literature, however a real producer-consumer system would likely run the enqueue and dequeue operations in distinct threads or processes.

8.5.4 Open System with Dedicated Threads

As a client-server type of producer-consumer setting, we dedicate each thread to a single operation type in our framework. In Figure 8.3a, the enqueue latency of RCQS is up to 9.5x faster than LCRQ, 27.7x than MSQ, and 16.5x than BQ, but slower than WFQ at ≤ 32 threads. From Figure 8.3b, RCQS has lower dequeue latency than the other algorithms, such as 30.6x below WFQ and 34.9x below LCRQ. The dequeuers of WFQ and LCRQ search for a slot with inserted indexed item, the dequeuers of WFQ help other threads before completing their own operation, and BQ is slower at empty or full queue. In comparison to FAA, the operations of RCQS take 5-19x longer, due to the atomic access instructions and CAS. The RCQS line remains at the bottom of Figures 8.3c,d with average wait latency 16-114μs and standard deviation up to 3.4ms. At 1 thread, the wait latency and standard deviation of LCRQ, WFQ and MSQ reach tens or hundreds of milliseconds due to their enqueuer-dequeuer contention, or the peer inspection by WFQ before returning the empty error. Similarly, the wait latency and standard deviation of LCRQ, WFQ, BQ and MSQ spike at 256 or 1024 threads due to the enqueue retries or queue creations triggered by a full queue.

We conclude that RCQS achieves significantly lower operation and wait latency than other state-of-the-art FIFO queues in a client-server open system. In addition, the item wait

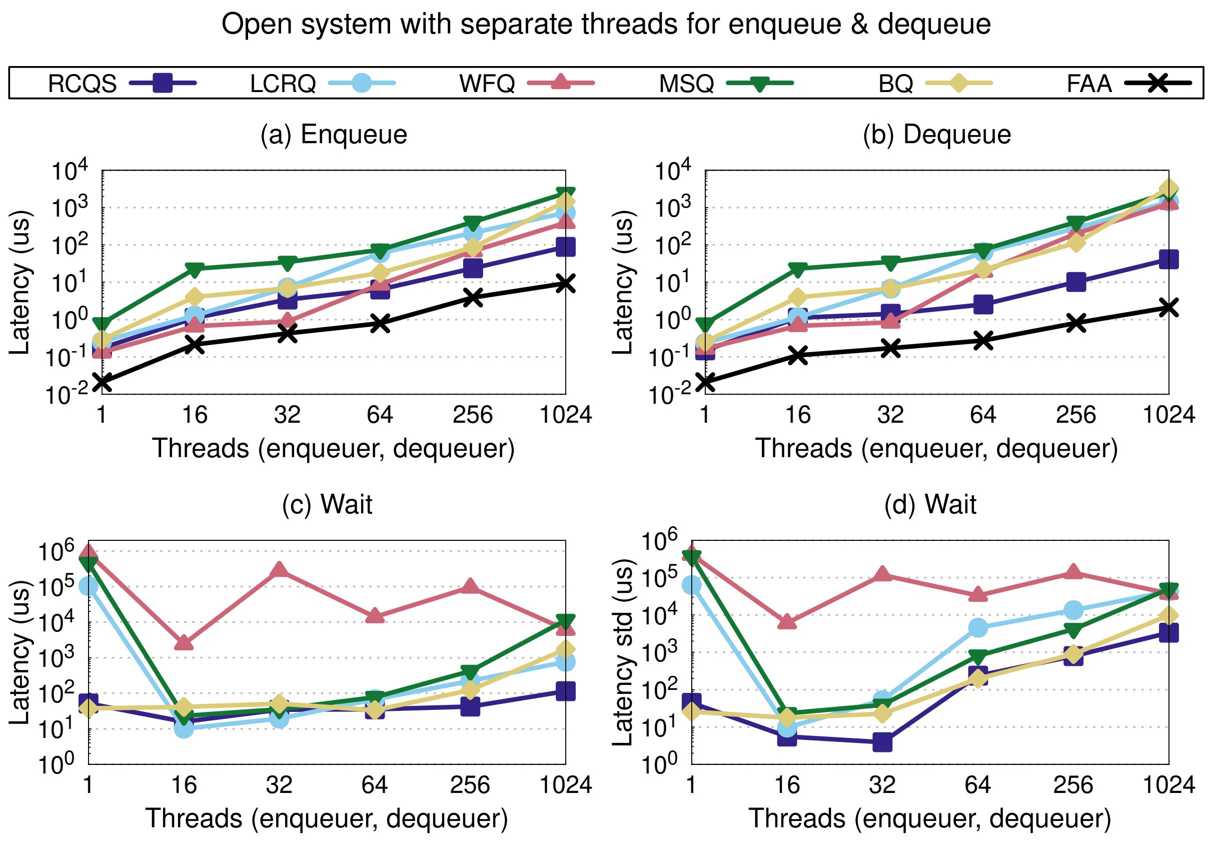


Figure 8.3: **Comparative queue evaluation in open system with separate threads for enqueue and dequeue.** Enqueue and dequeue latencies, wait latency average and standard deviation in 1-1024 threads (#enqrs=#deqrs).

latency with RCQS is stable in the order of microseconds, while the wait latency with the FIFO queues reaches tens or hundreds of milliseconds. The FIFO queue algorithms that we examined generate much higher latency due to the operation retries to maintain FIFO semantics, dynamic queue creations, and thread helping.

8.5.5 Closed System with Random Number of NOPs

In a closed system based on our framework, after the item insertion the enqueuer waits for a completion notification (through futex [221]). The dequeuer responds back after executing a random number of NOP instructions (up to 100K) as service emulation. We use an equal number of enqueuers and dequeuers (up to 2048 total). From Figure 8.4a, the average enqueue latency of RCQS lies in the range 0.5-1μs, unlike the other algorithms that take several milliseconds. From Figure 8.4b, RCQS keeps the enqueue latency 99.9%ile below 113μs, while the other algorithms take up

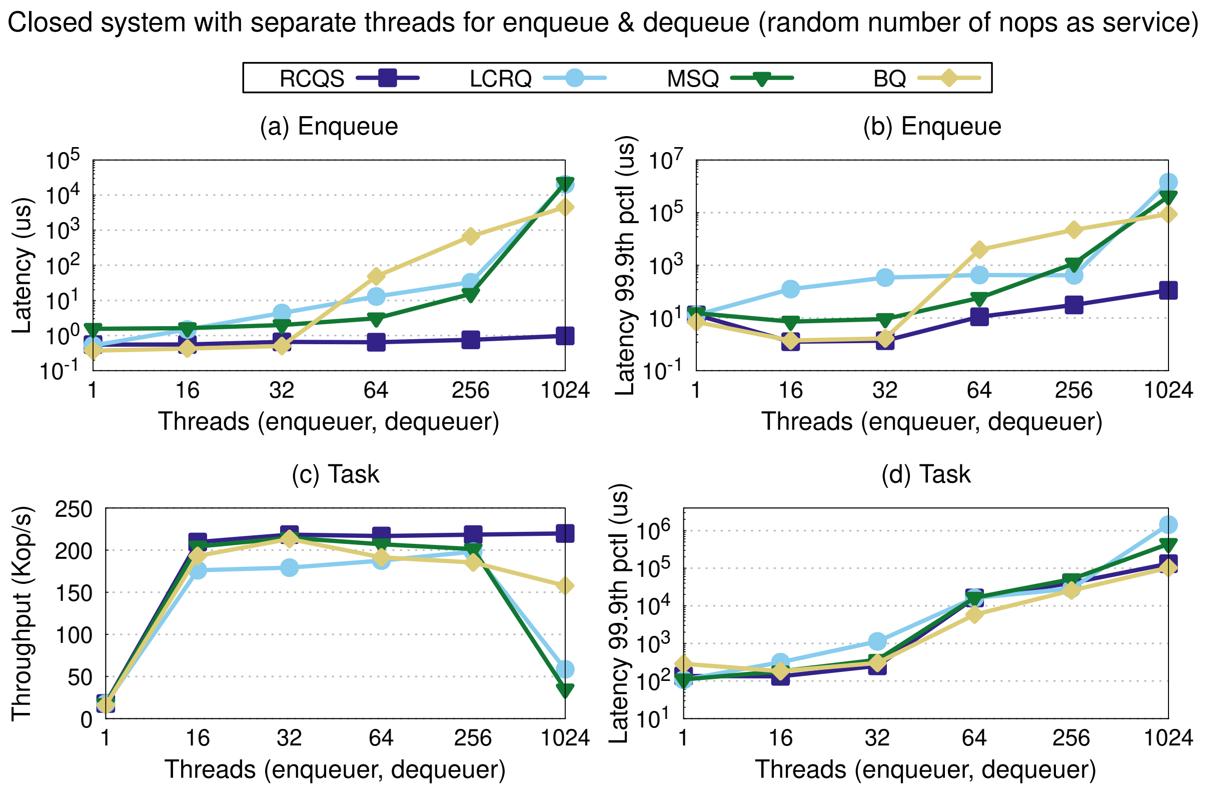


Figure 8.4: **Comparative queue evaluation in closed system with separate threads for enqueue and dequeue.** Enqueue latency (avg and 99.9%ile), task throughput and task latency (99.9%ile) for 1-1024 threads (#enqrs=#deqrs).

to several orders of magnitude longer (e.g., LCRQ reaches 1.5s). From Figure 8.4c, the task throughput of RCQS is almost flat at about 220Kop/s with $\geq$ 16 threads, and faster than BQ, LCRQ and MSQ up to 1.4x, 3.8x and 6.3x, respectively. In Figure 8.4d, we observe that the task latency 99.9%ile of BQ is comparable to that of RCQS, while LCRQ and MSQ are respectively 10.9x and 3.3x slower. We obtained similar results from other numbers of NOP instructions that we experimented with as service at the dequeuer.

Therefore, RCQS is a preferred approach with respect to the tail latency, the average operation latency and the task throughput in a closed system with balanced threads.

8.5.6 Closed System with Unbalanced Threads

An imbalance between enqueuers and dequeuers causes a frequent empty or full condition that should be handled effectively for correctness and performance stability. Over a closed system with immediate response by the dequeuer (null service), we vary

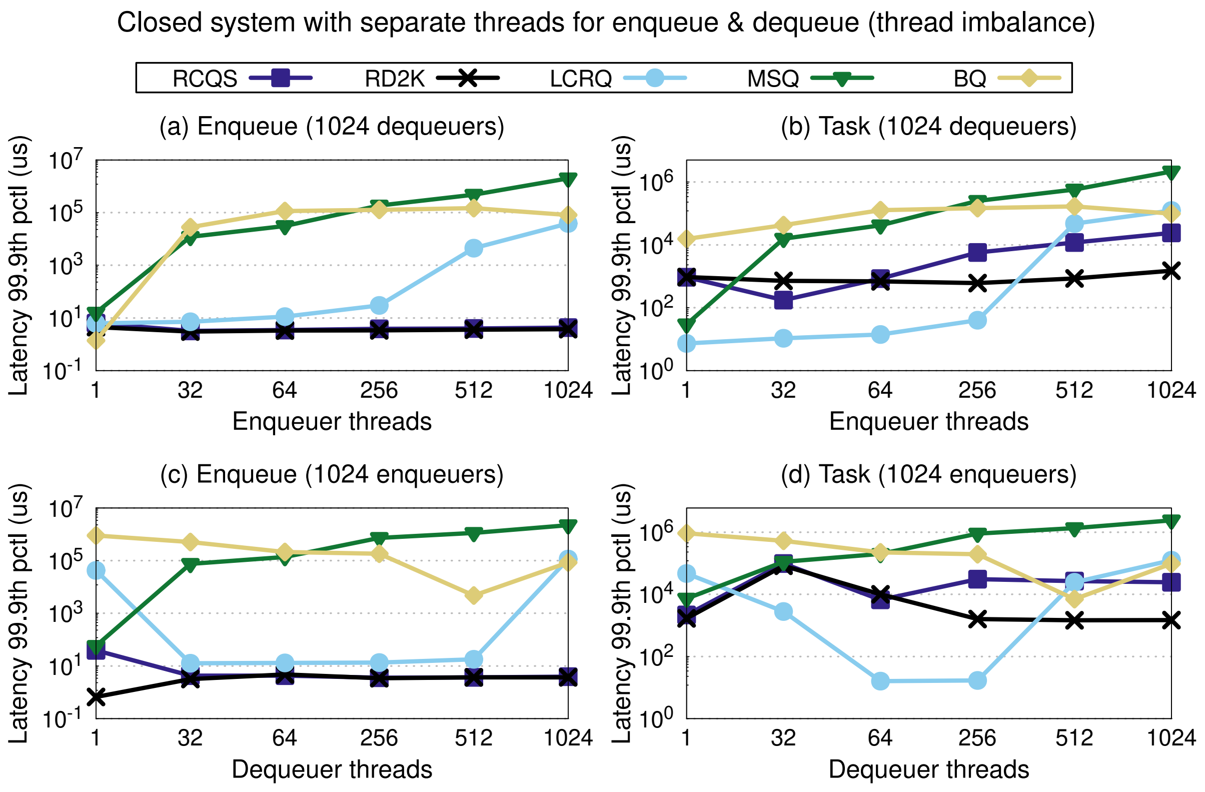


Figure 8.5: **Comparative queue evaluation in closed system with thread imbalance.** Enqueue and task latency 99.9%ile for 1024 dequeuer (a,b) or 1024 enqueuer (c,d) threads.

the number of enqueue threads up to 1024 and fix the number of dequeue threads to 1024. This setting emulates a system of variable application load and fixed server capacity. We include RD2K to explore the potential benefit from a larger array size. From Figure 8.5a, RCQS keeps the enqueue latency 99.9%ile in the tight range 3.2-6.3μs and RD2K further reduces it to 3-4.5μs. On the contrary, LCRQ takes up to 38.7ms, BQ 148ms and MSQ 2s. When the LCRQ dequeuers visit slots early, they mark them unusable and relocate the corresponding enqueuers. The BQ dequeuers check the empty condition through the tail atomically written by the enqueuers. Figure 8.5b shows the task latency 99.9%ile over the enqueuer thread count. RD2K and RCQS are less variable with up to 1.5ms and 24.1ms, while MSQ and BQ take orders of magnitude longer. LCRQ only takes up to 39.9μs at $\leq$256, but is 4-5x slower at 512 or 1024 enqueuers.

We additionally tried a variable number of dequeuers to emulate a fixed application load over a variable server capacity. From Figure 8.5c at 1 dequeuer, the enqueue latency 99.9%ile is 0.67μs for RD2K, 39.4μs for RCQS, and 58.2μs for MSQ, but three to four orders of magnitude higher for LCRQ and BQ. LCRQ and BQ take longer due to their slow handling of the full queue through a new CRQ allocation or atomic head-tail comparison. At ≥32 dequeuers, the enqueue latency 99.9%ile remains relatively flat except for LCRQ that is 4.9x slower than RCQS at 32-512 and peaks to 114ms at 1024 dequeuers. From Figure 8.5d, LCRQ has shorter task latency at 64 or 256 dequeuers, but higher than RCQS and RD2K toward the ends of the curve.

Overall, RCQS and RD2K manage the imbalance between the enqueuer and dequeuer threads with low and relatively stable enqueue and task latency. In contrast, the performance of the FIFO queues that we examined is not stable wehen varying the number of enqueuers or dequeuers. For instance, these queues encounter a large 99.9%ile enqueue latency increase under contitions of high load and full queues.

8.5.7 Open System with Unbalanced Work

In an open system, we create unbalanced enqueue and dequeue rates through a random number of NOP instructions before starting an en- queue or after completing a dequeue. We include the RDHP variation of RCQ to examine the benefit from retry at a different slot after an unsuccessful attempt. From Figure 8.6a, the enqueue latency 99.9%ile (lower is better) of LCRQ and RDHP drop up to 10.1x and 11.8x below RCQS. From Figure 8.6b, the dequeue latency 99.9%ile of RCQS and RDHP take microseconds at ≤2K, instead of milliseconds or seconds by LCRQ, BQ and MSQ. Moreover, RCQS, RDHP and LCRQ achieve faster item transfer with wait latency 99.9%ile of microseconds rather than milliseconds or seconds by BQ, WFQ and MSQ.

In Figure 8.6c, the reduced dequeue rate tends to increase the enqueue latency 99.9%ile except for the flat but slower MSQ and BQ. At 2K NOPs, RDHP is 14.6x faster than RCQS. At 200K, we note that the LCRQ tail latency takes 176μs instead of milliseconds or seconds by the other algorithms. In Figure 8.6d, the dequeue latency 99.9%ile of RDHP is up to 14.5x faster than RCQS, with both in the microsecond range. In contrast, the rest of the algorithms take milliseconds to handle the lower dequeue rate. Moreover, RCQS and RDHP achieve wait latency 99.9%ile orders of magnitude lower.

The rate imbalance causes conditions of empty or full queue that the FIFO algorithms handle through canceled slots, new queues, atomic accesses, peer helping, or synchronizations

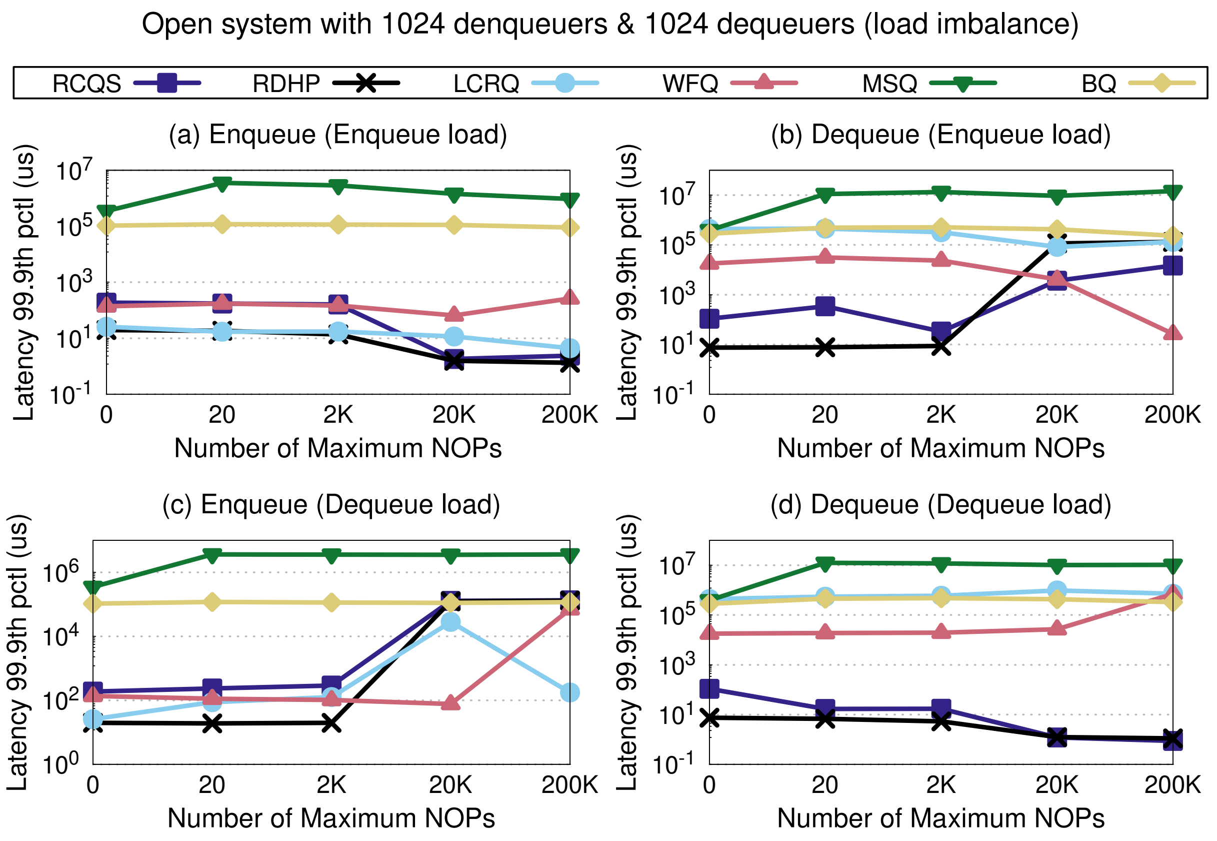


Figure 8.6: **Comparative queue evaluation in open system with work imbalance.** Enqueue and dequeue latency 99.9%ile across different loads at the producer (a,b) or consumer (c,d) side.

penalizing the operation and wait latency. On the contrary, the RCQ algorithms under the above conditions keep assigning threads to the slots at lower cost.

8.5.8 Variants of RCQ

We examine the latency 99.9%ile sensitivity to the CAS type, queue size, blocking semantics, retry location, support of empty/full, and bag semantics. The lines of RCQS and RCQD in Figure 8.7 overlap in both the open and closed system, which indicates that the CAS (CAS vs CAS2) type does not affect the overall performance. In an open system (Figures 8.7a,b), RDHP reduces up to 9.5x and 99x the enqueue and wait latency of RCQS, which indicates that the operation retry at the next slot when unsuccessful at the current slot helps enqueue and dequeue performance. The size of the queue does not seem to help performance in the open system, as RD2K has higher enqueue and wait latency than RCQS. In addition, the blocking RCQB, the

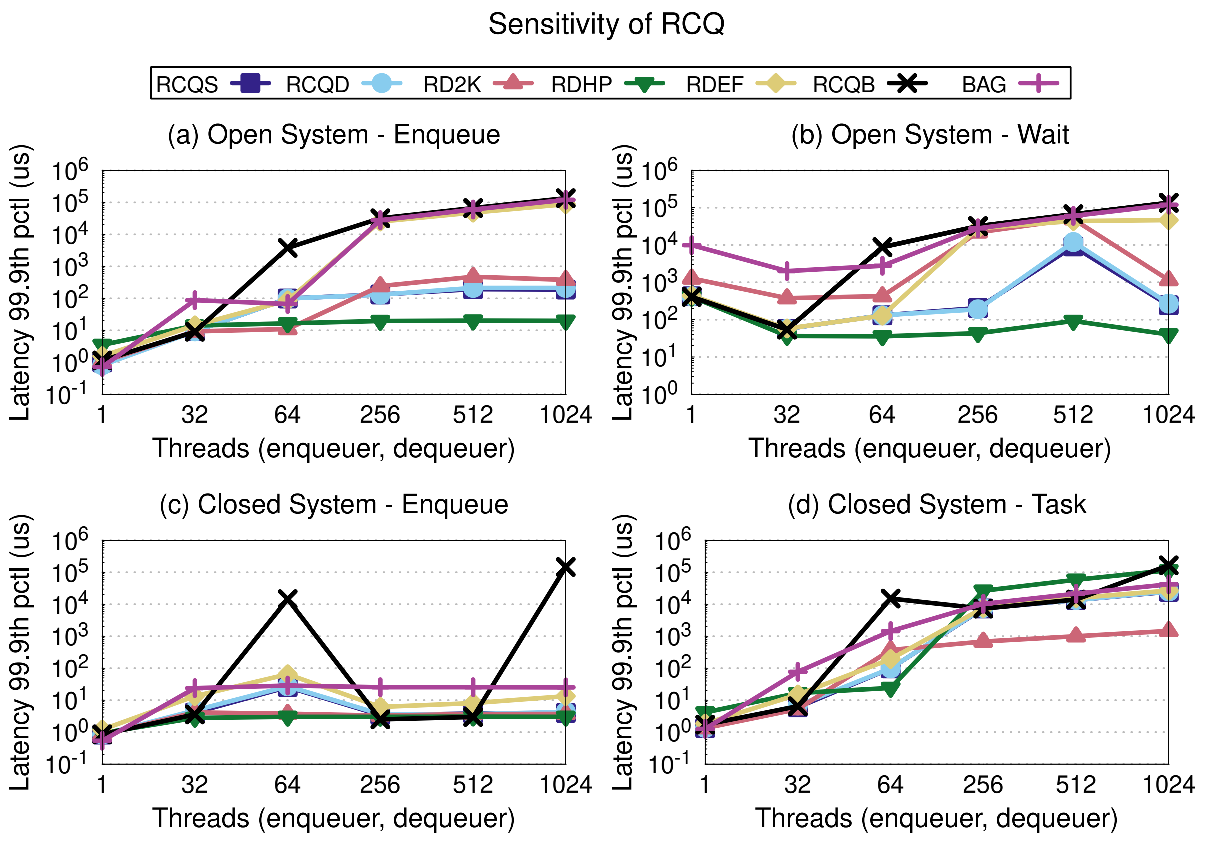


Figure 8.7: **Sensitivity analysis of RCQ variants in open and closed system.** Enqueue and wait or task latency 99.9%ile for open and closed system (NULL service) with 1-1024 threads (#enqrs=#deqrs).

RDEF and the BAG are up to two orders of magnitude slower than RCQS. From the enqueue and task latency of a closed system (Figures 8.7c,d), the BAG and RCQB are slower than RCQS/RCQD by an order of magnitude or more, while RD2K and RDHP have faster enqueue by 6.8x and 8.5x. The RD2K also reduces significantly the task latency of RCQS/RCQD which indicates that a larger queue helps in the closed system.

We conclude that the larger queue size and different slot retry help sometimes, the blocking queue is slower, the empty/full operation may increase the wait latency, the choice of CAS or CAS2 does not matter, and the examined bag algorithm approaches the slow end of the RCQ algorithms.

8.5.9 Results on Opteron CPU

We compare RCQS to previously known algorithms over a 64-core AMD Opteron machine. In the open system (Figure 8.8) RCQS is several factors to orders of mag-

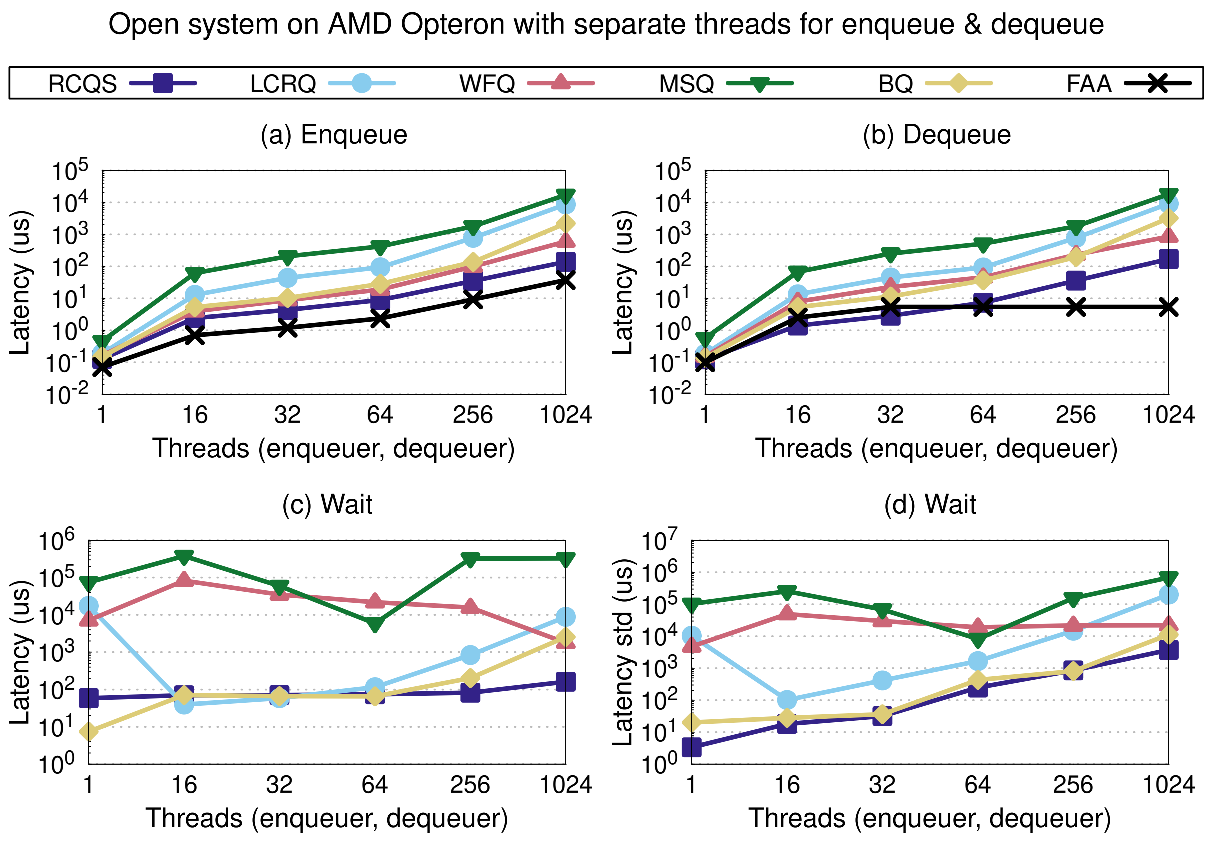


Figure 8.8: **Comparative queue evaluation in open system on AMD Opteron.** Enqueue, dequeue, and wait (average, 99%ile) latency of open system (#enqrs=#deqrs).

nitude faster with respect to enqueue latency, dequeue, and wait latency. Figure 8.8a shows that the enqueue latency of RCQS is only 1.8-3.8x higher than FAA. It is also up to 4.3x lower than the enqueue latency of WFQ, and up to 15.9x lower than the enqueue latency of BQ. In addition, the enqueue latencies of LRCQ and MSQ are up to 62.2x and 123.9x higher. Figure 8.8b depicts the dequeue latencies in the open system. The dequeue latency of RCQS is only 1.7-5.3x higher than FAA and is the lowest of all the other queue algorithms that we tested. The dequeue latencies in WFQ, BQ, LCRQ, and MSQ are up to 5.1x, 19.1x, 54.3x, and 106.9x higher. The high dequeue latency of the other algorithms is justified due to the strict FIFO semantics that require from the dequeuers to synchronize with the enqueuers or to cancel queue slots. The low dequeue latency of RCQS helps the queue to keep the item wait latency low and stable (Figures 8.8c,d). In contrast, the strict FIFO algorithms raise the wait latency up to 5428.5x (MSQ with 16 enqueuers and 16 dequeuers). LCRQ has up to 297.6x higher average with up to 4111.2x higher standard deviation wait latency than RCQS, while WFQ raises the average wait latency of RCQS up to 1185x, and the

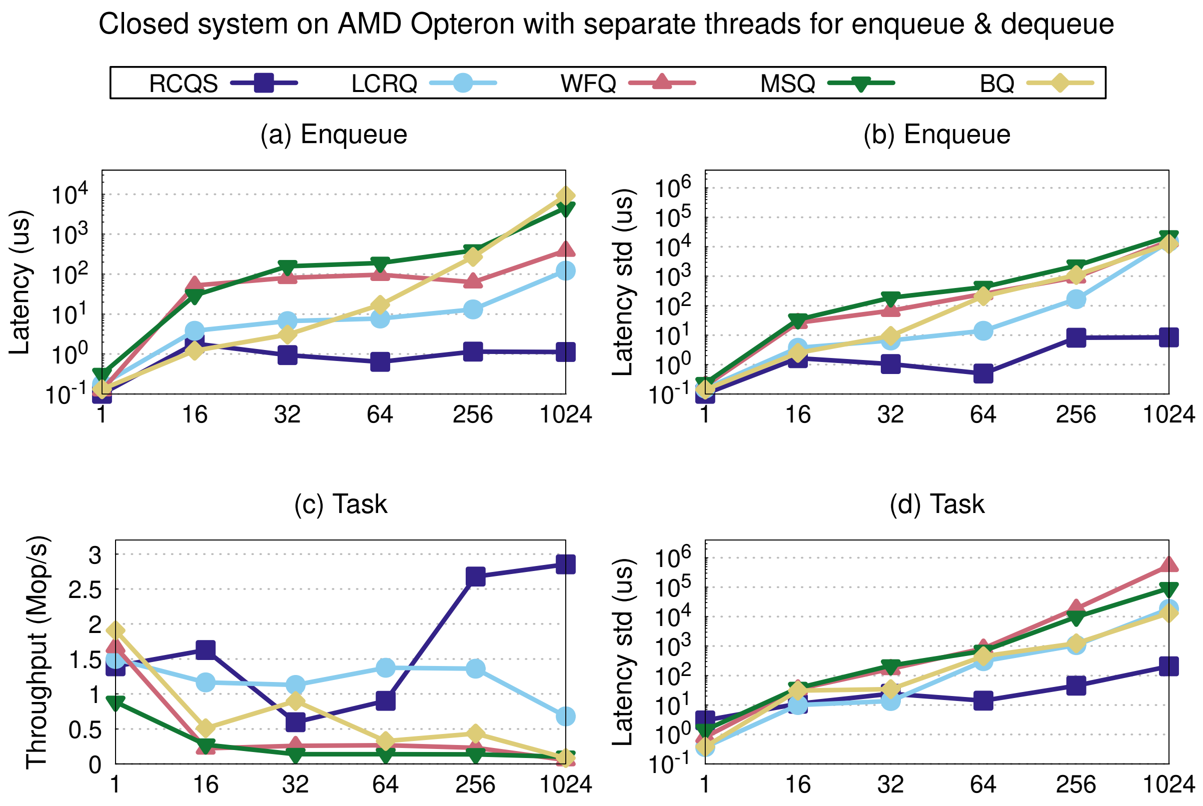


Figure 8.9: **Comparative queue evaluation in closed system on AMD Opteron.** Enqueue latency (average, 99%ile), dequeue latency (average), task throughput (average), and task latency (99%ile) of closed system (#enqrs=#deqrs).

standard deviation latency up to 1420.4x.

Similarly, in the closed system (Figure 8.9) RCQS is faster with respect to enqueue latency, task latency, and task throughput than the strict FIFO algorithms that we examined. Figure 8.9a shows that the enqueue latency of RCQS is stable (0.10-1.13us) and substantially lower than the enqueue latencies of the other queue algorithms. LCRQ raises the average enqueue latency up to 7.1x with 32 enqueuers and 32 dequeuers and up to 107.6x in an oversubscribed setting with 1024 enqueuers and 1024 dequeuers. In a similar fashion, WFQ raises the enqueue latency of RCQS up to 343.4x, MSQ raises it up to 4044.7x, while BQ has the highest enqueue latency that is up to 8210.8x higher than RCQS. RCQS has also the lowest standard deviation of enqueue latency (Figure 8.9b) which is up to 1768.2x lower than LCRQ, up to 1879.5x lower than WFQ, up to 2688.5x lower than MSQ, and up to 1503.4x lower than BQ. We notice that the MSQ algorithm has the highest standard deviation of enqueue latency due to the dynamic allocation of a new queue slot to handle each enqueue operation. Furthermore, RCQS achieves scalable task performance as Figure 8.9c depicts. The task throughput of RCQS is up to 52.8x higher than WFQ, up to 4.2x higher than LRCQ, up to 27.9x higher than MSQ, and up to 34x higher than BQ. Finally, as Figure 8.9d shows RCQS keeps the standard deviation of task latency significantly lower than the other queues.

We conclude that RCQS has a stable behavior and a substantial latency and throughput advantage over the other algorithms at comparable ratio on the AMD system.

8.6 Summary

We introduce the RCQ family of queues to increase concurrency by allowing the operations to complete out-of-order after they are assigned sequentially to the queue slots. We define and justify several correctness and progress properties of the blocking RCQB and the lock-free RCQD algorithm. The latency and throughput are critical performance metrics in current manycore systems with devices of high throughput and low latency. Assuming that the FIFO queue is typical choice in producer-consumer software stacks, we examined the implications of strict FIFO or relaxed ordering to several operation, wait and task metrics. In comparison to existing algorithms across different workload and system settings, the lock-free RCQ algorithms achieve factors to orders of magnitude higher throughput and lower operation and wait latency. Across different levels of concurrency or thread imbalance, a single array of fixed small size (e.g., 256 slots) achieves stable operation with limited resource requirements.

If threads of the same type continuously arrive and compete at a slot, RCQ cannot completely rule out starvation. We experimentally evaluated this possibility through statistics of the enqueue/dequeue and wait/task latency or throughput in balanced or unbalanced threads. From our standard deviation and percentile measurements, we found that the RCQ algorithms combine stable behavior with low latency and high throughput. In contrast, the existing strict or relaxed algorithms often generate much higher tail latency due to the synchronization costs and operation retries, or the relaxation semantics. To the best of our knowledge, our work is the first to comprehensively evaluate the performance statistics of the queue operations and the queue wait or task completion in open and closed systems. We conclude that the RCQ family of queues provides a promising fast and scalable alternative to the existing strict or relaxed queues in order to meet the communication demands of current manycore systems and low-latency devices.

Chapter 9

The Asterope Algorithm for Fast Memory Copy

We recognize the complexity of the software and hardware parameters involved in critical system operations. With initial focus on the memory copy (memcpy) function, we introduce a methodology based on exhaustive search to optimize the performance across different platforms. We introduce the Asterope algorithm to experimentally generate optimal memcpy parameters for two x86-64 processor models from different vendors.

With experiments on standalone memcpy and a distributed file system, we demonstrate that the memory copy algorithms that Asterope produces for two different systems can track or even improve the performance of the fastest memory copy routine. We also demonstrate higher function and system performance up to 2.4x and 1.9x in comparison to Linux kernel memcpy.

9.1 Design Space

We develop a software facility that takes advantage of the available hardware features to achieve fast data transfers over main memory. The data transfer is an iterative process of load and store instructions that copy data over consecutive addresses with non-overlapping source and destination. We aim to minimize the total transfer time in order to accelerate the communication between the producers and consumers of client-server requests. Depending on the thread scheduling policy and the processor architecture, the cores serving the communicating producer and consumer may share caches. Although the consumer would benefit from finding the transferred data resident in cache, the actual gain is sensitive to the amount of data and the type of subsequent computation. Thus, we focus on getting the data fast to the destination location leaving the caching itself beyond the scope of this work.

9.1.1 Bottlenecks

The data transfer throughput is limited by the bandwidth of the memory channels and their path to the registers. If the source and destination addresses lie on the same or different memory channels of a socket, then the throughput is respectively limited by half or the entire bandwidth of a memory channel. In remote socket memory accesses, the bandwidth is further limited by the links of the NUMA interconnect (e.g., UPI links of Intel Cascade Lake). An additional transfer limitation is the number and size of the concurrent data requests per core. Multiple outstanding load or store requests are served through the Line Fill Buffers (doubling as write-allocating or write-combining buffers). Therefore, by application of Little's law, the transfer throughput is approximated by the total capacity of the Line Fill Buffers divided by the memory access latency [258, 256].

9.1.2 Streaming

The non-temporal (streaming) moves avoid polluting the higher levels (e.g., L2/L3) of the processor cache and reduce the coherence traffic. The amount of cache space involved depends on the speed with which the requested data is fetched and subsequently flushed to the destination address [253]. The load throughput depends on the cache level at which the requested data is found and the number of cache misses

Move Instructions in the x86 Architecture

Extension [Reg]	Type	Integer [Note]	Floating-Point	Prefetch
X86 Core [R(S|D)I]	T	movsb, movsw, movsl, movsd, movsq		
MMX [MM]	T	movd, movq		
SSE [XMM]	T		movss, movaps, movups	
	NT	movntq	movntps	
SSE2 [XMM]	T	movdqa, movdqu	movsd, movapd, movupd	prefetch(0|1|2)
	NT	movntdq, movnti [GPR]		prefetchnta
SSE4.1 [XMM]	NT	movntdqa [load]		
AVX, AVX2, AVX-512, [(X|Y|Z)MM]	T	vmovd, vmovq, vmovdq, vmovdqa, vmovdqu	vmovaps, vmovups, vmovapd, vmovupd	
	NT	vmovntdq, vmovntdqa [load]	vmovntps	

Table 9.1: The available load, store, and prefetch instructions in x86.

that the processor can handle concurrently [274]. If the stores incur write allocates, they may lead to read-modify-write cycles. As a result, two loads are required per store and the maximum copy throughput drops from half to one third of the memory channel bandwidth. The developer may use non-temporal stores to skip the write allocates and take advantage of write combining to coalesce multiple stores to entire CLs.

9.1.3 Register Type

The x86 architecture includes a number of 64-bit general purpose registers, known as RI, RSI, and RDI, that can be used for copying data across different memory locations.

However, due to the size of the registers only 64 bits of data can be copied within a single operation. Fortunately, modern processors are also equiped with vector registers that hold more than 64 bits of data. Three well-known vector register families of the x86 architecture are the xmm, ymm, and zmm that can hold 128, 256, and 512 bits respectively.

The family of instruction sets which enables the usage of vector registers is known as ingle Instruction Multiple Data (SIMD). SIMD contains instructions that allows the cpu to execute the same operation on multiple data points simultaneously. The number of simultaneous operations depends on the size of the used registers.

Intel introduced in the late 90's the SIMD Extensions (SSE) that operate on the 128-bit xmm registers. SSE contains several sets of instructions, such as SSE2, SSE3, and SSE4.1.A more recent family of SIMD instructions is the AVX that comprises two sets, namely AVX, and AVX2. These sets operate on 256-bit ymm registers. AVX-512 is a 512-bits extension of the 256-bit operations of AVX2 that operate on zmm registers. Table 9.1 summarizes the available move instructions and registers in the x86 architecture.

9.1.4 Register Size

The move instructions used for the transfer specify the opcode and register size along with the core frequency of the executed code. Larger registers may complete a transfer of specific size with fewer operations but through larger instruction opcodes and longer byte sequence. Architectural power restrictions may lead to lower operation frequency that reduces the actual transfer throughput for specific instructions. Therefore, the processor vendors (e.g., Intel [254]) often suggest to apply comparative benchmarking in order to identify the most effective register size for each workload per architecture.

9.1.5 Buffering

The instructions of software prefetching and non-temporal move consume temporary internal buffers for handling cache misses or write-combining. The small number (e.g., 10-12) of temporary internal buffers limits the concurrency of outstanding memory accesses and the achieved transfer throughput. A suggested solution is to transfer the data in two steps through a small amount (e.g., 2-4KB) of buffer space located in main memory and cached in the processor (e.g., L1) [275, 276]. Thus, the data from the source address is transferred through the registers to the cached memory buffer before it will reach through the registers again the destination address. The memory buffer has sufficient capacity to probably relieve the contention for temporary internal buffers. As a result, the loaded data can free internal buffers faster and the stored data can be combined faster into CLs. In practice, the actual benefit (if any) depends on the architecture and the applied parameter tuning through benchmarking.

9.2 Prefetching Pipeline

We strive to hide the latency of future load requests through hardware or software prefetching concurrent with ongoing requests. Prefetching transfers data to the processor caches without contention for registers. *We can model the resulting data transfer with a pipeline of three stages: the prefetch, the load and the store*. Assuming that the prefetched data arrives timely at the cache, the load is expected to take negligible time in comparison to the other two stages.

9.2.1 Hardware Prefetching

The hardware prefetching is automatically initiated when the processor detects a predictable access pattern at a cache level. Indeed, a memcpy operation over consecutive addresses trivially generates an access pattern of unit stride. The target cache type (instruction or data) and level (e.g., L1 or L2/L3) of the prefetch is specified by the activated hardware prefetcher. The hardware prefetchers are limited by the fixed cache type and level they target, the initial cache misses from demand requests they need to start, and the location of the prefetched data within a single page.

9.2.2 Software Prefetching

The software prefetching flexibly uses special instructions to hint the prefetch of data CLs. The data is transferred to L1 via temporary internal buffers for a non-temporal instruction, or to the specified cache levels for temporal instructions. Temporal prefetching is preferred for data that will be used soon and can fit at the targeted cache level. In contrast, the non-temporal prefetching limits the cache pollution by bringing the data through temporary internal buffers to L1 and avoiding the replacement activity at higher cache levels (e.g., L2/L3).

9.2.3 Parameters

We call *transfer block* the data amount transferred per memcpy iteration and set it equal to a CL pair (128B). The *prefetch unit* is the data amount prefetched by a single instruction [260] with default value the CL size (64B). The *prefetch size* is the amount of data fetched by a sequence of prefetching instructions. We invoke prefetching at the beginning of the memcpy iterations at which the current transferred amount becomes multiple of the prefetch size. The *prefetch distance* is the length in bytes by which the data is requested ahead of its actual load during the memcpy.

9.2.4 Tuning

We fetch the data to the cache in advance so that it can be stored to the destination memory address with low load latency.

The prefetching consumes bandwidth from the memory device, the memory channel, the socket interconnect and the coherence structure, but also storage space from the shared or private caches of the core, the temporary internal buffers and the registers. Resource contention arises from the demand requests, the speculative requests, and the software or hardware prefetch requests.

The time required to prefetch the data depends on the prefetch size and distance. The prefetch size is multiple of the prefetch unit and the transfer block size, but it is limited by the cache space left by the loads and stores. The prefetch distance follows from the prefetch size and the prefetch bandwidth. The prefetch bandwidth can be derived from the memory latency and the temporary buffer space left by the concurrent loads (e.g., misses) and stores (e.g., write-combining). The prefetch and block size determine the number of switches between memory reads and writes (turnarounds). A larger prefetch size incurs fewer switches, but it involves more registers and increases the concurrent requests, the prefetch distance, and the cache space. Given the number and complexity of the factors involved, we apply benchmarking in order to identify the preferred prefetch size and distance.

The x86-64 Instructions Used by Asterope

Regs/Pref	Load		Store	
	Non-temp	*Temp*	*Non-temp*	*Temp*
RSI/RDI	-	rep movsq	-	rep movsq
MMX	-	movq	movntq	movq
XMM	movntdqa	movaps	movntps	movaps
YMM	vmovntdqa	vmovaps	vmovntps	vmovaps
ZMM	vmovntdqa	vmovaps	vmovntps	vmovaps
Prefetch	prefetchnta	prefetcht0,1,2	-	-

Table 9.2: Explored move instructions per register size. *nt:* non-temporal, *q:* quadword, *dqa:* aligned double quadword (int), *aps:* aligned packed single-precision (float).

Listing 9.1: Definitions of the Asterope algorithm

```
#define CLSZ 64                                          // cache line (bytes)
#define BSZ (2 * CLSZ)                                   // block size (bytes)
#define MXPS 16                                       // max pref size (blocks)
#define MXPD 16                                       // max pref dist (blocks)
enum op_type = {load_t, load_nt, store_t, store_nt};
enum prf_type = {prf_t0, prf_t1, prf_t2, prf_nt};
enum reg_type = {rsirdi, mmx, xmm, ymm, zmm};
uint xfer_size[] = {128, 8KB, 256KB, 1MB, 4MB, 32MB};
```

9.3 The Asterope Algorithm

The memcpy performance depends on multiple hardware components with features that vary across different architectures, vendors, and models. *Our key idea is simply to identify the maximum performance per transfer size by applying an exhaustive search over the load, store and register type, and the prefetch size, distance and type*. We focus on the x86 architecture, which is popular and supports a wide range of instructions, register sizes and prefetch parameters. The search space includes temporal or non-temporal move instructions (Table 9.2) that support either (i) string data copy with the RSI/RDI address registers, or (ii) SIMD data move through the MMX, XMM, YMM, ZMM data registers of respective size 64, 128, 256 and 512 bits. The load or store core unit implementing a move instruction remains the same across the different data

Listing 9.2: The Asterope optimization algorithm

```
asterope (char *to, char *from, uint xfsz[]){
  uint xi, bi, ps, pd, pi;
  enum prf_type pt;
  enum op_type lt, st;
  enum reg_type rt;
  for (xi = 0 to (sizeof (xfsz) - 1))                          // xfer size index
    // search for (ps, pd, pt, lt, st, rt)
    // with max memcpy performance per xfsz[xi]
    for (ps = 0 to MXPS)                                        // prefetch size
      for (pd = 0 to MXPD)                                  // prefetch distance
        for (pt = prf_t0 to prf_nt)                             // prefetch type
          for (lt = load_t to load_nt)                              // load type
            for (st = store_t to store_nt)                         // store type
              for (rt = rsirdi to zmm) {                        // register type
                // clear cpu caches
                // record memcpy duration
                for (bi = 0 to ((xfsz[xi] / BSZ) - 1)) {
                  // prefetch stage
                  if (ps && !(bi mod ps))
                    for (pi = 0 to (ps - 1))
                      blkmemprf (from+(pd+pi)*BSZ,pt);
                  // load and store stages
                  blkmemcpy (to, from, lt, st, rt);
                  // next transfer block
                  from += BSZ;
                  to += BSZ;
                }
                // fence for non-temporal store
              }
}
```

types supported by the hardware (e.g, int, float) [277]. Therefore, we only include instructions of one data type (e.g., dqa or aps suffix in Table 9.2) to maximize the performance for specific locality or register settings.

We provide in C-like pseudo-code the definitions (Listing 9.1) and implementation of the Asterope algorithm (Listing 9.2). We omit the handling of unaligned addresses for code readability. The transfer sizes that we examine are multiples of the transfer block (2 CLs) for experimentation efficiency. The hardware prefetching is enabled by default. We hint the software prefetching with temporal instructions (*prefetcht0,1,2* to multiple cache levels) or non-temporal (*prefetchnta* to L1). We specify the prefetch size and distance (lines 17, 18) in transfer block multiples (max 32 CLs). We prefetch each CL only once with the loop of block prefetches (*blkmemprf*) executed (line 27) at prefetch size multiple from the start. The block memcpy (*blkmemcpy*) function transfers one block (line 31) with the specified load, store and register types.

The Asterope algorithm returns the parameter tuple with maximum memcpy performance per examined transfer size. We use the examined transfer sizes to partition the respective domain into non-overlapping left half-open intervals. For the transfer sizes of each interval, our routine (called *Asterope memcpy*) uses the returned parameter tuple corresponding to the right endpoint of the interval.

9.4 Experimental Evaluation

In this section, we experimentally measure the average performance of the Asterope and other memcpy routines running standalone. At system level, we measure the read I/O throughput of a distributed filesystem client copying data between the application buffers and the client cache.

9.4.1 Experimentation Environment

Intel Xeon Gold 5218 (Cascade Lake) [16C(32HT)@2.3GHz]

L1	L2	L3	Cl	Ch	Bw/Ch	Lat
1MB	16MB	22MB	2	6	19.87GB/s	88.9ns

AMD Opteron 6378 (Abu Dhabi) [16C@2.4GHz]

768KB	16MB	16MB	1	4	12.8GB/s	84.7ns

Table 9.3: Basic processor parameters including bandwidth per channel and local-NUMA-node memory latency.

We provide numbers from two machines: (i) a dual-socket server with Intel Xeon processors and 128GB, or (ii) a quad-socket server with AMD Opteron and 256GB RAM. Table 9.3 summarizes several hardware features per socket as specified by

Evaluated Memory Copy Routines

Name	**Description of Instructions(x86) and Parameters**
Asterope	as specified in Table 9.2
Glibc	GNU libc memcpy (x86-64) consisting of multiple routines
Linux5	rep movsq/movsb with RSI/RDI
Musl	32-bit unsigned pointers
Buffered	4KB buffer, prefetchnta (init. 320B), movdqa/movntdq (64B)
Polytropon	prefetchnta (init. 640B, rep. 128B), movaps/ups/ntps (128B)

Table 9.4: Description of the memcpy routines.

the vendors or measured in our testbed [278, 62]. Both servers have the memory interleaving disabled by default. The maximum theoretical memcpy throughput with load and store on the same channel can be roughly estimated from half the memory channel bandwidth.

We evaluate the memcpy from Asterope, Glibc [86], Linux kernel v5 [279] (Linux5), Musl [220], Buffered [276] and Polytropon. We summarize the features of each routine in Table 9.4. Some code dependence complications prevented us from including the Cosmopolitan [280] C library, but we report notably higher relative improvements (e.g., over Glibc). We implemented a kernel module to activate write-combining memory and examine the `movntdqa` non-temporal load instruction, but found it several factors slower than temporal load over write-back. In total, we examine six different parameters in 138,720 total combinations, which takes several hours to complete over a single machine. For each transfer size, we identify the highest performing Asterope configuration by measuring the average throughput over 100 iterations (with Linux *clock_gettime*). Similarly, for the other memcpy routines we report the average throughput over 100 iterations. The reported numbers are relatively stable with 95% confidence interval within 2% of the measured average.

9.4.2 Standalone Memory Copy

Figure 9.1 illustrates the memcpy throughput at different transfer sizes using 1 thread (on Debian 11 Linux). Asterope achieves 7.93GB/s at 256KB in Xeon, which makes it 1.4x faster than Musl and 1.6x over Glibc and Linux5. In Opteron, the maximum throughput of Asterope is 6.79GB/s at 256KB transfer size. The advantage of Aster-

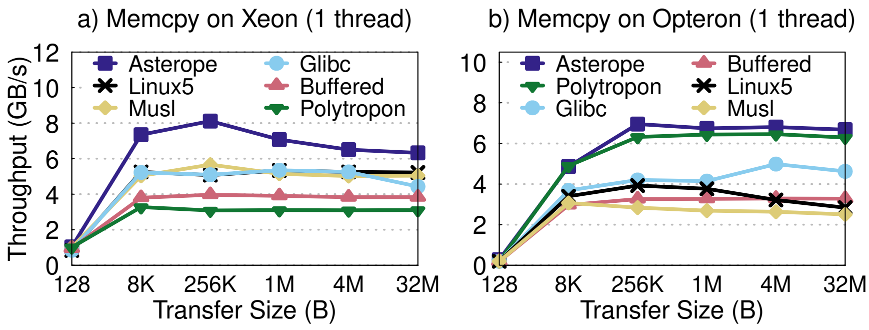


Figure 9.1: **Standalone copy throughput.** This figure depicts the copy throughput across 6 memcpy routines and 2 cpus. Asterope achieves the best possible performance on both systems.

ope is 1.7x over Glibc and 2.5x over Musl at 256KB, 1.5x over Polytropon at 128B, and even 2.4x over Linux5 at 32MB. In Table 9.5, we show the registers, instructions and prefetch parameters picked by Asterope. Asterope prefers the non-temporal instructions and prefetching for Opteron, and a variety of prefetch sizes and distances across the different transfer sizes over the two processor models.

Automatic Settings by Asterope (Xeon)

Size	RT	LT	ST	PT	PS	PD	GB/s
128B	MMX	T	NT	T0	2	0	1.01
8KB	YMM	T	T	T1	14	0	7.17
256KB	XMM	T	T	T0	4	30	7.93
1MB	MMX	T	T	T0	2	28	6.9
4MB	MMX	T	T	T0	2	28	6.36
32MB	MMX	T	T	T0	2	30	6.18

Automatic Settings by Asterope (Opteron)

Size	RT	LT	ST	PT	PS	PD	GB/s
128B	MMX	T	T	T0	2	0	0.27
8KB	XMM	T	NT	NT	2	12	4.75
256KB	XMM	T	NT	NT	2	30	6.79
1MB	XMM	T	NT	NT	2	30	6.58
4MB	XMM	T	NT	NT	4	28	6.65
32MB	XMM	T	NT	NT	2	20	6.6

Table 9.5: The register, instruction and prefetch parameters (PS/PD in CLs) of Asterope for max performance.

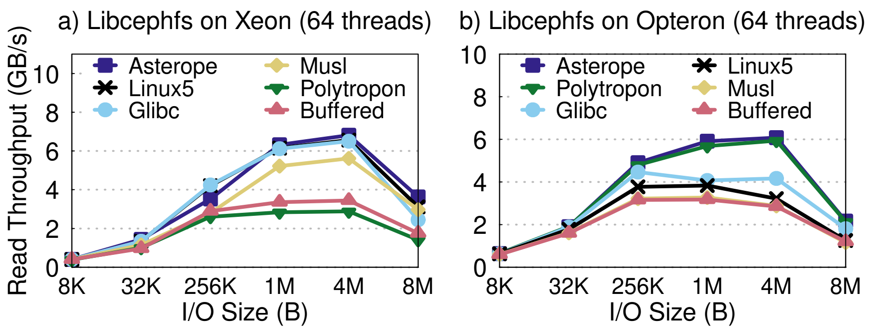


Figure 9.2: **Read I/O performance of libcephfs**. This figure depicts the read I/O performance of libcephfs over Ceph. The libcephfs that uses the memcpy routine returned by the Asterope algorithm tracks or improves the performance of the fastest memcpy routine on both systems.

9.4.3 Memory Copy in a Distributed Filesystem Client

We also examine the performance sensitivity of a distributed filesystem to the memcpy routine for data copy between the application buffers and the client cache. We consider a Ceph [24] storage cluster consisting of 6 object servers and 1 metadata server configured as VMs on a dedicated quad-socket Opteron machine. We link the libcephfs library to an application to read 100 times an entire 1GB file through the specified I/O size with 64 threads over 8 cores. We run the client benchmark on a distinct Xeon or Opteron machine (running Debian 9 Linux) with libcephfs object cache set to 32GB and accessing the Ceph cluster over a 20GB/s (bonded) Ethernet link.

Figure 9.2 shows the total read throughput of the different processor models and memcpy routines used by the client over warmed up object cache. *Asterope approximately tracks or improves the performance of the fastest routine, whether it is Glibc in Xeon or Polytropon in Opteron*. Interestingly, Asterope achieves 1.5x advantage over Glibc (Xeon/8MB, Optreron/1MB,4MB) and 1.9x over Linux5 (Opteron/4MB).

9.5 Summary

In this chapter, we introduce Asterope, a methodology to optimize common software operations through an exhaustive search across several relevant software and hardware factors. We presume that the optimization should be infrequently conducted at the software installation on a specific platform.

It is encouraging that the memory copy routines that Asterope produces track or improve the performance of the fastest routine in different systems. Additionally, they increase up to 1.9-2.4x the microbenchmark and system performance of respective production implementations (e.g., Linux kernel).

Chapter 10

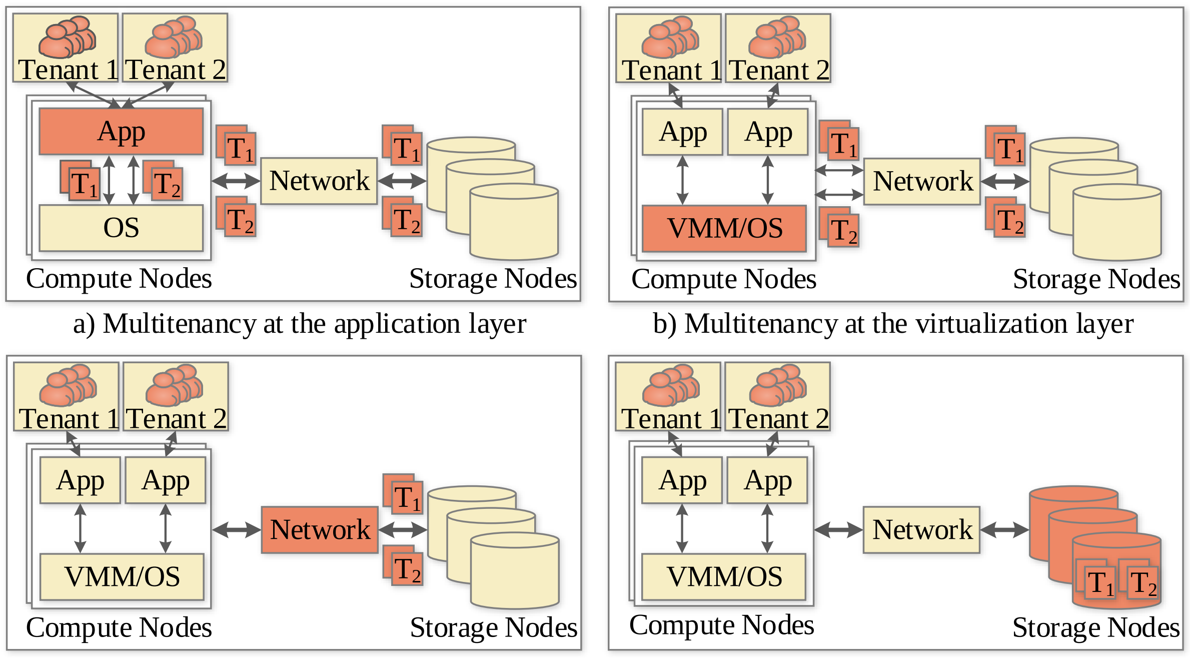
Multitenant Access Control

In this chapter, we motivate the need to natively support multitenant access control at the filesystem-level. We first discuss the layers of the cloud infrastructure software stack at which multitenant access control can be realized. Then, we focus on supporting multitenancy at the filesystem-level and we emphasize its benefits and challenges.

10.1 Background

In a cloud infrastructure that supports multitenancy, a shared storage system accommodates data from different tenants. However, tenants typically do not trust each other, and in the case of a public cloud they even do not trust the cloud provider. Thus, the cloud infrastructure must incorporate a multitenant access control mechanism in order to ensure the isolation and security of tenant data.

An access control mechanism for tenant data can be incorporated at various layers



a) Multitenancy at the application layer
b) Multitenancy at the virtualization layer
c) Multitenancy at the network layer
d) Multitenancy at the storage layer

Figure 10.1: **Levels of storage multitenancy**. This figure depicts the layers of the cloud infrastructure at which multitenant access control can be enforced. A multitenant access control mechanism can be incorporated into (a) a shared application, (b) the shared VMM, (c) the shared network infrastructure, or (d) the shared storage infrastructure.

of the infrastructure (Figure 10.1): (i) at the application layer, (ii) at the virtualization layer, (iii) at the network layer, or (iv) at the storage layer.

10.1.1 Multitenancy at the application layer

In application-level multitenancy (Figure 10.1a), a single application instance on top of a shared compute, network, and storage infrastructure serves multiple tenants. As a result, the application incorporates a multitenant access control mechanism that enforces tenant separation by isolating and controlling access to tenant data. The network and storage layers of the infrastructure are not tenant aware, but the application provides different storage partitions (e.g., folders or databases) [281, 282].

Multitenancy at the application layer maximizes the operational cost efficiency of the cloud infrastructure, because it results into the highest degree of resource sharing. However, it increases the degree of complexity of the application code and logic and the probability of software bugs and vulnerabilities [283]. It also complicates data sharing across different tenants of the same application, and precludes it completely across different applications.

10.1.2 Multitenancy at the virtualization layer

In virtualization-level multitenancy (Figure 10.1b), the virtualization layer (e.g., the hypervisor, or the operating system) is responsible to track information flow between guests (e.g., through tenant labels) and enforce access control. In this setting, each tenant uses its own application that runs in a virtual machine or a container owned by the tenant, while the virtualization layer, the network infrastructure, and the storage system are shared across all the tenants of the cloud infrastructure [284, 285].

Multitenancy at the virtualization layer incurs significant performance overheads due to the addition of isolation layers and policy enforcers at the hypervisors [101]. In addition, it does not enable flexible sharing of tenant data at the filesystem-level.

10.1.3 Multitenancy at the network layer

Each tenant may use its own application instance and rely on the shared network infrastructure to enforce multitenant access control in order to arbitrate tenant access into a shared tenant-unaware storage system (Figure 10.1c). For instance, network-level access control can be enforced by a Virtual Private Cloud based on encrypted communication channels, VLAN settings, or a firewall [286, 284, 287].

Multitenancy at the network layer provides strict isolation to tenant data. However, it was not designed to handle the scalability needs of the enormous number of tenants and virtual machines in modern cloud environments. In addition, it does not provide the required flexibility, because it cannot incorporate separate access (e.g., sharing) or performance (e.g., quality of service) policies to meet the diverse needs of different tenants [284].

10.1.4 Multitenancy at the storage layer

Each tenant may use its own application and rely on the shared storage system to enforce multitenant access control (Figure 10.1d). In fact, the storage system can be tenant aware and incorporate an access control mechanism to enforce the isolation and security of tenant data [90].

Cloud storage security could be preferably enforced at the storage infrastructure rather than the application or network level for enhanced functionality and improved resilience to programming bugs in the application or the guest system [288]. In addition, multitenancy at the storage layer enables flexible and protected data sharing across different end users, applications, and tenants [16, 289, 290].

Consolidation at the filesystem level is increasingly advocated as a desirable multitenancy paradigm because it enables improved storage efficiency across the different users and machines co-located in the datacenter [88, 89, 168, 90]. The block-based interface isolates a virtual disk into a protection domain fully controlled by the guest; this is an attractive approach due to the versioning and migration properties of virtual disks [88]. Instead, a file-based interface supports easy resource administration and optimized performance, but also provides the "killer" advantage of configurable guest isolation and sharing at fine granularity [88, 89, 90, 168, 18].

10.2 Problem Definition and Motivation

Supporting multitenant access control in the shared filesystem raises a number of isolation, efficiency, and flexibility issues, which we discuss below.

Isolation Common distributed filesystems where originally designed for enterprise, and High Performance Computing (HPC) environments. As such, they are ill-suited to meet the security challenges that are unique to cloud environments. Specifically, such filesystems provide a single shared namespace to their users for the naming of files and folders, and rely on a single identity space to enforce access control. However, if each tenant manages the identity space of its users separately, the use of a shared filesystem introduces a possibility of conflict involving the use of the same identity by users belonging to different tenants [90].

Efficiency Multitenancy at the filesystem-level without identity conflicts is supported by HekaFS [90, 291], a cloud filesystem that enables a tenant to assign identities to local principals through hierarchical delegation. HekaFS gives tenants complete freedom to choose and manage their own user identities, but eventually it maps them to globally unique identities before they reach the filesystem.

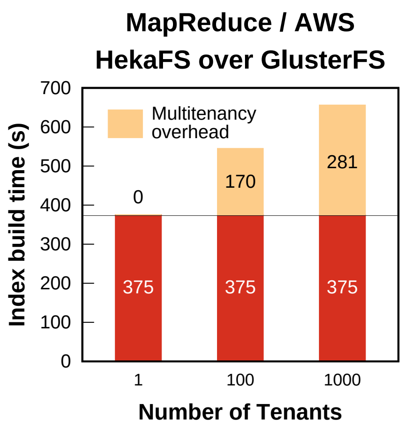


Figure 10.2: **Identity mapping overhead.** This figure shows the overhead of identity mapping added by HekaFS over GlusterFS for multitenancy support. For 1 tenant we use vanilla GlusterFS.

We experimentally motivate our work by examining the performance overhead of identity mapping currently used by HekaFS to support multitenancy over the GlusterFS distributed filesystem. We consider the Phoenix v2.0 [292] shared-memory implementation of MapReduce, and use the reverse index application to generate the full-text index of a 1.01GB file collection. In Figure 10.2, we measure the duration of index build over GlusterFS for 1 tenant, and over HekaFS for 100 or 1,000 tenants. The reverse index application only takes 375s in GlusterFS, but requires 545s (45% more) or 656s (75% more) in HekaFS configured with 100 or 1,000 tenants, respectively. From measurements of the I/O system calls, we found the increased number of tenants to be accompanied by higher latency in specific metadata operations (e.g., opendir, close) of the filesystem.

Flexibility Given the increasing popularity of filesystem storage, the OpenStack Manila is an architecture introduced to integrate filesystem shares (i.e., shared file trees) with guest machines [293]. However, the specification of the Manila protocol that connects the filesystem to the guest is work in progress. Today, cloud storage resources are typically isolated across individual tenants at the network level instead of actually being shared at the filesystem level. The OpenStack Keystone service [294] (or the AWS AIM [295]) federates the identities of different tenants (accounts), but only manages the administrative entities rather than the end users of the OpenStack (AWS) services. As a result, file sharing across end users is not possible.

Using identity mapping to support multitenancy also complicates file sharing as it requires all the storage nodes to assign global identities cooperatively and share a common mapping table [296]. In addition, users may wish to express access control that refers to identities that a given system has not yet encountered [297].

10.3 Solution: Native Multitenancy at the Filesystem Level

In this study, we explore the idea to *support native multitenant access control in the shared filesystem layer of the cloud infrastructure to improve isolation, efficiency, and flexibility*. Supporting multitenancy natively in the filesystem may increase efficiency, as it removes the need for additional mapping layers above the filesystem. Furthermore, the trend toward lightweight virtualization through containers or library operating systems can take advantage of secure storage offloading to the provider and relieve the guest machines from unnecessary complexity and overhead [54, 298]. Similarly, a thin filesystem client can efficiently integrate personal devices into cloud shared folders, especially as security features are included in the processor chipset hardware [299]. Additionally, it enables configurable isolation and sharing at file granularity which is a key requirement in cloud environments. For instance, in analytics workloads, a shared distributed filesystem often maintains the accumulated data and allows temporary local caching at compute nodes for processing. Thus, a multitenant filesystem could facilitate the storage consolidation for collaborative analytics jobs of the same or different customers [300].

10.4 Summary

Multitenancy can be supported at various levels in a cloud environment, from the tenant application, down to the infrastructure services, such as storage. Supporting multitenancy at the filesystem level is desirable, as it allows configurable isolation and sharing, and offers high performance. However, it raises a number of challenges regarding the support of the individual end-users of the tenants, instead of supporting only administrative entities, and the scalable management of possibly colluding identity spaces. Previous approaches rely on identity mapping techniques to support multitenancy at the filesystem-level and resolve identity conflicts. We experimentally demonstrate that this technique introduces high performance overheard to support a large number of tenants. In contrast, we explore the idea to support multitenant access control *natively* in the fileystem in order to imporve isolation, efficiency, and flexibility.

Chapter 11

The Dike System

In a cloud environment that serves multiple tenants (independent organizations), supporting multitenancy natively at the filesystem is desirable because it enables data sharing, administration efficiency, and performance optimizations. In this chapter, we first define the entities involved in a multitenant filesystem and present relevant security requirements. Then we introduce the design of the Dike authorization architecture. It combines native access control with tenant namespace isolation and compatibility to object-based filesystems. We introduce secure protocols to authenticate the participating entities and authorize the data access over the network. We alternatively use a local cluster and a public cloud to experimentally evaluate a Dike prototype implementation that we developed. At several thousand tenants, our prototype incurs limited performance overhead below 21%, unlike a solution from industry whose multitenancy overhead approaches 84% in some cases.

11.1 System Requirements

In the present section, we outline the general requirements of our system. We first define our goals, then we discuss our assumptions and finally we define a system trust and threat model.

11.1.1 Goals

In the proposed access control, we set the following goals:

1. **Isolation.** Each tenant is free to choose identities for its users. Thus we isolate the identity space and access control of different tenants to prevent namespace collisions.

2. **Sharing.** Provide flexible access control to enable secure file sharing within a tenant or among different tenants.

3. **Efficiency.** Natively support multitenant access control to achieve the required performance and scalability for enormous numbers of users or files.

4. **Interface.** Leverage the architectural characteristics of widely-adopted filesystems for backward compatibility with existing applications.

5. **Manageability.** Maintenance support at the file level allows the cloud provider to uniformly and flexibly manage the storage resources of different tenants.

11.1.2 Definitions and Assumptions

The *user* is an entity (e.g., individual or application) that receives authorizations and serves as unit of accountability in the system. We call *tenant* an independent organization whose users consume networked services from a cloud *provider* [17]. The *domain* generally refers to any organization, including the provider itself, that accesses the cloud resources for consumption or administration purposes. The *directory* refers to a registry of users and their attributes, while *folder* is a catalog of files in the filesystem.

Domains

D: set of domains in the system

P: set of providers in the system ($= \{p\}$ by default)

T: set of tenants in the system

$D = P \bigcup T$

$s(d)$: set of all domains sharing any file with domain $d \in D$

Users

U: set of users in the system

U_d: set of all users belonging to domain $d \in D$

$U = \bigcup_{d \in D} U_d$, with $U_i \bigcap U_j = \emptyset, \ \forall i, j \in D$

Files

F: set of files in the system

F_d^i: all files in domain $d \in D$ accessible by domain $i \in D$

F_d: set of all files owned by domain $d \in D$ (i.e., $F_d = F_d^d$)

$F = \bigcup_{d \in D} F_d$, $F_i \bigcap F_j = \emptyset, \ \forall i, j \in D$

F^i: set of all files accessible by domain $i \in D$

$F^i = \bigcup_{d \in s(i)} F_d^i$, with $s(i)$ as defined above

Permissions

$P_d^i(u, f)$: permissions of user $u \in U_i$ to file $f \in F_d$, $i, d \in D$

$P_d^i(f)$: per-user permissions of file $f \in F_d$ in domain $i \in D$

$P_d^i(f) = \bigcup_{u \in U_i} P_d^i(u, f)$

$P_d^{\bar{d}}(f)$: permissions of file $f \in F_d$ in domains $i \in D, \ \forall i \neq d$

$P_d^{\bar{d}}(f) = \bigcup_{u \in U_i, \, i \neq d} P_d^i(u, f)$

$P_d(f)$: per-user permissions of file $f \in F_d$ in all domains

$P_d(f) = P_d^d(f) \bigcup P_d^{\bar{d}}(f)$

Figure 11.1: **The basic entities in Dike.**

A *server* implements service actions, and a *client* provides local access to a service over the network. We collectively refer to the clients or servers of the system as *nodes*. Through their local client, the users of a tenant access storage services at file granularity. The user who creates a file in a tenant is the *owner user* of the file and assigns access permissions with a discretionary model. A file is owned by the tenant to which the owner user belongs. In case of a file shared across multiple tenants, the assignment of access rights in an non-owning tenant is undertaken by the administrator user of that tenant.

In Figure 11.1 we summarize some basic entities with their properties in the Dike system. The set of domains D is derived from the union of the tenant set T with the provider singleton P (extension for multiple providers left for future work). The sets of users belonging to different domains are disjoint and disjoint are also the sets of files owned by different domains ($\forall i, j \in D,\ U_i \bigcap U_j = F_i \bigcap F_j = \emptyset$). The permissions of file f owned by domain d is derived from the union of the per-user file permissions across all the domains in the system ($P_d(f) = P_d^d(f) \bigcup P_d^{\bar{d}}(f)$).

We adopt the architecture of a distributed object-based filesystem because it is scalable and typically used in cloud environments. The system consists of multiple metadata and object servers: a metadata server (MDS) manages information about the file namespace and access permissions, and an object server (or object storage device, OSD) stores file data and metadata in the form of objects. A client can only access an OSD after an MDS communicates to the OSD the necessary authorization capability (signed token of authority) for the requested access.

11.1.3 System Trust

An independent provider operates the datacenter nodes at which the filesystem clients and servers run. Multiple virtual nodes generally share a physical host. A client runs on a virtual node inside the datacenter or on an external mobile device with sufficient hardware security or virtualization support (e.g., ARM TrustZone [301], cTPM [299]). A secure protocol for clock synchronization keeps the time synchronized across the nodes of the system.

Standard cryptographic primitives ensure the confidentiality, integrity and freshness of the network communication. Secure hardware (e.g., Trusted Platform Module) at each physical machine applies static measurement to certify the integrity of the system software stack [302, 177]. A cryptographically signed statement of authenticity serves as certificate of node integrity [91]. A central monitor (*attestation server*) inside (or outside [15]) the datacenter applies static remote attestation to build up the infrastructure trust.

The distributed filesystem protects the confidentiality and integrity of stored data and metadata by permitting networked accesses to authorized users and handling the revocation or delegation of permissions. The filesystem infrastructure enforces the access policy at the granularity of individual files, although possible extension to file collections or byte ranges within a file is compatible with our design [303, 304].

The co-located tenants may not trust each other. Each tenant is responsible to specify the file access permissions of its individual users. A tenant can share data with other tenants under the access permissions specified to the provider. In the terminology of the attribute-based access control model with tenant trust (MT-ABAC [305]), we follow the type-γ trust that lets the trustor tenant control the existence of the trust relation, and the trustee tenant control the assignment of cross-tenant attributes to its own users.

11.1.4 Threat Model

The adversary can be an external attacker or authenticated user remotely attempting unauthorized filesystem access by trying to steal or generate certificates, keys and tickets, or using network attacks to eavesdrop, modify, forge, or replay the transmitted packets of the filesystem protocols. The unauthorized requests may attempt to read or tamper with the stored data and metadata of the filesystem, including the access policy that grants file operation permissions within and across tenants or the provider. The adversaries may also exchange certificates through an out-of-band channel in an attempt to bypass the sharing policy enforced by the filesystem.

In general, an employee of the provider (*honest-but-curious*) may attempt to read the filesystem data of the tenants, but does not have incentive to corrupt the filesystem state or collude with adversaries. A tenant may choose to activate data sharing with the provider for enabling security services supported by the system (e.g., anti-virus scanning). For transparency reasons, the file owner can always use the tenant view to inspect the applicable access policy including possible file sharing with other tenants and the provider.

The privileged software (e.g., hypervisors) is measured before being trusted to run system services and isolate the processes of different tenants. We do not consider runtime integrity monitoring (e.g., to prevent zero-day attacks), given the lack of established methods for dynamic remote attestation in virtualization environments [306]. We also leave out of scope the malicious manipulation through physical access or side-channel attacks (e.g., power analysis, cache timing, bus tapping). We target filesystem access control without any explicit attempt to provide general solutions to denial of service, key distribution, traffic analysis, and general multitenant sharing of resources other than storage [307].

11.1.5 Encryption and Keys

Public keys are used as identities that uniquely identify the entities of the system, such as users and services. A tenant is identified through the key of its tenant authentication service. For privacy protection, a privacy certification authority ensures the unlinkability of multiple keys referring to a single user (this technique is not currently supported in our prototype implementation) [301]. The private keys of an entity can only appear in plaintext form at the volatile memory of an attested node. They are stored persistently in encrypted form, or partitioned across multiple nodes for improved protection [21]. The interacting entities securely communicate over symmetric keys, which are agreed upon with public-key cryptography dynamically. We leave outside the scope of the present work the consideration of data encryption by the filesystem servers on the storage media.

11.2 System Design

In this section we introduce the Dike architecture of multitenant access control for networked storage shared at the file level. First, we define a hierarchical identity management scheme to provide tenants with the ability to manage their users localy. Then, we define a multitenant authentication and authorization scheme that consists of protocols that enable the secure authentication of the system entities and control the access to tenant resources. At the end of this section, we provide a detailed security analysis in order to analyze how our secure protocols can prevent tenant and provider attacks and describe potential implications.

11.2.1 Identity Management

Identity management refers to the representation and recognition of entities as digital identities in a specific domain [308]. A multitenant environment complicates the secure operation of a shared filesystem due to potentially conflicting needs from multiple independent organizations. Below we consider possible schemes of identity management for multitenant filesystems.

Centralized A common directory administered by the provider maintains the identities of users for all the tenants [309]. Such an approach lacks the required scalability and isolation because it centrally manages all the identities, and restricts the tenants from making free identity choices.

Peer-to-peer Assuming globally unique identities, different tenants communicate directly with each other to publicize the identities of their users and groups [310]. Cross-tenant file sharing is possible through direct inclusion of remote users into the access policy of a file. A drawback is the need to periodically keep up to date the contributed users and groups across the collaborating tenants.

Mapped A shared filesystem maps the identities of users to globally unique identities [90]. For instance, the global identity space is partitioned into disjoint ranges, and each range is assigned to a different tenant. Identity mappings incur extra runtime overhead because they are applied dynamically whenever a server receives a request. Moreover, granting the permissions of local-user classes to remote users opens up the way for violating the principle of least privilege.

Hierarchical In order to isolate its identity space, each tenant maintains a private authentication service to manage locally the identities of its users. The authentication services of legitimate tenants are registered with the underlying common filesystem of the provider. Requests incoming from the clients of an approved tenant are processed by the filesystem according to the stored access permissions of each file.

11.2.2 Authentication

In Dike we adopt the hierarchical identity management because it offers the isolation and scalability properties required for multitenancy. An attestation server is used to

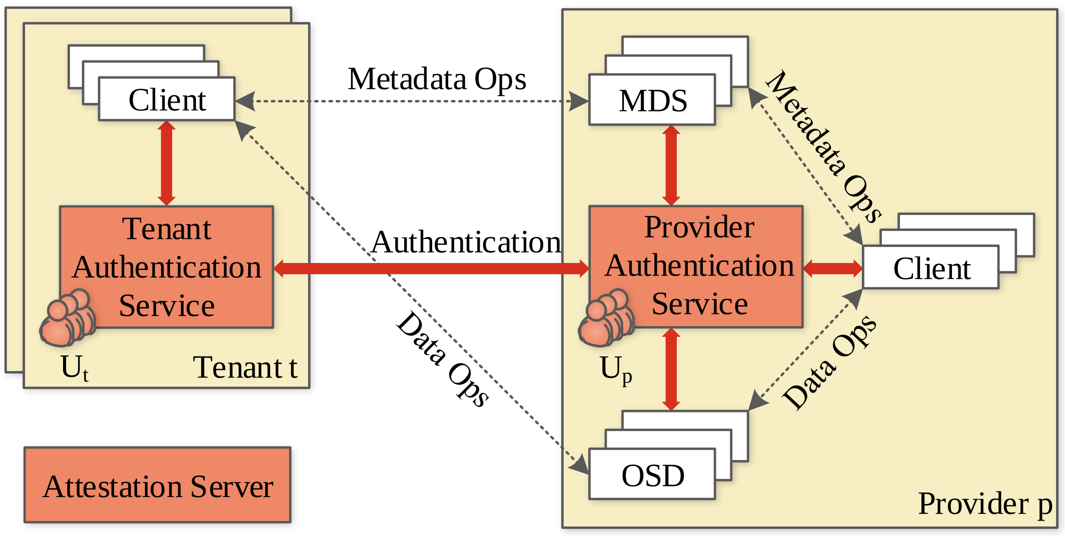


Figure 11.2: **The hierarchical architecture of Dike.** The hierarchical architecture of Dike. This figure shows the components of the Dike architecture and their interactions. The interaction of the attestation server with all the nodes is omitted for readability. We use dark background for the new entities added in Dike with respect to an existing object-based filesystem.

bootstrap the system trust (§11.1.3). We partition the task of filesystem entity authentication among the provider and the tenants (Figure 11.2). Each tenant uses a separate *tenant authentication service (TAS)* to authenticate the local clients and users. Additionally, the provider operates a *provider authentication service (PAS)* to authenticate the metadata servers, object servers and tenant authentication servers of the system. The clients (users) are distinguished into the *tenant clients* (*users*) that can only access the resources of the filesystem belonging to a particular tenant, and the *provider clients* (*users*) that are trusted to access the entire filesystem.

The user identities and the file permissions are explicitly visible to the filesystem infrastructure. This is necessary in order to run the enforcement of the access-control policy as a service of the provider rather than the tenant. For accesses of provider users the authentication process remains similar with the only difference that both the clients and users are directly authenticated by the PAS rather than a TAS.

11.2.3 Operation Example

In Figure 11.3, we show the interaction between a tenant client and the filesystem nodes in order to allow a user data access from the filesystem. We follow the involved

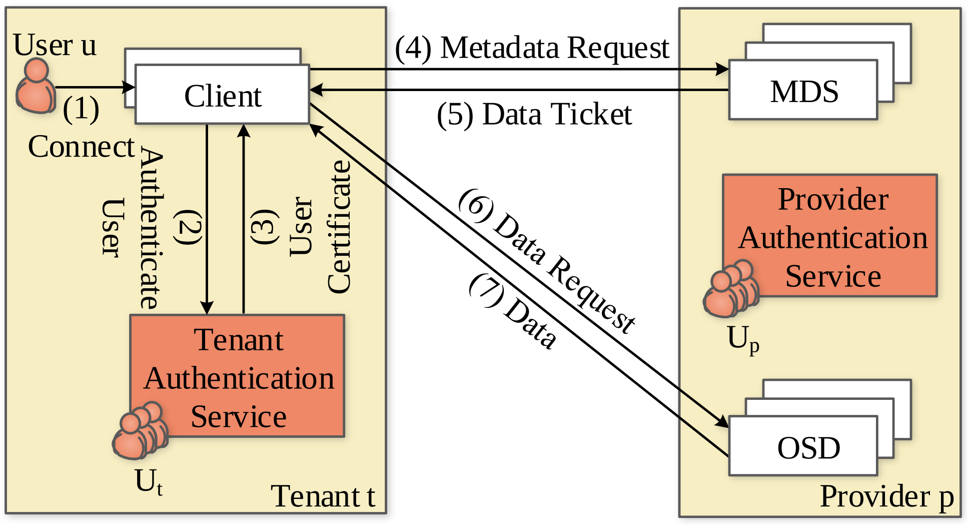


Figure 11.3: **Steps for file access in Dike.** Steps for file access in Dike. Over lines in the parentheses we enumerate the steps of file access by a user in the Dike multitenant access-control system. Important steps required for the attestation and authentication of the system nodes are omitted for readability. The new entities of Dike are denoted with dark background.

steps at a relatively high level before introducing next the secure protocols (§11.2.4). A client is initially authenticated by the TAS (not shown). The tenant user connects to the client (step 1). On behalf of the user, the client contacts the TAS (2) and receives back a user authentication certificate (3) to request filesystem access (4) from the metadata server (MDS). The MDS validates the received user authentication using a PAS-issued certificate of TAS. Then, the MDS issues to the client a *data ticket* (5) to access an object server (steps 6-7). The data ticket securely specifies the user and permissions of the authorized operation over a file.

11.2.4 Secure Protocols

Next we use secure protocols to describe the authentication (Figure 11.4) and authorization (Figure 11.5) steps in more detail. In the protocol definitions, the participating entities include the privacy certification authority v, the attestation server a, the user u, the client c, the TAS t, the PAS p, the metadata server m, and the object server o. The notation $x \rightarrow y : z$ denotes the transmission of message z from entity x to entity y.

The private and public keys of entity x are denoted as K_x^{-1} and K_x respectively.

$\mathcal{P}_0$: **Attestation by server a of entity $x \in C, T, P, M$, or O** $x \to a : \{\{K_x, H(w), N\}\, K^{-1}_{AIK(x)}\}K_{xa}, \{K_{xa}\}K_a,\ \{K_{AIK(x)}\}\, K_v^{-1}$ $a \to x : \{\mathcal{C}^a_x, N+1\}K_{xa}$ $\mathcal{C}^a_x = \{K_x, T_s, T_e\}\, K_a^{-1}$
$\mathcal{P}_1$: **Authentication by PAS p of entity $x \in C, T, M$, or O** $x \to p : \{\mathcal{C}^a_x, K_{xp}, N^{(1)}\}K_p$ $p \to x : \{\mathcal{C}^p_x, N^{(1)}+1\}K_{xp}$ $\mathcal{C}^p_x = \{K_x, T_s^{(1)}, T_e^{(1)}\}\, K_p^{-1}$
$\mathcal{P}_2$: **Authentication of client c by TAS t** $c \to t : \{\mathcal{C}^a_c, K_{ct}, N^{(2)}\}K_t$ $t \to c : \{\mathcal{C}^p_t, \mathcal{C}^t_c, N^{(2)}+1\}K_{ct}$ $\mathcal{C}^t_c = \{K_c, T_s^{(2)}, T_e^{(2)}\}\, K_t^{-1}$
$\mathcal{P}_3$: **Authentication of user u by TAS t** $c \to t : \{\{K_u, N^{(3)}\}K_u^{-1}\}K_{ct}$ $t \to c : \{\mathcal{C}^t_u, N^{(3)}+1\}K_{ct}$ $\mathcal{C}^t_u = \{K_u, T_s^{(3)}, T_e^{(3)}\}\, K_t^{-1}$

Figure 11.4: **Dike authentication protocols.** This figure lists the protocols for the authentication of a tenant, provider, server, client, and user in Dike.

We use the symbol K_{xy} for the secret key established between entities x and y. The notation $\{z\}K$ denotes the message z encrypted with the secret or public key K, or signed with the private key K. The operation $H(z)$ refers to the hashing of message z. We use the letter N for referring to a nonce, and the symbols T_s and T_e for referring to the starting and ending time of a ticket validity.

The symbols C, P, T, M and O stand for the client, PAS, TAS, MDS and OSD sets respectively. In our design a TAS serves as proxy for a tenant and the PAS as proxy for the provider. The attestation server knows the public key K_v of the privacy certification authority, and all the nodes receive the public key K_a of the attestation server. Additionally, the TAS, MDS, and OSD know the public key K_p of the PAS, and each client securely obtains the public keys K_p, K_t, K_m, and K_o.

The privacy certification authority v signs the public part of the attestation identity key $K_{AIK(x)}$ previously provisioned for entity x through a standard protocol [306]. Then, the remote attestation server a verifies the authenticity of a node (software/hardware) configuration w through protocol $\mathcal{P}_0$. If the verification succeeds, the requesting entity obtains an attestation certificate signed with the private key K_a^{-1}. Then each entity x, standing for t, m or o, uses the protocol $\mathcal{P}_1$ to get authenticated by the PAS and receive the certificate $\mathcal{C}_x^p$. A tenant client c uses the protocol $\mathcal{P}_2$ to get authenticated by the TAS t, establish a secure channel over secret key K_{ct}, and receive the certificate $\mathcal{C}_c^t$.

On behalf of user u, an authenticated client c applies the protocol $\mathcal{P}_3$ to receive a user certificate $\mathcal{C}_u^t$ signed by the TAS. The client c ensures its authenticity by holding the secret key K_{ct}, and the user u proves her authenticity by having the client sign the request with the private key K_u^{-1}. Using protocol $\mathcal{P}_4$, the client c agrees on a secret key K_{cm} with the metadata server m. Then, the client applies protocol $\mathcal{P}_5$ to request the data ticket $\mathcal{T}_{c(u)o}^m$ on behalf of user u to access file f. In addition to the public keys K_c and K_o that specify the client and the object server, the data ticket contains the handle ($handle$) of file f and the file permissions ($perms$) applying to user u.

With protocol $\mathcal{P}_6$, the client transmits to OSD o the request for operation op on the $handle$, encrypted with $K_{c(u)o}$, and receives back the reply ($opreply$). The secret key $K_{c(u)o}$ is encrypted with the public key K_o. The above steps are similar in the case of a provider user accessing the filesystem, although both the user and client are directly authenticated by the PAS rather than the TAS.

We use timestamps to ensure the freshness of the issued tickets and certificates. We detect replay attacks in the exchanged messages using nonces and having them increased by one before they are returned. Additionally, in several protocols ($\mathcal{P}_0$, $\mathcal{P}_1$, $\mathcal{P}_2$ and $\mathcal{P}_4$) we use nonces to detect potential unauthorized modification of the request and confirm the authenticity of the recipient server.

$\mathcal{P}_4$: **Secret K_{cm} shared between client c and server m**

$c \rightarrow m : \{\mathcal{C}_t^p, \mathcal{C}_c^t, K_{cm}, N^{(4)}\}K_m$

$m \rightarrow c : \{N^{(4)} + 1\}K_{cm}$

$\mathcal{P}_5$: **Data ticket for file f to client c on behalf of user u**

$c(u) \rightarrow m : \{\mathcal{C}_u^t, f, N^{(5)}\}K_{cm}$

$m \rightarrow c(u) : \{\mathcal{T}_{c(u)o}^m, N^{(5)} + 1\}K_{cm}$

$\mathcal{T}_{c(u)o}^m = \{K_c, K_o, perms, handle, T_s^{(5)}, T_e^{(5)}\}\, K_m^{-1}$

o may be replaced by object server group g for efficiency.

$\mathcal{P}_6$: **Run op on $handle$ for client c on behalf of user u**

$c(u) \rightarrow o : \{\mathcal{T}_{c(u)o}^m, op, N^{(6)}\}K_{c(u)o}, \{K_{c(u)o}\}K_o$

$o \rightarrow c(u) : \{opreply, N^{(6)} + 1\}K_{c(u)o}$

Figure 11.5: **Dike authorization protocols.** This figure lists the protocols for authorizing a filesystem request in Dike.

11.2.5 Multiview Auhorization

In the *multiview* authorization methodology that we introduce, the filesystem selectively makes the metadata accessible to different entities in the form of views. The filesystem administrator of the provider uses the *provider view* to specify permissions for entire tenants or individual users. Instead a tenant administrator uses a *tenant view* to configure the metadata made accessible by the provider to the respective tenant. A user obtains filtered access to a subset of the provider or tenant view according to the applicable permissions.

The authorization policy of the filesystem is specified in the permissions maintained for each file by the MDS. We support two types of access permissions for a file or folder, the Unix and the Access Control List (ACL). The MDS isolates on distinct data structures (e.g., linked lists) the policies of a file that apply to different tenants. It additionally stores separately the policy for the provider users.

A Dike access policy can configure a file as private or shared across the users of a single or multiple tenants and let the filesystem natively support cross-tenant accesses. The certification hierarchy in general-purpose public-key cryptography can have an arbitrary number of levels. In the past, this possibility has received negative criticism due to the potential complexity that it introduces [310]. Instead, Dike only uses a two-level hierarchical structure to let the TAS of each tenant be certified by the PAS. Dike also differs from the multi-realm Kerberos protocol, in which remote accesses require tickets of the remote realm to be granted either directly, or hierarchically through a common ancestor [311].

11.3 Security Analysis

In this section, we analyze the properties of the Dike architecture by explaining the secure steps followed by the protocols and describing the potential implications of tenant and provider attacks.

Permissions granted In Dike, the provider cooperates with the tenants to enforce the filesystem access control. Only authenticated clients and users are allowed to access the filesystem. The data is only disclosed to or modified by users authorized by the system according to the permissions specified by the owner user. A user can only access the data belonging to or shared with her authenticating tenant. Accordingly, a user operation is restricted to the minimum of the file access permissions specified in the policy of the file owning tenant and the user authenticating tenant.

Authentication enforcement The client and user are authenticated by the TAS (or PAS) before they can receive the authorization data ticket from the MDS to securely access a particular OSD. The MDS verifies the authenticity of the requesting client and user before it signs the data ticket with its private key to make any unauthorized tampering detectable. Then, the authenticated user accesses a file according to the policy specified at the MDS. The OSD uses the public key of the MDS to verify the authenticity of the data ticket and returns the requested data encrypted with the secret key agreed with the client.

Authorization enforcement The data ticket is securely transferred from the MDS to the OSD over the K_{cm} and the $K_{c(u)o}$ secret keys through the client. The OSD is responsible to check the requested operation against the permissions securely contained in the data ticket and enforce the access control specified at the MDS. The authorized file access is restricted to a specific tenant and user as requested by the protocols that establish the key K_{cm} and ticket $\mathcal{T}^{m}_{c(u)o}$. Direct filesystem accesses from the provider users similarly require user authentication from the PAS before they can establish communication with the MDS.

Sharing violation The support for file sharing between different tenants is enabled through activation of the necessary access permissions at the MDS. The clients or users from different tenants cannot collude to bypass the MDS policy enforcement because an issued data ticket specifies the authorized client and user belonging to a specific tenant. Potential cross-tenant policy violation is prohibited by design because the filesystem access is restricted through the tenant view, and the permissions of different domains are stored and managed separately by the MDS.

Tenant attacks An attacker is unlikely to penetrate the client of a tenant and impersonate a legitimate user, because a user certificate cannot be requested without access to the user's private key within an attested client. The harm from a tenant user impersonation is limited to the private or shared files that are accessible by the compromised tenant, and cannot affect the system-wide access policy. Depending on the severity of the attack, possible steps to isolate the attacker include the revocation of access permissions to the compromised user by the tenant and disabling of file sharing to the penetrated tenant by the provider.

Provider attacks The attack is more challenging in the unlikely case that it compromises an administrator account of the provider and exposes the system permissions. The implications of such an attack can be contained if the tenants apply external protection techniques or the provider supports remote monitoring from outside the cloud to detect the incident [15, 313]. More generally, in lack of trust to the provider, a tenant may externally apply techniques of encryption, hashing, auditing and multi-cloud replication to strengthen end-to-end confidentiality and integrity, or ensure data restoration in case of a provider compromise [19].

11.4 System Prototype

In this section, we describe our implementation of the Dike multitenant access control over a distributed filesystem. The prototype development is based on Ceph, a flexible platform with scalable management of metadata and extended attributes [24]. We expanded Ceph to experiment with the scalability of native multitenant access control according to the Dike design. Apart from the added multitenancy support, our current prototype implementation relies on the existing authentication and authorization functionality of Ceph. As a result, the tasks of both the TAS and PAS are currently carried out in part by the MON.

11.4.1 Overview

We developed the Dike prototype by modifying the client and the metadata server (MDS) of the Ceph. The implementation required the addition of 2023 commented C++ lines into the codebase of Ceph v0.61.4 (Cuttlefish). We extended the client interface with two new operations to grant or remove tenant access for a file or a folder (*ceph_grant_tenant_access* and *ceph_revoke_tenant_access*). These operations specify the ownership and permission on a file or folder in a way similar to *setattr*, but they additionally accept a tenant ID as an argument. For each of these two operations we also added a corresponding handler at the MDS that invokes the respective method at a CInode object to assign or remove tenant permissions. The Dike implementation required to modify all the filesystem functions of the original Ceph related to permissions handling, including the inode constructor.

11.4.2 Tenant Identification

In a filesystem mount request to an MDS, a client has to uniquely identify the accountable tenant. In our current prototype, we derive a unique tenant identifier (TID) by applying a cryptographic hash function to the public key of the tenant (we use RIPEMD-160). The client embeds the TID into an expanded MClientSession request, and sends it to the MDS over the secure channel established with the session key. The MDS extracts the TID from the received message, and stores it in the session state. The secure session between the filesystem and an authenticated client can only serve the actions permitted to the users of the identified tenant.

11.4.3 Permissions

Our current implementation only supports Unix-like permissions for users and groups, but it makes straightforward the addition of access-control lists in a future version. The Ceph version that our prototype relied on did not directly support access-control lists, but newer Ceph releases have been gradually adding this feature. Based on the supplied TID, a client obtains tenant view of the filesystem for access by a tenant user. The permissions of the tenant view are stored in the extended attributes of the filesystem. We manage the extended attributes in memory as key-value pairs stored in a C++ map structure (red-black tree). As key, we use the string tid:model:perm, in which the field tid holds the tenant identifier, the perm specifies the type of permissions, and the model is set to "UNIX" for Unix permissions or "ACL" for access control list. In the Unix model, we set the value of the key-value pair to "uid:gid:mode", where the uid and gid fields refer to the user and group identifiers, and the mode contains the file permissions. For global configuration settings, we also support the provider view, which enables full access permissions to the administrator of the entire filesystem. The respective permissions are stored in the regular inode fields.

11.4.4 Capabilities

A file capability is only sent to a client whose authenticating tenant is permitted to access the file. In order to enforce the access policy, we expanded the returned capability of Ceph to include the tenant identifier and the respective file ownership metadata. A client cannot directly read or write the access control information stored in the extended attributes of a file. Instead, only the filesystem is allowed to access the extended attributes on behalf of authorized client requests.

11.4.5 Administrative Tools

Finally, we implemented an administrator tool that combines the functionalities of the Unix *chmod* and *chown* utilities, but accepts a tenant ID as an additional argument. The tool invokes the new calls that we added to the client interface and can be used by a file system administrator to assign or revoke tenant permissions on files and folders.

11.5 Experimental Evaluation

In this section, we experimentally evaluate the performance of the Dike prototype with a microbenchmark, and two application-level benchmarks to answer the following questions. Overall, we demonstrate that Dike adds multitenancy support to Ceph at limited performance overhead and the overhead of native multitenancy is much lower than the overhead of multitenancy through identity mapping (§ 11.5.3), especially when the number of tenants rises to hundreds or thousands.

11.5.1 Experimentation Environment

We conducted several experiments on a local cluster and the Amazon public cloud using a microbenchmark and two application benchmarks. Below, we present the details about the configuration of the experimentation environments and benchmarks that we used.

Local cluster Our first testbed relies on an isolated local cluster consisting of 64bit x86 servers running Debian 6.0 Linux. We used up to a total of 11 machines: 5 machines for the filesystem nodes and 6 machines for the client hosts. Each filesystem server is equipped with 1 quad-core x86-64 CPU at 2.33GHz, 3-6GB RAM, 2 SATA 7.2KRPM 250GB hard disks, 1Gbps link and Linux v3.9.3. A server with 6GB RAM is used as MDS. From the remaining 4 servers with 3GB RAM, 3 are OSDs and 1 is MON. Each OSD uses the first disk to store the root filesystem and a journal file of 1GB size, and it has the second disk formatted with the XFS filesystem to store objects. Each client host is equipped with 2 quad-core x86-64 CPUs at 2.33GHz, 4GB RAM, 2 SATA 7.2KRPM 500GB hard disks, 1Gbps link, and runs Linux v3.5.5 with Xen v4.2.1. We set up each client guest with 1 dedicated core, 512MB RAM, 2 blktap devices for root and swap partitions, and Linux v3.9.3.

Cloud platform Our second testbed consists of EC2 instances from the US East region of the Amazon Web Services (AWS). We use a total of 36 instances: 3 instances of type "m1.large" (4x64bit cores, 15GB RAM) as fileservers, 32 instances of type "t1.micro" (1x64bit core, 615MB RAM) as microbenchmark clients, and 1 instance of type "c.medium" (2x64bit cores, 1.7GB RAM) as application client. All instances run Red Hat Enterprise Linux Server v6.4 with Linux v3.9.3. On the three fileservers we alternatively run three servers of Ceph, Dike, GlusterFS, and HekaFS with replication factor 3 (for GlusterFS and HekaFS see §11.5.3, §12.7.3). GlusterFS and HekaFS manage both data and metadata on all three fileservers. Instead, Ceph and Dike run an OSD instance on each fileserver, but also use two of the fileservers to additionally run the MDS and the MON, respectively. (Ceph uses a single MDS, because the version that we used provides unstable support of multiple active MDSs).

Benchmarks We measure the metadata performance using the mdtest v1.9.1 from LLNL [314]. This is an MPI-based microbenchmark running over a parallel filesystem. Each spawned MPI task iteratively creates, stats and removes a number of files and folders. At the end of the execution, the benchmark reports the throughput of different filesystem operations.

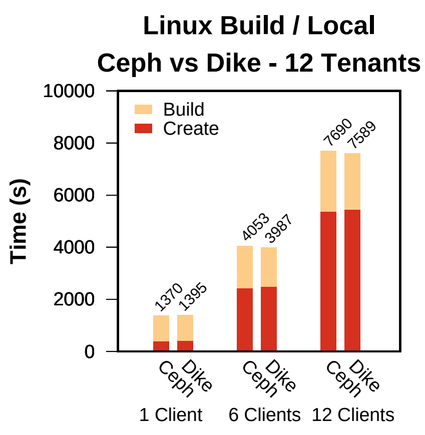
In the MapReduce application workload, we used Stanford's Phoenix v2 shared-memory implementation of Google's MapReduce. We study the reverse index, which receives as input a collection of HTML files, and generates as output the full-text index with links to the files. Our dataset contains 78,355 files in 14,025 folders and occupies 1.01GB. We measure the latency of index build broken down across several metadata operations.

In the Linux Build application workload, we store the source of the Linux kernel (v3.5.5) in a shared folder of the filesystem. Then we use soft links to make the code accessible in private folders of the tenants. We measure the total time to create the soft links and build the system image.

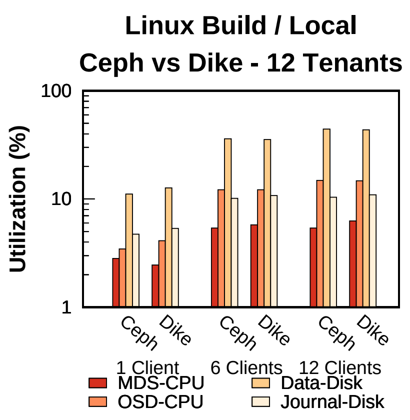
In our experiments, we treat as main measured metric the throughput of mdtest, the index build time of MapReduce, and the compilation time of Linux Build. We repeated the experiments at least 3 times and as many times as needed (up to 20) to constrain the 95% confidence-interval half-length of the main metric within 5% of the average value.

11.5.2 Client Scalability

In Figure 11.6 we compare the performance of the Linux compilation over Ceph and Dike. In Figure 11.6a, we measure separately the average time to create the soft links and build the system image. As the number of clients increases from 1 to 12, the overhead of Dike remains relatively low. In particular, it drops from 4.6% to 1.6%



(a)



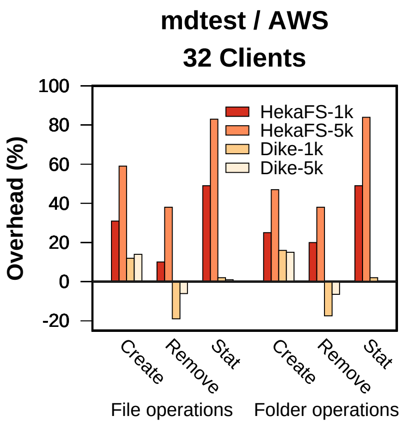
(b)

Figure 11.6: **Scalability of Dike with Linux compilation.** (a) Performance and (b) resource utilization of the Linux build workload between Ceph and Dike across different numbers of clients. We use Dike with 12 configured tenants. For the OSD case, we depict the average utilization across the three OSD nodes of the system.

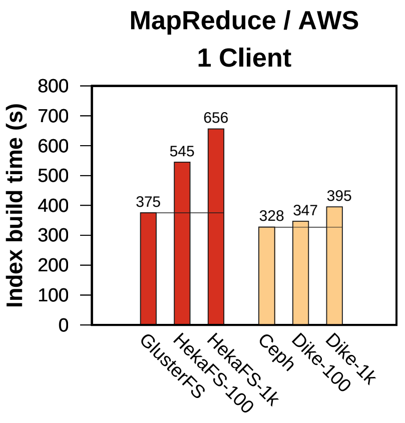
in link creation and from 0.7% to -8.1% in image build. Indeed, Dike with 12 clients takes 2,145s which is lower than 2,334s in Ceph. We notice the highest increase in the experiment duration at 1 client, as Dike takes 1,395s (1.8% higher) instead of Ceph that takes 1,370s. By looking at the resource utilization of the MDS and the OSDs in Figure 11.6b, we observe Ceph with 1 client to utilize 3.5% the CPU and 11.1% the data disk of the OSDs. Instead, Dike utilizes these two resources 4.1% (19.1% higher) and 12.7% (14.0% higher), respectively, leading to slightly longer experiment duration, as pointed out above for the case of 1 client.

11.5.3 Comparative Multitenancy Overhead

In this section we compare the multitenancy overhead of Dike over Ceph with the respective overhead of HekaFS over GlusterFS. GlusterFS is an open-source, distributed filesystem from RedHat. It supports translator layers for the addition of extra features. HekaFS is a cloud filesystem implemented as a set of translators over GlusterFS. In order to isolate the identity space of different tenants, HekaFS uses a mapping layer to translate the local user identities of the tenants to globally-unique identities [90]. HekaFS appears to strictly store the files of different tenants on distinct private folders, which makes it unclear whether it currently supports secure file sharing between tenants [90].



(a)



(b)

Figure 11.7: **Comparison of multitenancy overheads in AWS.** This figure summarizes the comparison of relative overheads by Dike over Ceph and HekaFS over GlusterFS in the filesystem throughput of (a) mdtest with up to 5,000 tenants and (b) MapReduce with up to 1,000 tenants.

We run 32 mdtest clients in the AWS testbed (1 t1.micro EC2 instance/client). The clients equally divide the creation of 48,000 files, and each client equally divides the respective number of files among its processes. We alternatively create 1,000 or 5,000 private folders and activate access permissions for a single tenant per folder to emulate 1,000 (Dike-1k, HekaFS-1k) or 5,000 distinct tenants (Dike-5k, HekaFS-1k), respectively.

Overall, the performance overhead of Dike remains similar across file and folder operations as depicted in Figure 11.7 (a). However, the overhead of file and folder create reaches 12-16% in Dike-1k and 14-15% in Dike-5k. Instead, the overhead in both file and folder stat of HekaFS over GlusterFS approaches 49% with 1k tenants and 84% with 5k tenants, i.e., nearly up to two orders of magnitude higher than that of Dike (0-2%).

In Figure 11.7 (b) we further explore the multitenancy overhead using MapReduce over AWS. We use 1 client over a filesystem configured with 100 and 1k tenants, respectively. Based on the measured time of index build, the overhead of Dike over Ceph lies in the range 6-20%, and that of HekaFS over GlusterFS in 45-75%.

We conclude that HekaFS under either mdtest or MapReduce ends up at much higher performance overhead in comparison to Dike with a moderate (a hundred) or large number (a thousand) of tenants.

11.6 Summary

In this chapter, we consider the security requirements of scalable filesystems used by virtualization environments. Then we introduce the Dike system design including secure protocols to natively support multitenant access control. In contrast to other approaches, our multitenant access control architecture allows access at file-level granularity, rather than block-level granularity. As a result, the support for tenant identity management and access control is offered natively by the distributed filesystem. Using hierarchical identity management, the task of access control is split between the tenant and the provider, with authentication services residing at both ends. The provider authentication service (PAS) is used for initial authorization of the tenant authentication service (TAS) as well as the metadata servers (MDS) and object storage devices (OSD). Once a filesystem client has authenticated with the TAS, a user can request data access. To this end, the user first authenticates with the TAS. Then the client requests a metadata ticket from the TAS on behalf of the user. Using the metadata ticket, the client requests a data ticket from the MDS and uses it to access data at the OSDs. The MDS maintains the access permissions on a per file/directory basis, allowing Dike to share data among tenants. With a prototype implementation of Dike over a production-grade filesystem (Ceph) we experimentally demonstrate a limited multitenancy overhead below 21% in configurations with several thousand tenants.

Chapter 12

Related Work

In this chapter, we review literature and systems that are related to this dissertation. First, we describe research efforts to strengthen the isolation of cloud infrastructures (§12.1). Then, we discuss related works on container storage (§12.2) and user-level I/O (§12.3). We also discuss efforts that improve interprocess communication (§12.4). In this regard, we provide a detailed review of concurrent queue algorithms (§12.5), and discuss works that optimize the memory copying (§12.6). We close the chapter by surveying previous works for security and access control in multitenant cloud environments (§12.7).

12.1 Isolation

Below we review research efforts that (i) propose library operating systems to enable the direct interaction of applications with hardware resources, (ii) explore the partitioning of system structures to improve isolation, (iii) provide methods for the dynamic allocation of resources to cloud applications, and (iv) rely on virtualization to strengthen the isolation of container applications.

12.1.1 Library Operating Systems

The Exokernel operating system architecture [153] separates the resource protection mechanisms of the operating system (*control plane*) from resource usage and management (*data plane*), in order to enable the secure management of hardware resources by untrusted library operating systems (libOSes). A thin software layer, known as the exokernel, multiplexes and securely exports the underlying hardware resources to the layer at the top. The top layer consists of user applications which are linked with libOSes. A libOS implements the system abstractions that the application needs during its life-cycle and has direct access to granted hardware resources. To this end, the performance of applications benefits from the reduced number of kernel crossings during I/O access.

Arrakis [63] adopts the exokernel architecture, but proposes more lightweight libOSes than the original exokernel. It relies on modern hardware that supports I/O virtualization in order to remove from the libOS heavyweight mechanisms, such as virtual memory and processes. As a result, the libOS, which performs the role of the data plane, implements only the network stack, the filesystem, and the necessary user-level device drivers to directly access the hardware resources. The exokernel acts as the control plane and assigns device partitions to applications. Each application consists of a single process and links with a libOS in order to directly access hardware resources.

IX is similar to Arrakis but its main focus is on network I/O. It proposes the implementation of a custom network stack inside a libOS, which the linked application can use to directly access a network device that supports hardware-based virtualizaiton.

Drawbridge [315] reduces the size of the libOS further by removing device drivers, network protocols, and thread management and relying on the host OS for such functionalities. To this end, the libOS includes a small part of the operating system interface to support unmodified applications, as well as the necessary application libraries and a Platform Adaption Layer (PAL), The PAL is a software component that lets the libOS to communicate with the underlying host OS, in order to request system services.

A different point in the design space treats the VMM as the control plane instead of the operating system [135, 137, 138, 140, 139]. Above the VMM lie the lightweight VM guests, which are composed of a single-process application linked to a library OS. The library OS is known as a *unikernel* and provides only the basic system services to the application. A lightweight guest can be constructed by selecting a user application and performing dependency tracking from source code and configuration files in order to eliminate unnecessary features from the target VM. The application VM will ultimately contain the necessary network services, the modules explicitly referenced in the configuration files, and other applications which must work together (e.g., the database backend of a web server), all linked as libraries into the application. To change the application configuration later, the whole unikernel VM must be recompiled.

The application and its system libraries can be reimplemented from scratch in a type-safe language (e.g., OCaml), to eliminate unnecessary legacy software layers (e.g., Mirage [137]). However, such an approach cannot support legacy applications. A different approach (e.g., OS^v [140]) is to emulate the interface of a commodity OS (e.g., Linux), and port system components, such as the filesystem, the network stack, and the threading mechanism. Such an approach can support legacy applications without any modifications. In both approaches, the application shares the same address space with the library OS and hence there is no notion of kernel-space. This allows the application to interact with the assigned resources directly.

The above approaches either restructure the traditional filesystem architecture (e.g., Exokernel, Arrakis, IX), or rely on hardware-level virtualization (e.g, Unikernels), in order to allow user-level applications to directly access hardware resources. A common drawback of these approaches is that the applications require modifications to run due to either the lack of multiprocess support or the reliance to high-level languages (e.g., OCaml). In contrast, our work targets unmodified applications that run in containers on commodity operating systems.

12.1.2 Partitioning

Operating systems and hypervisors require substantial tuning (e.g., swappiness [51], cgroup [50]) to overcome the performance interference among colocated data-intensive containers or virtual machines (VMs) [45, 110]. Serverless functions run on containers to hide the server management, but suffer from I/O performance degradation when colocated on the same VM [316]. The network stack processing has been isolated across containers through novel kernel resource accounting (Iron [317]), or offloading over message queues to a kernel thread (IsoStack [318]).

System isolation is improved with complex component partitioning of the hypervisor (LightVM [111]) or kernel (VirtuOS [319]). FlexSC introduced the exception-less system calls to improve multicore processor efficiency in system-intensive workloads [66]. Heracles uses processor hardware and Linux kernel methods to colocate latency-critical and best-effort tasks [108]. The above approaches improve isolation with hardware-based or kernel-based partitioning instead of the user-level functionality per container pool of Danaus. The gVisor sandbox limits the system API attack vector with an isolated user-space kernel, but currently at substantial I/O performance cost [183, 202].

IceFS [203] disentangles the data structures of a local filesystem into isolated cubes that share no physical resources, access dependencies or transactions in order to provide fault isolation. An existing filesystem must be modified to use the cube abstraction of IceFS. The cube abstraction localizes the server failures and recoveries to achieve the fault isolation among tenants at the server machines. However, server-side tenant services still rely on shared system components, such as the VFS and the page cache for I/O access. Additionally, IceFS does not isolate the filesystem clients of a distributed filesystem at the application host. In contrast, our Polytropon toolkit reuses library-based filesystems at the user-level in order to provide isolated I/O services to both the client and the server hosts of tenant applications. Polytropon removes the VFS layer completely from the I/O path and moves the caching layer at user-level separately for each pool. As a result, a bug on the filesystem of a pool cannot affect other pools. Furthermore, our Danaus client architecture uses the Polytropon components to isolate the tenant I/O at the container hosts.

MultiLanes [104] uses file-backed virtualized block devices and partitions the VFS data structures to reduce the container contention over a shared local filesystem. However, it does not provide any solution for the Linux dirty page flusher threads that do not respect the cgroup resource limits. Additionally, Multilanes cannot provide fault isolation as the containers still share the same codebase for I/O handling. Instead, Polytropon isolates the storage I/O traffic of the tenants through independent filesystem services running on dedicated resources at user level.

12.1.3 Dynamic Resource Allocation

The PARTIES [320] is a cloud runtime system that monitors the execution of co-scheduled workloads and dynamically allocates hardware resources for latency QoS. Although, it does not guarantee the fair allocation of system services, such as the page cache, or low-level system resources, like kernel locks, the dynamic hardware allocation or server filesystem partitioning could provide complementary end-to-end isolation benefit to our framework.

12.1.4 Virtualization

X-Containers [321] target the security isolation of single-concerned containers offering binary compatibility and concurrent multiprocessing. The processes of an X-Container run at the same privilege level with the Linux-based X-LibOS guest and rely on the Xen-based X-kernel for managing page tables and context switching. Their advantages are shown in a comprehensive comparison with several cloud-native systems. LXDs [71] safely run the kernel device drivers in isolated domains, while Hodor [70] achieves efficient intra-process library isolation with user-level protection domains. Graphene [105] is a library operating system (libOS) that enables multiprocess POSIX state sharing over streams of remote procedure calls. The security model uses a reference monitor to allow communication within a sandbox of mutually trusted processes with narrow ABI.

The above approaches target the safe sharing or isolation of the system state for flexible extensibility or multiprocessing. In contrast, the Polytropon toolkit that we introduced targets the I/O performance isolation of container pools from different tenants. We use the filesystem services to run data-intensive applications on a stock kernel without hardware virtualization or sandbox protection overheads.

12.2 Container Storage

Slacker relies on snapshot/clone operations of an NFS shared network filesystem to efficiently serve container images through the Docker daemon, and uses a bitmap inside the host kernel to deduplicate the client cache [185]. TotalCOW uses block address caching and data comparison to avoid I/O and cache inefficiencies of container images [109]. Both systems target the efficient shared image deduplication of container clones over block-based volumes lacking native support for container isolation or file-level sharing. The PolarFS distributed filesystem achieves low latency in cloud databases with lightweight network and I/O stacks [322]. Wharf obviates the image registry by placing the image layers on a shared NFS storage backend [186]. Danaus takes advantage of data sharing at both the client host and the storage backend to improve the container I/O efficiency.

12.3 User-level Filesystems and I/O

Below, we review related research and systems that explore the handling of I/O at user-level. First, we explore state-of-the-art user-level frameworks that enable the execution of filesystem code at user level, and then we focus on library-based I/O.

12.3.1 Frameworks

The Filesystem in Userspace (FUSE) is a widely used user-level filesystem framework that enables the execution of a filesystem as a user-level process. FUSE consists of a kernel module that registers a FUSE filesystem with VFS, and a user-level library (libfuse) that is used to implement a user-level daemon that executes the filesystem. Applications communicate with the FUSE-based filesystems through the kernel, while the filesystem code executes at user-level. The filesystem requests reach the kernel-level FUSE through the VFS, which inserts them into a queue. The FUSE kernel module makes this queue accessible to the user-level FUSE library through the */dev/fuse* virtual device. The user-level filesystem daemon reads the */dev/fuse* file to extract a filesystem request from the queue, processes it, and writes the response back to */dev/fuse*. The FUSE kernel module picks the response and replies to the application through VFS. FUSE [48] was shown to degrade the performance of some applications up to 83% due to multiple user-kenel crossings, despite the maturity from several optimizations. The applications register request handlers inside the kernel with ExtFuse [323] to meet specific functionality and performance goals. DirectFUSE [212] avoids the FUSE kernel overheads by directly linking a process to the FUSE filesystem libraries through libsysio.

Rump [196] consists of libraries running kernel filesystems at user level accessed with linking or through the kernel. The SFS [195] toolkit enabled the user-level implementation of filesystems accessed through in-kernel NFS clients. The FiST [324] language specifies a filesystem functionality at high level and automatically generates the kernel source code.

Instead of routing the I/O traffic through VFS that the above systems do, Polytropon uniquely targets isolated filesystems accessed and running at user level using the libservices abstraction of per-tenant user-level shared services.

12.3.2 Library-based I/O

Several filesystems provide safe direct access to local storage and network devices through a user-level library. For instance, a library is used to intercept the application I/O calls and communicate with a user-level router to virtualize the container network (SlimSocket [325], FreeFlow [326]). In non-cache-coherent multicores, an application communicates over RPC with the Hare local filesystem, but accesses the buffer cache through shared memory [199]. Aerie and SplitFS implement local filesystems for storage class memory with user-level libraries [198, 77]. Arrakis provides direct device access to applications through hardware virtualization [63]. Graphene executes multiprocess applications using shared POSIX abstractions over libOS [105]. DAFS enabled local file sharing over RDMA with a user-level non-POSIX API [197]. The Linux Kernel Library (LKL) packs the Linux kernel into a user-level library and allows applications to transparently use the Linux kernel code at user-level for direct access to network and storage devices.

The above works lack multitenant container I/O support. Instead, the Polytropon toolkit can build isolated I/O stacks per tenant at both the client and the server host, rather than virtualizing the fast local I/O devices or managing I/O for microkernels.

12.4 Communication

The Linux io_uring [327] is an asynchronous I/O interface with two ring buffers to submit I/O requests and receive completion events from the kernel. SkyBridge [69] takes advantage of hardware virtualization to safely switch the physical address of the page table. dIPC [328] uses a tag to specify the domain of each page and a list of tags for the accessible pages of a domain. The efficiency of zero-copy transfers through page remapping has been previously explored in network packet communication with results depending on the message size and the operation side [329, 330]. We explored reducing the data copies from two to one with the CMA method but obtained lower performance leaving the zero-copy case for future work. IOFlow [42] applied queueing rules between the virtual machines and shared storage to enforce end-to-end flow policies. Mercury [235] is a generic RPC interface supporting the native network transport protocols of HPC. FlexSC [66] introduced the exception-less system calls to improve multicore processor efficiency in system-intensive workloads. The Raven [331] microkernel used a message queue over user-level shared memory and semaphores for client-server communication without kernel data copying. Instead, the Polytropon IPC distinctly optimizes the communication for user-level endpoints and uses multiple ring buffers from a process to a filesystem service with futex-based synchronization [221].

12.5 Concurrent Queues

In this section, we provide an extensive review of state-of-the-art algorithms that implement FIFO and relaxed queues, as well as pool and bag data structures.

12.5.1 FIFO Queues

The Concurrent Ring Queue (CRQ) [229] is an array-based queue that denies to insert an item when declared closed and uses an index field to identify the allowed FIFO requests. The Linked Concurrent Ring Queue (LCRQ) is a linearizable lock-free FIFO queue implemented with a linked list of CRQs. LCRQ creates a new CRQ to insert an item when the current CRQ is closed. The linearizable wait-free FIFO queue (WFQ) [231] connects the threads in a ring with each dequeuer pointing to an enqueue and a dequeue peer. A dequeuer follows a slow path to help complete the unsuccessful fast-path enqueue or dequeue requests. The Broker Queue is a linearizable blocking FIFO queue optimized for fine-granularity parallel work distribution on a GPU [228]. It applies FIFO ordering by connecting the enqueue and dequeue operations through unique tickets.

A partial method waits for a precondition before it can proceed. Scherer and Scott [262] model such an operation with a request and a follow-up method, each with its own invocation, response and linearization point. The authors implement a dual data structure with nonblocking operations and reservations for partial requests. The PTLQueue [263] is a multiproducer/multiconsumer queue implemented over an array with power-of-two length and partial operations with tickets for strict FIFO ordering.

Michael and Scott introduced a non-blocking lock-free queue and a blocking two-lock queue, both based on a linked list of dynamically allocated nodes [226]. The lock-free queue has been criticized for low performance due to the compare-and-swap (CAS) retry, and the two-lock queue for limited concurrency due to locking. The Scalable Baskets Queue (SBQ) [332] reduces the CAS contention of Michael-Scott's queue in two ways. It uses transactional CAS (TxCAS) for advancing the tail during enqueue, and adds the items with failed tail advancement to a basket data structure of the current tail node. SBQ is up to 1.6x faster than the state-of-the-art WFQ algorithm.

Shann et al. [333] introduced a linearizable queue algorithm based on a finite array to avoid the list memory management. The algorithm uses CAS for atomic updates and a modification counter to track update versions. Tsigas and Zhang [268] reduced the CAS execution frequency in a non-blocking array-based queue by letting the head and tail pointers lag behind the actual head and tail of the queue. Shafiei [269] includes counters in the queue indexes for their correct update with a CAS instruction. Evesquoz [334] introduces non-blocking array-based queues whose atomic operations use the CAS or LL/SC (load-linked/store-conditional) hardware instructions over pointer-wide variables. The CC-Synch [335] algorithm applies the combining technique in cache-coherent multiprocessors. The algorithm uses atomic instructions for low synchronization cost but performs serial work that is linear in the number of threads.

12.5.2 Relaxed Queues

Several structures provide relaxation guarantees. The Segmented Queue [249] is a linked list of segments implemented with fixed-size arrays. The enqueuer and dequeuer search at the last and first segment for empty cell or available item. A k-FIFO [191, 336] queue is a bounded array or unbounded list of k-segments, which allows up to k enqueue and k dequeue parallel operations. The Distributed Queue [250] (DQ) is an array of Michael-Scott FIFO queues, called partial queues. An operation is performed on a partial queue selected from d random queues (d-RA), round-robin using b counter pairs (b-RR), or a least-recently-used (LRU) estimation from the head and tail counters. The d-RA structure is linearizable to *pool* semantics, while b-RR, LRU to a k-relaxed queue. Local linearizability [251] permits thread-local copies of the data structure and limits the consistency requirement to the thread-local inserted items. A two-dimensional design framework [252] defines k-out-of-order data structures by allowing a number of data structure copies for parallelism and a number of consecutive operations per copy for locality. For k up to 10^4, the evaluation measures the tradeoff between throughput and error distance from a sequential linked list. The k-LSM [337] relaxed priority queue is implemented from a shared log-structured merge tree (LSM) and additional thread-local ones merged to the shared when they reach size k. Scal [272] is an open-source benchmarking framework implementing workloads and many concurrent data structures for comparative evaluation.

The MultiQueue is a collection of multiple queues that inserts an item to a random queue and returns the highest-priority item from a few randomly picked queues [338, 230]. The SprayList [339] is a skiplist that relaxes the path restrictions through which a thread picks an item to lazily remove. As measure of computational power, the consensus number (CN) of an object refers to the largest number of processes that can decide the same value using any number of the object and atomic registers. Shavit and Taubenfeld [340] show that CN = 1, 2 or ∞ if a queue is relaxed with respect to insert location, dequeue support, or peeked item order. ZMSQ is a relaxed priority queue combining a pool with a relaxed blocking heap (mound) [341]. The extractMax operation returns a high-priority item within a bounded number of consecutive calls. Instead of probabilistically relaxing the item priority in the applied operations, RCQ maintains strict sequential order in the operation assignment, but allows the operations to complete concurrently.

12.5.3 Pools and Bags

The pool is a data structure implementing a collection of unordered objects [59]. The ED-tree [342] is a distributed pool structure that combines multiple levels of elimination-diffracting balancers with a collection of lock-free queues. A linearizable lock-free bag data structure considers the support of the empty boolean operation more important than the insertion order [343]. The MultiLane is a blocking concurrent multiset implemented as a collection of blocking concurrent sub-collections [344]. The structure is not FIFO even when the sub-collections are. Although a concurrent pool or bag can achieve high throughput, it provides no notion of wait fairness. Instead, RCQ favors the earlier enqueue and dequeue operations by sequentially assigning them over the structure elements before serving them. As previously suggested [227, 345], we consider workloads with different operation balances generated by threads dedicated to enqueues or dequeues rather than only operation pairs.

12.6 Memory Copy

Alappat et al. [346] evaluate different cache levels, dynamic replacement policies, and streaming instructions. A profiling method estimates the percentage of unnecessarily reduced frequency due to AVX2/AVX-512 workloads [254, 347]. Ainsworth [81] examines automated compiler-generated software prefetching and programmable hardware prefetching for complex data structures. NightWatch [82] uses online locality monitoring to restrict the cache size of cache polluters. Rus et al. [78] estimate the data reuse distance to identify string operations benefiting from non-temporal instructions. Lee et al. [80] found software prefetching more effective than hardware across different access patterns.

The non-temporal move instructions improve the performance of cross-socket communication [83]. Nemesis optimized the memcpy of different message sizes with string copy and non-temporal store [84]. The memcpy operation was previously offloaded to special DMA channels that transfer data in L2 block units [85]. Sears increased the cache locality of memcpy with non-temporal prefetching [223]. Early HPC research emphasized the relevance of memory bandwidth enhancements (e.g., prefetching) due to the fast-growing processing performance [79].

12.7 Security and Access Control

Below, we review previous works that propose mechanisms for the security and access control in distributed filesystems, virtualization, and cloud storage.

12.7.1 Distributed Filesystems

All the principals of a distributed system are often registered into a central directory (e.g., Kerberos [311]). Secure data transfer between clients and storage through an object-based interface is addressed in the Network-Attached Secure Disk model [348]. The Self-certifying File System prefetches and caches remote user and group definitions to flexibly support file sharing across administrative domains [310]. Plutus applies cryptographic storage to support secure file sharing over an untrusted server [349]. It aggregates into filegroups the files with identical sharing attributes, and lazily revokes readers by deferring the re-encryption of files until their update. In trust management, a credential binds the keys of principals to the authorization to perform certain tasks [92]. The users from different institutions are mapped to an authoritative list of identifiers in order to maintain uniform ownership and permission metadata over Lustre [350]. Hypergroup is an authorization construct that was introduced for the scalable federation of filesystems in grid environments [351].

In object-based filesystems, an authenticated principal obtains a capability from the metadata service to access an object server [24]. Secure capabilities based on public-key cryptography were previously shown to achieve scalable filesystem authorization [352]. The extended capability of the Maat protocol securely authorizes I/O for any number of principals or files at a fixed size through Merkle hash trees [303]. IDEAS applies inheritance for scalability in role-based access control [353]. Secure protocols have been proposed to enforce authentication, integrity and confidentiality at the data traffic of the Google File System [354]. However, the above research does not directly address the multitenancy of consolidated cloud storage.

12.7.2 Virtualization

Multitenancy isolation without sharing support was previously achieved in the filesystem by either running separate virtual machines per tenant, or shielding the filesystem processes of different tenants [101]. File-based storage virtualization is enabled by Ventana through versioning and access-control lists (ACLs) [88]. Server file ACLs have system-wide effect, while guest file ACLs are controlled by the guests. A shared proxy server at the host provides file access to local guests from networked object servers. Ventana serves multiple virtual machines, but without tenant isolation.

VirtFS uses a network protocol to connect a host-based fileserver to multiple local guests without isolating their respective principals [89]. The system stores the guest credentials either directly on the file ACLs or indirectly as file extended attributes. The scalability of Ventana and VirtFS is limited by the centralized NFS-like server running at the host. Instead we advocate the networked access of a scalable distributed filesystem (e.g., Ceph) directly by the guests.

The Manila File Shares Service is an OpenStack project under development for coordinated access to shared or distributed filesystems in cloud infrastructures [293]. The architecture securely connects guests to a pluggable storage backend through a logical private network, a hypervisor-based paravirtual filesystem, or a storage gateway at the host. Dike is complementary by adding multitenancy support to Ceph for natively isolating the different tenants at the storage backend.

Bethencourt et al. investigated the realization of complex access control on encrypted data with the encrypting party responsible to determine the policy through access attributes (ciphertext-policy attribute-based encryption) [355]. Instead, we focus on the native multitenancy support by the filesystem.

Pustchi and Sandhu introduced the multitenant attribute-based access control model (MT-ABAC) and considered different types of attribute assignment between the trustor and trustee tenant [305]. Ngo et al. introduce delegations and constraints for MT-ABAC over a single and multiple providers [356].

Our work is complementary to the above studies because we introduce entity definitions and protocol specifications for a cloud filesystem, and develop a system prototype for the comprehensive experimental evaluation of the multitenant access-control scalability in data storage.

12.7.3 Cloud Storage

Different tenants securely coexist in HekaFS [90]. A tenant assigns identities to local principals, and translates the pair of tenant and principal identifier to a unique system-wide identifier through hierarchical delegation. However, HekaFS complicates sharing and applies global-to-local identity mapping that was previously criticized as cause for limited scalability [309]. We experimentally measure the overhead resulting from the multitenancy support added by HekaFS over GlusterFS.

Existing cloud environments primarily apply storage consolidation at the block level. Guests access virtual disk images either directly as volumes of a storage-area network (SAN) or indirectly as files of network-attached storage (NAS) mounted by the host [357]. The S4 framework extends Amazon's S3 cloud storage to support access delegation over objects of different principals via hierarchical policy views [289]. CloudViews targets flexible, protected, and performance-isolated data sharing. It relies on signed view as a self-certifying, database-style abstraction [16].

Our support for fine granularity of access control by the provider distinctly differentiates our work from existing systems. In particular, the AWS Elastic File System [290] only allows cross-tenant access at the granularity of entire filesystems (e.g., mount to a client) rather than files and individual users.

The abstraction of Secure Storage Regions enables the limited storage resources of a Trusted Platform Module to be multiplexed into persistent storage [358]. HAIL combines error-correction redundancy with integrity protection and applies MAC aggregation over server responses to achieve high availability and integrity over distributed cloud storage [14]. CloudProof allows customers of untrusted cloud storage to securely detect violations of integrity through public-key signatures and data encryption [359].

Secure Logical Isolation for Multitenancy (SLIM) has been proposed to address end-to-end tenant isolation in cloud storage. It relies on intermediate software layers (e.g., gateway, gatekeeper, guard) to separate privileges in information access and processing by different cloud tenants [103].

The data-protection-as-a-service architecture introduces the secure data capsule as an encrypted data unit packaged with security policy; the execution of applications is confined within mutually-isolated secure execution environments [18]. Excalibur introduces the policy-sealed data as a trusted computing abstraction for data security [360]. Shroud leverages oblivious RAM algorithms to hide access patterns in the datacenter [361]. We also target secure storage in the datacenter, but with native multitenancy support at the file level through the access-control metadata of object-based fileservers.

12.8 Summary

The systems community has proposed a number of solutions to improve resource isolation. Works like IceFS and Multilanes propose kernel structure partitioning but the static partitioning introduces performance overheads and there is a high engineering effort to refactor the system kernel. Works like PARTIES explore the dynamic resource allocation to containers, but they do not guarantee the fair allocation of system services (e.g., page flushing), and low-level resources (e.g., locks). Finally, works like LightVM and X-Containers propose lightweight hardware virtualization which complicates dynamic resource allocation.

User-level functionality has been explored extensively to achieve several goals, including reliable microkernel modularity, safe system extensibility, and fast access to network or storage devices. Today, the virtualization of the operating-system interfaces opens up the opportunity to introduce new system abstractions that combine the multitenant configuration flexibility with the bare-metal performance of resource-demanding applications. Although there is a wealth of past experimental work on improving fault and performance isolation with kernel and hardware techniques, there is a shortage of user-level approaches to natively support the required isolation of multiprocess storage I/O in container-based infrastructures. In addition, the community has introduced various frameworks that run the filesystem at user level for easier software development, reuse of kernel code, improved security, and fast device access. Nevertheless, they do not consider the storage needs of multitenant containers.

The community has proposed a large number of FIFO queues that use sophisticated methods to handle concurrency, like the dynamic creation of new queues to insert elements fast, and the use of thread helping techniques to complete pending operations. Priority relaxation has been also extensively studied. Related works achieve configurable relaxation by maintaining multiple lock-based queues and insert an item to a random queue, or pick a few queues to remove the highest-priority item from. However, a strict priority policy, the maintenance and handling of multiple queues, and the steps needed to bound relaxation, limit the concurrency of these algorithms.

Security and access control has been also extensively studied in the context of distributed filesystems, virtualization, and cloud environments. However, current distributed filesystems either do not support multitenancy or support it with non-scalable techniques.

Chapter 13

Future Work

In this chapter, we outline directions in which our work could be extended in the future. These fall into four main categories, the dynamic resource allocation to container pools (§13.1), the relocation of kernel services to user-level (§13.2), the exploration of possible applications and extensions of our relaxed queues (§13.3), and cloud security (§13.4).

13.1 Dynamic Resource Allocation

The partitioning of host resources across different tenants trades the resource utilization for improved isolation. However, the static resource allocation is becoming a burden for both the tenants and the provider as enterprise workloads are becoming more and more heterogeneous, while the demand is usually uncertain [31]. Hence,

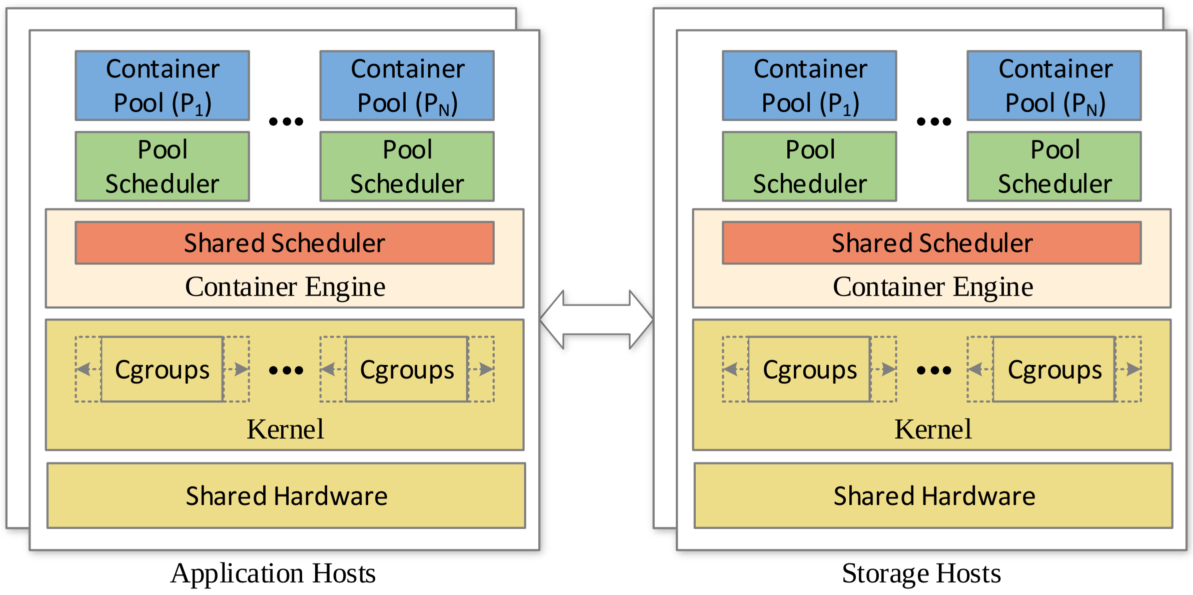


Figure 13.1: **Dynamic resource allocation.** This figure illustrates a possible architecture for dynamic resource allocation to container pools using two-level user-level schedulers.

the requirement for dynamic resource allocation is becoming important for both economical and performance reasons. Tenants should pay and grant a system resource only for the time the resource is actually needed for the execution of their workload. At the same time, the provider would be able to achieve high resource utilization and meet the needs of more tenants, in order to maximize its profit [20].

Currently, resource scheduling is performed by the host kernel which allocates resources to containers with cgroups. We are particularly interested to investigate the splitting of this centralized scheduling functionality into two layers. The bottom layer of the scheduler may lie in the container engine or in the kernel and will have the responsibility to provide up-to-date information about the available resources and verify the pool credit. The top layer may run independently for each pool, with the responsibility to observe the current resource conditions and claim resources at fine-granularity when need arises (Figure 13.1). Thus, the bottom layer will be merely a reliable information provider about the status of resources in the host. Instead, the pool schedulers will collectively claim resources in a non-reputable way and will be subsequently charged by the provider for what they get.

To make the above approach feasible, we have to identify a concurrent data structure for sharing allocation state between the schedulers. Given the competitive nature of resource allocation, we need a decentralized method to let the tenant pools claim resources without compromising the system security through unapproved resource allocation, operation disruption, or other malfunction. In that direction, we should explore the possibility that the pools collectively handle resource pressure, instead of independently undertaking costly measures such as migration. While current state-of-the-art shared-state scheduling schemes consider the application schedulers trusted, they makes it hard to enforce global properties, such as capacity and fairness.

The local resource and device management at the host level is a critical operation. The allocation of hardware resources reserves the cpu shares, memory space, local storage space, and local device I/O bandwidth of the tenant application and server containers. It is a pragmatic approach to let the host kernel responsible for the hardware allocation using mechanisms, such as the Linux device drivers and cgroups (v1/v2). Furthermore, the resource scheduling services provided by the kernel (e.g., cgroups) could potentially also run at user level to enable the enforcement of flexible tenant policies [159]. A pool manager at each host can monitor the allocation of the host resources to pools using system-level information, like the Pressure Stall Information (PSI) provided by the Linux kernel [362]. The monitored allocation of the pool resources could enable their non-trivial overcommitment for reduced tenant cost and improved elasticity.

13.2 User-level Services

In this dissertation, we explored the feasibility of moving data-intensive I/O system services from the kernel to user level and provide distinct user-level services to the colocated tenants, in order to allow configuration, performance and security isolation, and flexibility. While in our prototype implementation we created libservices for a distributed filesystem client and a union filesystem, we can also build libservices for local datastores, network stack functions, and container storage components (e.g., image registry). Additionally, supporting network block volumes and local filesystems (e.g., LKL [211]) with libservices is interesting future work.

Furthermore, we can relieve the host kernel further by moving additional kernel services at the user level. By reducing the functionality of the shared kernel we effectively reduce its TCB and attack surface. Also, by reducing the interposition of the kernel during application I/O access, we avoid both indirect software overheads (e.g., due to mode switches) and the contention for shared resources. As a result, the container applications can benefit from stable and high performance.

A potential approach is to take advantage of the hardware virtualization features that are included in modern processors and I/O devices. Such devices perform the virtualization in hardware, and provide multiple virtual interfaces that can be directly and securely assigned to guests. This effectively separates data-plane from the control-plane, and removes the need for kernel intervention during I/O operations. For instance, the host network card could be split across different container pools with hardware virtualization. The filesystem service of each pool may incorporate a libservice that implements the user-level network stack [214] on top of another libservice which implements a user-level network device driver for device access.

At the same direction, a number of kernel-level functions for memory and process management could be implemented at user-level. Examples include the *mmap* function that maps files or devices into memory, and the *exec* function that loads an application binary from disk to replace a process image.

User-level file deduplication is also an interesting topic for future research. Danaus deduplicates cloned images at the file level but does not prevent the copy-on-write of entire modified files. As future work, the backend servers could support block deduplication of individual files in communication with the filesystem client to also prevent cache duplication at the host. Slacker relies on a Tintri VMstore server for server-side block-level deduplication, but it adopts a client kernel-based approach at the client to deduplicate entire volumes [185].

Another interesting extension would be the creation of a caching libservice and its integration on the filesystem service of a pool. The caching libservice may either use system memory to back the filesystem service cache, or a local storage device addressable at block or byte granularity.

13.3 Relaxed Queues

The RCQ family of queues that we have designed and implemented consists of queue algorithms that utilize a limited amount of memory space and achieve substantially lower operation and wait latency than existing algorithms. The relaxed operation ordering combined with the lock-free synchronization improve the operation concurrency and allow the items to depart faster from the queue.

We anticipate the proposed approach to benefit a wide range of queue applications in modern systems. In future work we plan to explore possible applications of the RCQ algorithms across a range of runtimes and services in parallel systems. In particular, work stealing is commonly used for the dynamic scheduling of task-parallel computations. A worker thread either pulls tasks from its own queue or steals tasks from the queues of other workers. We leave for future work the use of RCQ variants to support relaxed forms of queue ordering (e.g., FIFO or LIFO) in work stealing. Queues implemented in shared memory are increasingly used for the communication of the applications with systems services running at the kernel or user level. Given the encouraging results of our present work, we are interested to apply the RCQ algorithms in the parallel access of low-latency devices for data networking, storage, and processing acceleration (e.g., GPUs). Moreover, we anticipate further understanding of the individual RCQ variants through formal analysis of their properties in common and special cases.

13.4 Cloud Security

Secure access control is a challenging problem that organizations face in collaborative cloud environments. In our research we examined approaches for efficient and effective support of multitenancy in distributed filesystems used in the cloud.

As the cloud ecosystem is increasingly populated with lightweight virtual machines, such as containers, we anticipate that the infrastructure will need to more actively participate in the enforcement of secure access control [54, 298]. Possible benefits from this transition include: (i) reduced amount and complexity of guest software (application or system), (ii) less sensitivity to tenant software development or configuration bugs, and (iii) lower tenant administrative overhead for service deployment or management. Accordingly, the services operated by the tenant will need to focus more on application-specific functionality and be able to outsource to the provider the basic task of secure data access or sharing by one or multiple users in the same or a different tenant.

An interesting topic of future work is to consider weaker trust assumptions as the cloud use cases expand and the resulting multitenancy environment becomes more complex. For instance, the Dike access control that we proposed in this dissertation can be combined with end-to-end encryption. This will open up a number of interesting topics for future research. One such topic is how to efficiently and securely combine encryption with file sharing between the end users of the same or different tenants. Another interesting topic is how to support efficient revocation without the need for file re-encryption.

The transition from private computing environments to public cloud infrastructures has revolutionized the Internet Architecture through the offered elasticity and low cost. Nevertheless, there are several critical industries that remain skeptical in the adoption of the cloud approach for their computing needs. Specific examples include the sectors of health, finance and national defense. One of the primary reasons behind this is that in a public cloud the data and the computation may be disclosed or corrupted due to bugs, rogue administrators, or external attacks [363, 364, 365, 366].

The above issue necessitates the design of a secure environment for the execution of containers by relying on hardware-based security mechanisms (e.g., Intel SGX [367], AMD Secure Technology [181], ARM TrustZone [180]). Such an approach will enable a secure solution for deploying workloads with strong confidential computing guarantees in confidential containers that run on hardware-based trusted execution environments. As a result, tenants will be able to deploy confidential workloads onto public clouds without trusting their cloud provider.

Our plans for future work also include the study of filesystem multitenancy over multiple (possibly federated) clouds for improved scalability, flexibility, and security.

13.5 Summary

In this chapter, we described how our work can be extended in the future. First, we described how the dynamic resource allocation of host resources to tenant containers will benefit both the tenants and the provider of a cloud platform, by enabling resource elasticity. We discussed a decentralized method that enables the dynamic resource allocation using per-pool user-level shared-state scheduling. Second, we discussed the possibility of moving critical I/O services from the kernel to user-level by taking advantage of the libservices abstraction. Third, we discussed possible extensions of our relaxed queue algorithms and their integration to a range of applications and system services. Finally, we discussed interesting topics for future research that will secure both the data and the execution of containers, in order to enable tenants with strong confidentiality needs to adopt a public cloud platform.

Chapter 14

Lessons Learned and Conclussions

In this dissertation, we have presented systems and methods to advance the multitenant isolation of client and server services that support the data-intensive applications of different tenants on the same cloud infrastructure. In particular we have made the following contributions.

Multitenant I/O Management We demonstrated that the system kernel of a cloud machine serving software containers can become performance bottleneck and attack surface for the colocated tenants. The problem is caused by both the software components of the kernel (e.g., page cache, kernel locks) shared across the tenants in non-accountable and non-scalable ways and a number of kernel services that do not honor resource reservations (e.g., page flushing).

In this dissertation, we have shown that the relocation of the I/O services from the system kernel to the user level and the equipment of tenants with separate instances of these services can substantially improve isolation and resilience among the collocated tenants. This enables the colocated tenants to use their own reusable (hardware) and consumable (software) resources in order to isolate their I/O activity and performance.

We designed libservices as a new paradigm for structuring the client and server side of cloud applications and services of each tenant so that they can be deployed dynamically and run isolated from those of other tenants. With libservices, we built the Polytropon toolkit that consists of a POSIX-like API library linked to the cloud applications, an efficient interprocess communication facility with Relaxed Concurrent Queues for request transfer and optimized memory copy for data transfer, as well as a configurable stack of frontend and backend services all running at user level. Based on commodity hardware and operating system, the Polytropon components can be used to implement several I/O services (e.g., filesystems, network stack, distributed storage) at user level and bypass the kernel during their execution. At the client-side, we used Polytropon to build the Danaus client architecture in order to separately serve the containers of each tenant, by integrating a union filesystem with a distributed filesystem client both running at user level. Danaus handles I/O with the private resources reserved for the tenant and avoids the costly locking inside the shared kernel. Our results highlighted the importance of carrying out both the execution of the I/O services, and their communication with the applications at user level. Our system achieved both performance stability and performance advantage, when compared to a hybrid filesystem that runs at user level but uses the kernel for I/O communication, or a native kernel-level filesystem, under conditions of I/O contention, across a number of applications and microbenchmarks.

Fast Interprocess Communication We examined two key components of a communication mechanism, the queue data structure that is used by producers and consumers for communication, and the memory copy operation that is used for data transfer.

We designed Relaxed Concurrent Queues as a family of relaxed queues. The Relaxed Concurrent Queues increase operation concurrency by allowing the queue operations to complete out-of-order after they are assigned sequentially to the queue slots. We demonstrated that across different workload types, system settings, and processor types the lock-free RCQ algorithms combine low operation and wait latency with high task throughput and low performance variability. The properties are retained under concurrency of different levels, and imbalance between the enqueue and dequeue threads or rates. A single array of fixed size equal to 256 slots enables stable operation with limited memory and cpu requirements. In comparison to existing queues, the lock-free RCQ algorithms achieved factors to orders of magnitude lower latency, reduced variability, and higher throughput. We anticipate the proposed approach to benefit a wide range of queue applications in modern systems.

We also focused on the memory copy operation, which is involved in many places of the software stack, and is a critical operation that can greatly impact system performance. We introduced Asterope, an optimization algorithm for finding the optimal parameters for the memory copy operation on x86 architectures. Our results demonstrated that the memory copy functions produced by Asterope track or improve the best possible performance on different systems.

Multitenant Access Control We demonstrated that the cloud infrastructure manages primary administrative entities, but it can not manage the enormous user populations that belong to different tenants in scalable ways. To this end, we designed the Dike authorization architecture that combines native access control with tenant namespace isolation and compatibility to object-based filesystems. We designed a number of secure protocols to authenticate the participating entities and authorize the data access over the network. Our prototype implementation and experimental results demonstrated the feasibility of the Dike authorization architecture with low overhead at up to thousands of tenants.

14.1 Lessons Learned

In this section, we present a list of lessons learned during the research and writing of this dissertation.

14.1.1 Interface compatibility is hard but crucial

We consider application compatibility an important consideration when designing system services. The cloud ecosystem contains a large set of well-established applications, such as web services (e.g., Apache, Nginx), databases (e.g., MySQL, MongoDB), key-value stores (e.g., RocksDB), and analytics frameworks (e.g. Spark) that need to run without modifications. New system services need to retain the traditional interface in order to benefit such applications, otherwise the applications will continue to use legacy services.

The system kernel provides a common interface to user-level applications through system calls and a convenient abstraction of the storage software stack through VFS. By moving both the execution of critical I/O services and their communication with applications to user level, we remove the kernel from the I/O path and we loose the convenience of its interface.

In order to support unmodified applications, we had to build a filesystem library that implements a huge number of system and library functions. We also strove to adhere to POSIX semantics by paying attention to POSIX implementation details. For instance, we had to carefully manage file descriptors and file reference counting in order to provide correct semantics to file and process management functions, like *open*, *fork*, and *exec*. A dynamically-linked application can preload the filesystem library to communicate with a user-level filesystem built with Polytropon.

We also found, that some functions (e.g., *exec*) involve implicit kernel I/O which could not instantiate at user-level. An example of a system call that triggers kernel-originated I/O requests is the *exec* function. During the handling of this function, the kernel needs to perform I/O operations in order to load the application binary into the memory space of the process. To handle such cases, we introduced a dual interface that allows an application to access user-level I/O services either directly at user level, or through the kernel VFS using FUSE. The dual interface can be also used by unmodified statically-linked applications.

14.1.2 User-level filesystems are not optimized yet

Most of the existing user-level storage libraries are less mature than their kernel counterparts. For instance, while experimenting with the Danaus client, we found a global lock in the the library-based client of Ceph that is used to protect the concurrent execution of file operations. However, this lock does not exist in the kernel-based Ceph client, which adopts a more fine-grained locking scheme.

We believe that the wider adoption of user-level storage drivers will encourage their further development and optimization. Their improvement will facilitate the dynamic readjustment of the allocated resources (e.g., cache memory) and the accurate online monitoring of the utilization effectiveness.

14.1.3 The FIFO queueing semantics can be relaxed

A first-in-first-out (FIFO) queue is a typical choice in producer-consumer software stacks, as the queue avoids starvation by giving priority to the item with the longest wait time. While the FIFO ordering is fair, it is an inherently sequential policy that limits queue concurrency and delays operations. Indeed, a FIFO queue allows only a single item at a time to be inserted or removed at each end of the queue, while the correct implementation of the queue operations requires tracking the queue empty or full condition with state updates (e.g., item counting) that reduce concurrency. Recent data structures aim to increase concurrency by relaxing ordering and accommodate multiple FIFO queues to distribute the operations that insert or remove items.

The latency and throughput are critical performance metrics in current manycore systems with devices of high throughput and low latency. Assuming that the FIFO queue is typical choice in producer-consumer software stacks, we examined the implications of strict FIFO or relaxed ordering to several operation, wait and task metrics. We found that the existing strict or relaxed algorithms often generate much higher tail latency due to the synchronization costs and operation retries, or the relaxation semantics.

While the FIFO ordering is fair to the item with the longest wait time in the queue, the actual order of request service in a system depends on several factors beyond the queue priority, like the thread scheduling policy of the system and various delays that are generated by the system software and the underlying devices. To this end, the unpredictable system behavior in the service path weakens the relevance of the strict queue ordering.

For increased performance in the producer-consumer communication we designed a family of relaxed queues that split queue operations into two stages. The first stage assigns the operation sequentially into a queue slot, and the second stage completes the operation potentially out-of-order. Our queues achieved substantially improved performance and stability when compared with strict FIFO or other relaxed queues from the literature.

14.1.4 Simple designs are better

A key principle that we followed during the design process of the systems that we presented in this dissertation is to keep their design as simple as possible, since simple designs lead to better systems.

For instance, the Relaxed Concurrent Queue family consists of simple but novel queue algorithms thar relax queue operations. The algorithms avoid the costly dynamic memory management for adjusting the size of the queue according to the changing load levels. Instead, they allow multiple enqueuers and dequeuers to obtain the same slot number and attempt completing their operation on that slot. An operation consists of two simple steps, a first step that uses fetch-and-add to obtain a queue slot, and a second step that uses a CAS operation to complete the operation on that slot. Our algorithms are simple, but powerful, as they substantially outperform more advanced algorithms that implement FIFO or relaxed queues, and achieve stable performance.

14.1.5 Fewer copies may not always be the best approach

We considered various possible approaches for the exchange of data between the applications and the user-level filesystem services. We used the cross memory attach facility of the Linux kernel that copies data between two address spaces without crossing kernel memory, in order to transfer data between the application and the filesystem service with a single copy. We found that this method achieves better performance, when compared with a native copy method that is invoked twice to copy data across different address spaces using an intermediate shared memory buffer.

However, by taking into account the architectural details of the processor and optimizing the memory copy function, we experimentally demonstrated that an approach that involves two copies using the optimized method can outperform the single-copy approach that relies on a common memory copy function.

The above observation highlighted the need to automatically optimize critical functions, by finding the optimal parameters according to the underlying architecture of the target system. This is the key idea behind the Asterope algorithm that we introduced in this dissertation for the optimization of the memory copy routine.

14.1.6 Access Control should be outsourced to the infrastructure

In current cloud infrastructures, a shared filesystem typically stores disk images or file shares with access restricted to a single tenant without native sharing support. For instance, network-level isolation is commonly applied in the form of a Virtual

Private Cloud based on encrypted communication channels or VLAN settings [306]. The file access control is fully enforced by the tenant, and security violations from vulnerable configurations compromise the resources of the tenant.

The provider neither enforces the access control of individual users, nor directly supports the storage sharing among the users of different tenants. The infrastructure primarily manages administrative entities, but leaves the virtual machines of the tenant responsible to control the access rights of individual users over local or networked resources. Consequently, the tenant administrators are burdened with the task of deploying traditional directory services to both specify and enforce user access rights, while storage sharing remains inflexible and potentially error-prone.

By moving the enforcement of access control from the virtual machine to the shared filesystem, we essentially strengthen the protection of confidentiality and integrity in the stored data over a shared consolidated environment. An individual access is permitted if it identifies securely the identities of the requesting tenant and end user according to the policy specification that has been previously configured securely into the underlying filesystem.

14.2 Concluding Remarks

When we configure a multitenant environment we often face the dilemma of whether we should favor performance over utilization. It turns out that the aggressive resource utilization leads to high performance variability especially with I/O-intensive applications.

The problem of resource contention, especially in the public clouds, is somewhat unpredictable and often undermines the performance and resilience of the cloud-based services. We found that to a large extent this issue is caused by the computing resources of the cloud machines shared across competing tenants in non scalable ways. With the prevalence of operating-system-level virtualization through containers, the system kernel of a machine is contended among the colocated tenants. As a result, the hardware resources along with the software modules used by the shared kernel can become hot spots and bottlenecks in the application execution.

With our proposed architecture we make the container more independent with respect to the shared operating system and the colocated tenants. The deployment of container software is streamlined by running the I/O services at user level and instantiating them dynamically and separately for each tenant. With the design of efficient and truly concurrent queues, and the automated platform-specific optimization of critical functions, like memory copy, we build an efficient communication mechanism between applications and user-level services. The performance and security isolation is advanced because the application and container services of each tenant run on private resources and software components.

Additionally, our research opens up the road for the cloud infrastructure to participate more actively in the enforcement of secure access control. A multitenant access control mechanism which is implemented natively at the infrastructure is more scalable and relieves the tenant from high administrative overheads for user management.

The systems that we introduce in this dissertation can be used to build a range of services (e.g., key-value stores, local/distributed filesystems, analytics) that run entirely at user level. The proposed I/O isolation is the first step toward flexible resource management that guarantees the tenant reservations and provides additional resources dynamically as they become available per machine. The benefits include increased performance at high utilization, predictable latency and throughput, better scalability over multicore machines and low-latency devices, higher energy efficiency by reducing the number of the powered machines in the cloud infrastructures, improved reliability by dedicating distinct services to the different tenants instead of having them share the same modules of the kernel. The benefits transfer to the data-intensive applications of the tenants (e.g., web services, big-data analytics) and lower the cost of cloud computing.

Bibliography

[1] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A Distributed Storage System for Structured Data," *ACM Trans. Comput. Syst.*, vol. 26, no. 2, pp. 4:1–4:26, Jun. 2008.

[2] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. L. A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford, "Spanner: Google&Rsquo;s Globally Distributed Database," *ACM Transactions on Computer Systems (TOCS) TOCS Homepage archive*, vol. 31, no. 3, pp. 8:1–8:22, Aug. 2013.

[3] I. Zhang, N. Lebeck, P. Fonseca, B. Holt, R. Cheng, A. Norberg, A. Krishnamurthy, and H. M. Levy, "Diamond: Automating Data Management and Storage for Wide-Area, Reactive Applications," in *USENIX OSDI*, Savannah, GA, USA, 2016, pp. 265–283.

[4] Y. Mansouri, A. N. Toosi, and R. Buyya, "Data Storage Management in Cloud Environments: Taxonomy, Survey, and Future Directions," *ACM Computing Surveys*, vol. 50, no. 6, pp. 91:1–91:51, Dec. 2017.

[5] X. Zhang, J. Ai, Z. Wang, J. Lu, and X. Meng, "An Efficient Multi-dimensional Index for Cloud Data Management," in *Proceedings of the First International Workshop on Cloud Data Management*, Hong Kong, China, 2009, pp. 17–24.

[6] G. Margaritis and S. V. Anastasiadis, "Incremental Text Indexing for Fast Disk-Based Search," *ACM Transactions on the Web*, vol. 8, no. 3, pp. 16:1–16:31, Jul. 2014.

[7] A. P. Iyer and I. Stoica, "A scalable distributed spatial index for the internet-of-things," in *Proceedings of the 2017 Symposium on Cloud Computing*, 2017, pp. 548–560.

[8] F. Baig, H. Vo, T. Kurc, J. Saltz, and F. Wang, "SparkGIS: Resource Aware Efficient In-Memory Spatial Query Processing," in *Proceedings of the 25th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, 2017, pp. 28:1–28:10.

[9] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, ser. OSDI'04, San Francisco, CA, USA, 2004.

[10] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized streams: Fault-tolerant streaming computation at scale," in *ACM SOSP*, Farmington, PA, USA, 2013, pp. 423–438.

[11] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "Graphx: Graph processing in a distributed dataflow framework," in *USENIX OSDI*, Broomfield, CO, USA, 2014, pp. 599–613.

[12] Abadi, Martín and Barham, Paul and Chen, Jianmin and Chen, Zhifeng and Davis, Andy and Dean, Jeffrey and Devin, Matthieu and Ghemawat, Sanjay and Irving, Geoffrey and Isard, Michael and Kudlur, Manjunath and Levenberg, Josh and Monga, Rajat and Moore, Sherry and Murray, Derek G. and Steiner, Benoit and Tucker, Paul and Vasudevan, Vijay and Warden, Pete and Wicke, Martin and Yu, Yuan and Zheng, Xiaoqiang, "TensorFlow: a system for large-scale machine learning," in *USENIX OSDI*, Savannah, GA, USA, 2016, pp. 265–283.

[13] M. Creeger, "Cloud computing: An overview," *ACM Queue - Distributed Computing*, no. 5, Jun. 2009.

[14] K. D. Bowers, A. Juels, and A. Oprea, "HAIL: a high-availability and integrity layer for cloud storage," in *ACM Conf. on Computer and Communications Security*, Chicago, IL, USA, Nov. 2009, pp. 187–198.

[15] N. Santos, K. P. Gummadi, and R. Rodrigues, “Towards trusted cloud computing,” in *USENIX Workshop on Hot Topics in Cloud Computing*, San Diego, CA, USA, Jun. 2009.

[16] R. Geambasu, S. D. Gribble, and H. M. Levy, “CloudViews: communal data sharing in public clouds,” in *USENIX HotCloud*, San Diego, CA, USA, Jun. 2009.

[17] M. L. Badger, T. Grance, R. Patt-Corner, and J. M. Voas, “Cloud computing synopsis and recommendations,” National Institute of Standards and Technology, Tech. Rep. NIST SP - 800-146, May 2012.

[18] D. Song, E. Shi, I. Fischer, and U. Shankar, “Cloud data protection for the masses,” *Computer*, vol. 45, no. 1, pp. 39–45, Jan. 2012.

[19] A. Juels and A. Oprea, “New Approaches to Security and Availability for Cloud Data,” *Communications of the ACM*, vol. 56, no. 2, pp. 64–73, Feb. 2013.

[20] O. A. Ben-Yehuda, M. Ben-Yehuda, A. Schuster, and D. Tsafrir, “The rise of raas: The resource-as-a-service cloud,” *Communications of the ACM*, vol. 57, no. 7, pp. 76–84, Jul. 2014.

[21] E. Pattuk, M. Kantarcioglu, Z. Lin, and H. Ulusoy, “Preventing cryptographic key leakage in cloud virtual machines,” in *USENIX Security Symp.*, San Diego, CA, USA, Aug. 2014, pp. 703–718.

[22] E. Cortez, A. Bonde, A. Muzio, M. Russinovich, M. Fontoura, and R. Bianchini, “Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms,” in *ACM Symp. Operating Systems Principles*, Shanghai, China, Oct. 2017.

[23] S. Ghemawat, H. Gobioff, and S.-T. Leung, “The google file system,” in *ACM SOSP*, ser. SOSP ’03, Bolton Landing, NY, USA, 2003, pp. 29–43.

[24] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn, “Ceph: A Scalable, High-Performance Distributed File System,” in *USENIX Symposium on Operating Systems Design and Implementation*. Berkeley, CA: USENIX Association, Nov. 2006, pp. 307–320.

[25] J. G. Hansen and E. Jul, “Lithium: Virtual machine storage for the cloud,” in *ACM SoCC*, Indianapolis, IN, USA, Jun. 2008, pp. 15–26.

[26] M. Shamma, D. T. Meyer, J. Wires, M. Ivanova, N. C. Hutchinson, and A. Warfield, “Capo: Recapitulating storage for virtual desktops,” in *USenix FAST*, San Jose, CA, USA, Feb. 2011, pp. 31–45.

[27] R. Koller, L. Marmol, R. Rangaswami, S. Sundararaman, N. Talagala, and M. Zhao, “Write policies for host-side flash caches,” in *USENIX Conf. File & Storage Tech.*, San Jose, CA, USA, Feb. 2013, pp. 45–58.

[28] D. Quigley, J. Sipek, C. P. Wright, and E. Zadok, “Unionfs: User- and community-oriented development of a unification file system,” in *Ottawa Linux Symposium*, Ottawa, Canada, 2006, pp. 357–370.

[29] C.-H. Ng, M. Ma, T.-Y. Wong, P. P. C. Lee, and J. C. S. Lui, “Live Deduplication Storage of Virtual Machine Images in an Open-source Cloud,” in *Proceedings of the 12th International Middleware Conference*, Lisbon, Portugal, 2011, pp. 80–99.

[30] F. Armknecht, J.-M. Bohli, G. O. Karame, and F. Youssef, “Transparent Data Deduplication in the Cloud,” in *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, Denver, CO, USA, 2015, pp. 886–900.

[31] W. Vogels, “Beyond server consolidation,” *ACM Queue*, vol. 6, no. 1, Mar. 2008.

[32] U. Gurav and R. Shaikh, “Virtualization: A key feature of cloud computing,” in *ACM ICWET*, 2010, pp. 227–229.

[33] H. Lv, Y. Dong, J. Duan, and K. Tian, “Virtualization Challenges: A View from Server Consolidation Perspective,” in *SIGPLAN/SIGOPS VEE*, 2012, pp. 15–26.

[34] A. Klimovic, Y. Wang, P. Stuedi, A. Trivedi, J. Pfefferle, and C. Kozyrakis, “Pocket: Elastic ephemeral storage for serverless analytics,” in *USENIX Symposium on Operating Systems Design and Implementation*. Berkeley, CA: USENIX Association, Oct. 2018, pp. 427–444.

[35] Q. Liu and Z. Yu, “The Elasticity and Plasticity in Semi-Containerized Co-Locating Cloud Workload: A View from Alibaba Trace,” in *ACM Symposium on Cloud Computing*. New York, NY: ACM, Oct. 2018, pp. 347–360.

[36] K. Rzadca, P. Findeisen, J. Swiderski, P. Zych, P. Broniek, J. Kusmierek, P. Nowak, B. Strack, P. Witusowski, S. Hand, and J. Wilkes, "Autopilot: Workload Autoscaling at Google," in *ACM European Conference on Computer Systems*. New York, NY: ACM, Apr. 2020.

[37] "Manage data in docker," 2020, https://docs.docker.com/storage/.

[38] "Kubernetes concepts," 2020, https://kubernetes.io/docs/concepts/.

[39] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, Omega, and Kubernetes," *Commununications of the ACM*, vol. 59, no. 5, pp. 50–57, Apr. 2016.

[40] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. M. Carreira, K. Krauth, N. Yadwadkar, J. E. Gonzalez, R. A. Popa, I. Stoica, and D. A. Patterson, "Cloud Programming Simplified: A Berkeley View on Serverless Computing," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2019-3, Feb. 2019.

[41] "Amazon Web Services," 2020, https://aws.amazon.com/.

[42] E. Thereska, H. Ballani, G. O'Shea, T. Karagiannis, A. Rowstron, T. Talpey, R. Black, and T. Zhu, "IOFlow: A Software-Defined Storage Architecture," in *ACM Symposium on Operating Systems Principles*. New York, NY: ACM, 2013, pp. 182–196.

[43] S. Angel, H. Ballani, T. Karagiannis, G. O'Shea, and E. Thereska, "End-to-end Performance Isolation Through Virtual Datacenters," in *USENIX Symposium on Operating Systems Design and Implementation*. Berkeley, CA: USENIX Association, Oct. 2014, pp. 233–248.

[44] M. Dalton, D. Schultz, J. Adriaens, A. Arefin, A. Gupta, B. Fahs, D. Rubinstein, E. C. Zermeno, E. Rubow, J. A. Docauer, J. Alpert, J. Ai, J. Olson, K. DeCabooter, M. de Kruijf, N. Hua, N. Lewis, N. Kasinadhuni, R. Crepaldi, S. Krishnan, S. Venkata, Y. Richter, U. Naik, and A. Vahdat, "Andromeda: Performance, Isolation, and Velocity at Scale in Cloud Network Virtualization," in *USENIX Symposium on Networked Systems Design and Implementation*. Berkeley, CA: USENIX Association, Apr. 2018, pp. 373–387.

[45] P. Sharma, L. Chaufournier, P. Shenoy, and Y. C. Tay, “Containers and Virtual Machines at Scale: A Comparative Study,” in *ACM/IFIP/USENIX Intl Middleware Conference*. New York, NY: ACM, Dec. 2016, pp. 1:1–1:13.

[46] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, “An updated performance comparison of virtual machines and linux containers,” Jul. 2014, iBM Research Report RC25482, IBM Research Division.

[47] C. Min, S. Kashyap, S. Maass, and T. Kim, “Understanding Manycore Scalability of File Systems,” in *USENIX Annual Technical Conference*. Berkeley, CA: USENIX Association, Jun. 2016, pp. 71–85.

[48] B. K. R. Vangoor, V. Tarasov, and E. Zadok, “To FUSE or Not to FUSE: Performance of User-Space File Systems,” in *USENIX Conference on File and Storage Technologies*. Santa Clara, CA: USENIX Association, Feb. 2017, pp. 59–72.

[49] A. Leung, A. Spyker, and T. Bozarth, “Titus: Introducing Containers to the Netflix Cloud,” *Comm. ACM*, vol. 61, no. 2, pp. 38–45, Feb. 2018.

[50] A. Verma, L. Pedrosa, M. R. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, “Large-scale cluster management at google with borg,” in *ACM European Conference on Computer Systems*. New York, NY: ACM, Apr. 2015, pp. 18:1–18:17.

[51] R. Nakazawa, K. Ogata, S. Seelam, and T. Onodera, “Taming Performance Degradation of Containers in the Case of Extreme Memory Overcommitment,” in *IEEE International Conference on Cloud Computing*. Piscataway, NJ: IEEE, Jun. 2017, pp. 196–204.

[52] M. Lamourine, “Storage Options for Software Containers,” *USENIX ;login:*, no. 1, pp. 10–14, Feb. 2015.

[53] V. Tarasov, L. Rupprecht, D. Skourtis, A. Warke, D. Hildebrand, M. Mohamed, N. Mandagere, W. Li, R. Rangaswami, and M. Zhao, “In search of the ideal storage configuration for docker containers,” in *IEEE International Workshops on Foundations and Applications of Self* Systems (FAS*W)*. Tucson, AZ: IEEE, Sep. 2017, pp. 199–206.

[54] E. Reshetova, J. Karhunen, T. Nyman, and N. Asokan, “Security of os-level virtualization technologies,” in *Nordic Conference on Secure IT Systems*. Berlin, Germany: Springer, Oct. 2014, pp. 77–93, lNCS 8788.

[55] B. Tak, C. Isci, S. Duri, N. Bila, S. Nadgowda, and J. Doran, “Understanding security implications of using containers in the cloud,” in *USENIX Annual Technical Conference*. Berkeley, CA: USENIX Association, Jul. 2017, pp. 313–319.

[56] C. Delimitrou and C. Kozyrakis, “Bolt: I know what you did last summer... in the cloud,” in *ACM Intl Conf on Architectural Support for Programming Languages and Operating Systems*. New York, NY: ACM, Apr. 2017, pp. 599–613.

[57] B. Burns and D. Oppenheimer, “Design patterns for container-based distributed systems,” in *USENIX Workshop on Hot Topics in Cloud Computing*, Denver, CO, Jun. 2016.

[58] D. Taibi, V. Lenarduzzi, and C. Pahl, “Processes, motivations, and issues for migrating to microservices architectures: An empirical investigation,” *IEEE Cloud Computing*, vol. 4, no. 5, pp. 22–32, 2017.

[59] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming*. Waltham, MA, USA: Morgan Kaufmann, 2012.

[60] H. J. Chu and Y. Liu, “User Space TCP - Getting LKL Ready for the Prime Time,” in *Netdev 1.2 conference*, 2016.

[61] P. Druschel and L. L. Peterson, “Fbufs: A high-bandwidth cross-domain transfer facility,” in *ACM Symposium on Operating Systems Principles*, Asheville, NC, Dec. 1993, pp. 189–202.

[62] L. McVoy and C. Staelin, “lmbench: Portable tools for performance analysis,” in *USENIX Annual Technical Conference*, San Diego, CA, Jan. 1996, pp. 279–294.

[63] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe, “Arrakis: The Operating System is the Control Plane,” in *USENIX Symposium on Operating Systems Design and Implementation*. Berkeley, CA: USENIX Association, Oct. 2014, pp. 1–16.

[64] P.-L. Aublin, S. B. Mokhtar, G. Muller, and V. Quéma, "Zimp : Efficient inter-core communications on manycore machines," INRIA Rhône-Alpes, Tech. Rep., Apr. 2011.

[65] P. Enberg, A. Rao, and S. Tarkoma, "I/o is faster than the cpu: Let's partition resources and eliminate (most) os abstractions," in *Workshop on Hot Topics in Operating Systems*, May 2019, pp. 81–87.

[66] L. Soares and M. Stumm, "FlexSC: Flexible system call scheduling with exception-less system calls," in *USENIX Conference on Operating Systems Design and Implementation*. Vancouver, BC, Canada: USENIX Association, Oct. 2010, pp. 33–46.

[67] J. Axboe, "Efficient io with io_uring," 2019, https://kernel.dk/io_uring.pdf.

[68] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania, "The multikernel: A new os architecture for scalable multicore systems," in *ACM Symposium on Operating Systems Principles*, Big Sky, Montana, USA, Oct. 2009, pp. 29–44.

[69] Z. Mi, D. Li, Z. Yang, X. Wang, and H. Chen, "Skybridge: Fast and secure inter-process communication for microkernels," in *ACM European Conference on Computer Systems*. New York, NY: ACM, Mar. 2019.

[70] M. Hedayati, S. Gravani, E. Johnson, J. Criswell, M. L. Scott, K. Shen, and M. Marty, "Hodor: Intra-Process Isolation for High-Throughput Data Plane Libraries," in *USENIX Annual Technical Conference*. Renton, WA: USENIX Association, Jul. 2019, pp. 489–504.

[71] V. Narayanan, A. Balasubramanian, C. Jacobsen, S. Spall, S. Bauer, M. Quigley, A. Hussain, A. Younis, J. Shen, M. Bhattacharyya, and A. Burtsev, "Lxds: Towards isolation of kernel subsystems," in *USENIX Annual Technical Conference*. Berkeley, CA: USENIX Association, Jul. 2019, pp. 269–284.

[72] E. Y. Jeong, S. Wood, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park, "mtcp: a highly scalable user-level TCP stack for multicore systems," in *USENIX Symposium on Networked Systems Design and Implementation*, Seattle, WA, Apr. 2014, pp. 489–502.

[73] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion, "IX: A protected dataplane operating system for high throughput and low latency," in *USENIX Symposium on Operating Systems Design and Implementation*, Broomfield, CO, Oct. 2014, pp. 49–65.

[74] K. Yasukata, M. Honda, D. Santry, and L. Eggert, "Stackmap: Low-latency networking with the OS stack and dedicated nics," in *USENIX Annual Technical Conference*, Denver, CO, Jun. 2016, pp. 43–56.

[75] Z. Wang, "Performance optimization of memcpy in DPDK," 2017, https://software.intel.com/content/www/us/en/develop/articles/performance-optimization-of-memcpy-in-dpdk.html.

[76] I. Zhang, J. Liu, A. Austin, M. L. Roberts, and A. Badam, "I'm not dead yet! the role of the operating system in a kernel-bypass era," in *Workshop on Hot Topics in Operating Systems*, Bertinoro, Italy, May 2019, pp. 73–80.

[77] R. Kadekodi, S. K. Lee, S. Kashyap, T. Kim, A. Kolli, and V. Chidambaram, "Splitfs: Reducing software overhead in file systems for persistent memory," in *ACM Symposium on Operating Systems Principles*. New York, NY: ACM, Oct. 2019, pp. 494–508.

[78] S. Rus, R. Ashok, and D. X. Li, "Automated locality optimization based on the reuse distance of string operations," in *International Symposium on Code Generation and Optimization*, Chamonix, France, Apr. 2011, pp. 181–190.

[79] J. D. McCalpin, "Memory bandwidth and machine balance in current high performance computers," *IEEE Computer Society Technical Committee on Computer Architecture Newsletter*, pp. 19–25, Dec. 1995.

[80] J. Lee, H. Kim, and R. Vuduc, "When prefetching works, when it doesn't, and why," *ACM Trans. Archit. Code Optim.*, vol. 9, no. 1, Mar. 2012.

[81] S. Ainsworth, "Prefetching for complex memory access patterns," Ph.D. dissertation, Computer Laboratory, University of Cambridge, Jul. 2018.

[82] R. Guo, X. Liao, H. Jin, J. Yue, and G. Tan, "Nightwatch: Integrating lightweight and transparent cache pollution control into dynamic memory allocation sys-

tems," in *USENIX Annual Technical Conference*, Santa Clara, CA, Jul. 2015, pp. 307–318.

[83] H. Wang, M. Luo, K. Kandalla, S. Sur, and D. K. Panda, "Can streaming SIMD non-temporal instructions benefit intra-node MPI communication on modern multi-core platforms?" The Ohio State University, Columbus, OH, Tech. Rep. 29, 2010.

[84] D. Buntinas, G. Mercier, and W. Gropp, "Design and evaluation of nemesis, a scalable, low-latency, message-passing communication subsystem," in *IEEE International Symposium on Cluster Computing and the Grid*, Singapore, Singapore, May 2006, pp. 521–530.

[85] K. Vaidyanathan, L. Chai, W. Huang, and D. K. Panda, "Efficient asynchronous memory copy operations on multi-core systems and i/oat," in *IEEE International Conference on Cluster Computing*, Austin, TX, Sep. 2007, pp. 159–168.

[86] "The GNU C Library (glibc)," 2020, https://www.gnu.org/software/libc/.

[87] A. Singh, J. Ong, A. Agarwal, G. Anderson, A. Armistead, R. Bannon, S. Boving, G. Desai, B. Felderman, P. Germano, A. Kanagala, J. Provost, J. Simmons, E. Tanda, J. Wanderer, U. Hölzle, S. Stuart, and A. Vahdat, "Jupiter rising: A decade of clos topologies and centralized control in Google's datacenter network," in *ACM SIGCOMM Conf.*, London, UK, Aug. 2015, pp. 183–197.

[88] B. Pfaff, T. Garfinkel, and M. Rosenblum, "Virtualization Aware File Systems: Getting Beyond the Limitations of Virtual Disks," in *USENIX Symp. on Networked Systems Design and Implementation*, San Jose, CA, 2006, pp. 353–366.

[89] V. Jujjuri, E. V. Hensbergen, and A. Liguori, "VirtFS: Virtualization aware File System pass-through," in *Ottawa Linux Symp.*, 2010.

[90] J. Darcy, "Building a Cloud File System," *USENIX; login:*, vol. 36, no. 3, pp. 14–21, Jun. 2011.

[91] E. Wobber, M. Abadi, M. Burrows, and B. Lampson, "Authentication in the Taos operating system," *ACM Trans. Comput. Syst.*, vol. 12, no. 1, pp. 3–32, Feb. 1994.

[92] S. Miltchev, V. Prevelakis, S. Ioannidis, J. Ioannidis, A. D. Keromytis, and J. M. Smith, "Secure and flexible global file sharing," in *USENIX Annual Technical Conf., Freenix Track*, San Antonio, TX, Jun. 2003, pp. 168–178.

[93] J. G. Steiner, C. Neuman, and J. I. Schiller, "Kerberos: An authentication service for open network systems," in *USENIX Winter Conf.*, Dallas, TX, USA, Jan. 1988, pp. 191–202.

[94] https://aws.amazon.com/ec2.

[95] https://aws.amazon.com/s3/.

[96] https://azure.microsoft.com/en-us/services/cache/.

[97] https://azure.microsoft.com/en-us/services/cosmos-db/.

[98] https://aws.amazon.com/cloud9.

[99] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, "Efficient Software-Based Fault Isolation," in *ACM SOSP*, Asheville, NC, USA, 1993, pp. 203–216.

[100] D. Price, A. Tucker, and S. Microsystems, "Solaris zones: Operating system support for consolidating commercial workloads," in *Usenix LISA*, 2004, pp. 241–254.

[101] A. Kurmus, M. Gupta, R. Pletka, C. Cachin, and R. Haas, "A Comparison of Secure Multi-tenancy Architectures for Filesystem Storage Clouds," in *ACM/IFIP/USENIX Intl Middleware Conf.*, Lisboa, Portugal, Dec. 2011, pp. 460–479.

[102] L. Lu, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Fault isolation and quick recovery in isolation file systems," in *USENIX HotStorage Workshop*, San Jose, CA, USA, Jun. 2013.

[103] M. Factor, D. Hadas, A. Hamama, N. Har'el, E. K. Kolodner, A. Kurmus, A. Shulman-Peleg, and A. Sornioti, "Secure logical isolation for multi-tenancy in cloud storage," in *IEEE Intl. Conf. Massive Storage Systems and Technology*, Long Beach, CA, USA, May 2013.

[104] J. Kang, B. Zhang, T. Wo, C. Hu, and J. Huai, "MultiLanes: Providing Virtualized Storage for OS-level Virtualization on Many Cores," in *USENIX Conference on File and Storage Technologies*. Santa Clara, CA: USENIX Association, 2014, pp. 317–329.

[105] C.-C. Tsai, K. S. Arora, N. Bandi, B. Jain, W. Jannen, J. John, H. A. Kalodner, V. Kulkarni, D. Oliveira, and D. E. Porter, "Cooperation and security isolation of library oses for multi-process applications," in *ACM European Conference on Computer Systems*. New York, NY: ACM, Apr. 2014, pp. 9:1–9:14.

[106] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. L. Stillwell, D. Goltzsche, D. Eyers, R. Kapitza, P. Pietzuch, and C. Fetzer, "SCONE: Secure Linux Containers with Intel SGX," in *USENIX OSDI*, Savannah, GA, USA, 2016, pp. 689–703.

[107] Z. Zhuang, "Don't let Linux control groups run uncontrolled," Aug. 2016, https://engineering.linkedin.com/blog/2016/08/don_t-let-linux-control-groups-uncontrolled.

[108] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis, "Heracles: Improving Resource Efficiency at Scale," in *Intl Symposium on Computer Architecture*, Portland, OR, USA, Jun. 2015, pp. 450–462.

[109] X. Wu, W. Wang, and S. Jiang, "TotalCOW: Unleash the Power of Copy-On-Write for Thin-provisioned Containers," in *ACM APSys*, Tokyo, Japan, 2015, pp. 15:1–15:7.

[110] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, Omega, and Kubernetes," *Comm. ACM*, vol. 59, no. 5, pp. 50–57, 2016.

[111] F. Manco, C. Lupu, F. Schmidt, J. Mendes, S. Kuenzer, S. Sati, K. Yasukata, C. Raiciu, and F. Huici, "My VM is Lighter (and Safer) than your Container," in *ACM Symposium on Operating Systems Principles*, Shanghai, China, Oct. 2017, pp. 218–233.

[112] A. J. Feldman, W. P. Zeller, M. J. Freedman, and E. W. Felten, "SPORC: group collaboration using untrusted cloud resources," in *USENIX Symp. on Operating Systems Design and Implementation*, Vancouver, Canada, Oct. 2010, pp. 337–350.

[113] D. Wentzlaff, C. Gruenwald, N. Beckmann, K. Modzelewski, A. Belay, L. Youseff, J. Miller, and A. Agarwal, “An operating system for multicore and clouds: mechanisms and implementation,” in *Proceedings of the 1st ACM symposium on Cloud computing*, Indianapolis, IN, USA, 2010, pp. 3–14.

[114] C.-C. Tsai, B. Jain, N. A. Abdul, and D. E. Porter, “A study of modern Linux API usage and compatibility: what to support when you’re supporting,” in *ACM EuroSys*, London, UK, 2016, pp. 16:1–16:16.

[115] T. Dillon, C. Wu, and E. Chang, “Cloud computing: Issues and challenges,” in *IEEE Intl Conf. on Advanced Information Networking and Applications*, Perth, Australia, Apr. 2010.

[116] M. J. Bach, *The Design of the UNIX Operating System*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1986.

[117] G. J. Popek and R. P. Goldberg, “Formal requirements for virtualizable third generation architectures,” *Communications of the ACM*, vol. 17, no. 7, pp. 412–421, Jul. 1974.

[118] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, “Xen and the Art of Virtualization,” in *ACM SOSP*, Bolton Landing, NY, USA, Oct. 2003, pp. 164–177.

[119] E. Bugnion, S. Devine, M. Rosenblum, J. Sugerman, and E. Y. Wang, “Bringing Virtualization to the x86 Architecture with the Original VMware Workstation,” *ACM Transations on Computer Systems*, vol. 30, no. 4, pp. 12:1–12:51, Nov. 2012.

[120] E. Bugnion, S. Devine, and M. Rosenblum, “Disco: Running Commodity Operating Systems on Scalable Multiprocessors,” in *ACM SOSP*, Saint-Malo, France, Oct. 1997.

[121] C. A. Waldspurger, “Memory Resource Management in VMware ESX Server,” in *Usenix OSDI*, Boston, MA, USA, Dec. 2002.

[122] M. Rosenblum and T. Garfinkel, “Virtual Machine Monitors: Current Technology and Future Trends,” *IEEE Computer*, vol. 38, no. 5, pp. 39–47, May 2005.

[123] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, “Geiger: monitoring the buffer cache in a virtualized machine environment,” in *ACM ASPLOS*, Oct. 2006, pp. 14–24.

[124] W. Zhao and Z. Wang, “Dynamic memory balancing for virtual machines,” in *ACM SIGPLAN/SIGOPS VEE*, Mar. 2009, pp. 21–30.

[125] J. Sugerman, G. Venkitachalam, and B.-H. Lim, “Virtualizing I/O Devices on VMware Workstation’s Hosted Virtual Machine Monitor,” in *Usenix ATC*, Boston, MA, USA, Jun. 2001.

[126] J. Liu, W. Huang, B. Abali, and D. K. Panda, “High performance VMM-bypass I/O in virtual machines,” in *Usenix ATEC*, Boston, MA, USA, Jun. 2006.

[127] Y. Dong, J. Dai, Z. Huang, H. Guan, K. Tian, and Y. Jiang, “Towards high-quality I/O virtualization,” in *ACM SYSTOR*, Haifa, Israel, May 2009.

[128] M. Rosenblum and C. Waldspurger, “I/O Virtualization: Decoupling a logical device from its physical implementation offers many compelling advantages,” *ACM Queue*, vol. 9, no. 11, Nov. 2011.

[129] A. Gordon, N. Amit, N. H. El, M. Ben-Yehuda, A. Landau, A. Schuster, and D. Tsafrir, “ELI: Bare-Metal Performance for I/O Virtualization,” in *ACM ASPLOS*, London, UK, Mar. 2012.

[130] Uhlig, R. and Neiger, G. and Rodgers, D. and Santoni, A.L. and Martins, F.C.M. and Anderson, A.V. and Bennett, S.M. and Kagi, A. and Leung, F.H. and Smith, L., “Intel Virtualization Technology,” *IEEE Computer*, vol. 38, no. 5, pp. 48–56, May 2005.

[131] D. Abramson, J. Jackson, G. Neiger, G. Regnier, R. Sankaran, I. Schoinas, R. Uhlig, B. Vembu, and J. Wiegert, “Intel Virtualization Technology for Directed I/O,” *Intel Technology Journal*, vol. 10, no. 3, 2006.

[132] M. Righini, “Enabling Intel virtualization technology features and benefits,” Intel White Paper, Tech. Rep., 2012.

[133] C. Dall and J. Nieh, “KVM/ARM: The Design and Implementation of the Linux ARM Hypervisor,” in *ACM ASPLOS*, Salt Lake City, Utah, USA, 2014, pp. 333–348.

[134] PCI SIG, "PCI-SIG Single Root I/O Virtualization," http://www.pcisig.com/specifications/iov/single_root/, accessed: 2018-03-12.

[135] G. Ammons, D. D. Silva, O. Krieger, D. Grove, B. Rosenburg, R. W. Wisniewski, M. Butrico, K. Kawachiya, and E. V. Hensbergen, "Libra: A library operating system for a jvm in a virtualized execution environment," in *Virtual Execution Environments*, 2007, pp. 13–15.

[136] D. E. Porter, S. Boyd-wickizer, J. Howell, R. Olinsky, and G. C. Hunt, "Rethinking the library os from the top down," in *ACM ASPLOS*, 2011.

[137] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft, "Unikernels: library operating systems for the cloud," in *ACM ASPLOS*, Houston, TX, USA, Mar. 2013.

[138] M. Ben-Yehuda, O. Peleg, O. A. Ben-Yehuda, I. Smolyar, and D. Tsafrir, "The Nonkernel: A Kernel Designed for the Cloud," in *ACM APSys*, 2013.

[139] A. Kantee and J. Cormack, "Rump Kernels: No OS? No Problem!" *USENIX ;login:*, vol. 39, no. 5, 2014.

[140] A. Kivity, D. Laor, G. Costa, P. Enberg, N. Har'El, D. Marti, and V. Zolotarov, "OSv-Optimizing the Operating System for Virtual Machines," in *USENIX ATC*, Philadelphia, PA, USA, Jun. 2014.

[141] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh, "Terra: A Virtual Machine-Based Platform for Trusted Computing," in *ACM SOSP*, Bolton Landing, NY, USA, Oct. 2011.

[142] R. Sailer, E. Valdez, T. Jaeger, R. Perez, L. van Doorn, J. L. Griffin, and S. Berger, "sHype: Secure Hypervisor Approach to Trusted Virtualized Systems," IBM Research Division, Tech. Rep. RC23511 (W0502-006), 2005.

[143] G. Heiser and B. Leslie, "The OKL4 microvisor: convergence point of microkernels and hypervisors," in *ACM APSys*, 2010.

[144] U. Steinberg and B. Kauer, "NOVA: a microhypervisor-based secure virtualization architecture," in *ACM EuroSys*, 2010.

[145] Z. Wang and X. Jiang, "HyperSafe: A Lightweight Approach to Provide Lifetime Hypervisor Control-Flow Integrity," in *IEEE Symposium on Security and Privacy*, 2010.

[146] F. Zhang, J. Chen, H. Chen, and B. Zang, "CloudVisor: retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization," in *ACM SOSP*, 2011.

[147] P. Colp, M. Nanavati, J. Zhu, W. Aiello, G. Coker, T. Deegan, P. Loscocco, and A. Warfield, "Breaking up is hard to do: security and functionality in a commodity hypervisor," in *ACM SOSP*, 2011.

[148] J. Szefer, E. Keller, R. B. Lee, and J. Rexford, "Eliminating the hypervisor attack surface for a more secure cloud," in *ACM CCS*, Chicago, IL, USA, Oct. 2011.

[149] Z. Wang, C. Wu, M. Grace, and X. Jiang, "Isolating commodity hosted hypervisors with HyperLock," in *ACM EuroSys*, 2012.

[150] C. tai Lee, J. min Lin, Z. wei Hong, and W. tsong Lee, "An application-oriented Linux kernel customization for embedded systems," *Journal of Information Science and Engineering*, vol. 20, 2004.

[151] A. Kurmus, R. Tartler, D. Dorneanu, B. Heinloth, V. Rothberg, A. Ruprecht, W. Schröder-Preikschat, D. Lohmann, and R. Kapitza, "Attack Surface Metrics and Automated Compile-Time OS Kernel Tailoring," in *IEEE NDSS*, 2013.

[152] A. Ruprecht, B. Heinloth, and D. Lohmann, "Automatic feature selection in large-scale system-software product lines," in *ACM GPCE*, 2014.

[153] D. R. Engler, M. F. Kaashoek, and J. O. Jr., "Exokernel: An operating system architecture for application-level resource management," in *ACM Symposium on Operating Systems Principles*. New York, NY: ACM, 1995, pp. 251–266.

[154] J. Bottomley and P. Emelyanov, "Linux Containers," vol. 39, no. 5, Oct. 2014.

[155] Soltesz, Stephen and Pötzl, Herbert and Fiuczynski, Marc E. and Bavier, Andy and Peterson, Larry, "Container-Based Operating System Virtualization: A Scalable, High-Performance Alternative to Hypervisors," in *ACM Eurosys*, 2007, pp. 275–288.

[156] O. Laadan and J. Nieh, "Operating system virtualization: Practice and experience," in *ACM SYSTOR*, Haifa, Israel, 2010, pp. 17:1–17:12.

[157] T. Combe, A. Martin, and R. D. Pietro, "To Docker or Not to Docker: A Security Perspective," *IEEE Cloud Computing*, vol. 3, no. 5, pp. 54–62, Sep. 2016.

[158] M. G. Xavier, M. V. Neves, and C. A. F. D. Rose, "A performance comparison of container-based virtualization systems for mapreduce clusters," in *Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, 2014, pp. 299–306.

[159] S. Ahn, K. La, and J. Kim, "Improving I/O Resource Sharing of Linux Cgroup for NVMe SSDs on Multi-core Systems," in *USENIX Workshop on Hot Topics in Storage and File Systems*, Denver, CO, 2016, pp. 111–115.

[160] E. W. Biederman, "Multiple Instances of the Global Linux Namespaces," in *Ottawa Linux Symposium*, Ottawa, Canada, Jul. 2006, pp. 101–112.

[161] S. E. Hallyn and A. G. Morgan, "Linux capabilities: making them work," in *Ottawa Linux Symposium*, Ottawa, Canada, 2008.

[162] P. Menage, "GROUPS," https://www.kernel.org/doc/Documentation/cgroups/cgroups.txt, accessed: 2015-04-07.

[163] T. Harter, D. Borthakur, S. Dong, A. Aiyer, L. Tang, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Analysis of HDFS under HBase: a facebook messages case study," in *USENIX FAST Conf*, Santa Clara, CA, USA, 2014, pp. 199–212.

[164] D. T. Meyer, G. Aggarwal, B. Cully, G. Lefebvre, M. J. Feeley, N. C. Hutchinson, and A. Warfield, "Parallax: Virtual disks for virtual machines," in *ACM Eurosys*, Glasgow, Scotland, Mar. 2008, pp. 41–54.

[165] S. B. Vaghani, "Virtual machine file system," *ACM SIGOPS*, vol. 44, no. 4, pp. 57–70, Dec. 2010.

[166] R. T. Fielding and R. N. Taylor, "Principled design of the modern Web architecture," *ACM TOIT*, vol. 2, no. 2, pp. 115–150, May 2002.

[167] J. Chen, C. Douglas, M. Mutsuzaki, P. Quaid, R. Ramakrishnan, S. Rao, and R. Sears, “Walnut: A unified cloud object store,” in *ACM SIGMOD*, Scottsdale, AZ, USA, May 2012, pp. 743–754.

[168] D. T. Meyer, J. Wires, N. C. Hutchinson, and A. Warfield, “Namespace Management in Virtual Desktops,” *USENIX; login:*, vol. 36, no. 1, pp. 6–11, Feb. 2011.

[169] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, “The Hadoop Distributed File System,” in *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, 2010.

[170] S. A. Weil, A. W. Leung, S. A. Brandt, and C. Maltzahn, “Rados: A scalable, reliable storage service for petabyte-scale storage clusters,” in *ACM PDSW*, 2007, pp. 35–44.

[171] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck, “Scalability in the xfs file system,” in *Usenix ATEC*, San Diego, CA, USA, 1996.

[172] “BlueStore Internals,” http://docs.ceph.com/docs/master/dev/bluestore/.

[173] S. A. Weil, S. A. Brandt, E. L. Miller, and C. Maltzahn, “CRUSH: controlled, scalable, decentralized placement of replicated data,” in *ACM/IEEE Supercomputing*, Tampa, FL, USA, Nov. 2006.

[174] D. Xiu and Z. Liu, “A formal definition for trust in distributed systems,” vol. 3650, pp. 482–489, 2005.

[175] J. M. Rushby, “Design and verification of secure systems,” in *ACM SOSP*, 1981, pp. 12–21.

[176] B. Kauer, “OSLO: Improving the Security of Trusted Computing,” in *USENIX Security*, 2007, pp. 16:1–16:9.

[177] B. Parno, J. M. McCune, and A. Perrig, “Bootstrapping Trust in Commodity Computers,” in *IEEE Symp. on Security and Privacy*, May 2010, pp. 414–429.

[178] J. Greene, “Intel Trusted Execution Technology: Hardware-based Technology for Enhancing Server Platform Security,” Intel White Paper, Tech. Rep., 2010.

[179] I. Anati, S. Gueron, S. P. Johnson, and V. R. Scarlata, “Innovative Technology for CPU Based Attestation and Sealing,” Intel White Paper, Tech. Rep., 2013.

[180] “ARM Security Technology: building a secure system using trustzone technology,” Tech. Rep., 2009.

[181] AMD, “AMD Secure Technology,” http://www.amd.com/en-gb/innovations/software-technologies/security.

[182] A. Mosayyebzadeh, A. Mohan, S. Tikale, M. Abdi, N. Schear, T. Hudson, C. Munson, L. Rudolph, G. Cooperman, P. Desnoyers, and O. Krieger, “Supporting security sensitive tenants in a bare-metal cloud,” in *USENIX Annual Technical Conference*, Jul. 2019, pp. 587–602.

[183] “gVisor,” https://github.com/google/gvisor.

[184] A. Enright, M. Bentley, M. Radwan, E. Nono, K. Squizzato, and A. Pinon, “An Introduction to Storage for Docker Enterprise,” 2020, https://success.docker.com.

[185] T. Harter, B. Salmon, R. Liu, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, “Slacker: Fast Distribution with Lazy Docker Containers,” in *USENIX Conference on File and Storage Technologies*. Santa Clara, CA: USENIX Association, Feb. 2016, pp. 181–195.

[186] C. Zheng, L. Rupprecht, V. Tarasov, D. Thain, M. Mohamed, D. Skourtis, A. Warke, and D. Hildebrand, “Wharf: Sharing Docker Images in a Distributed File System,” in *ACM Symposium on Cloud Computing*. New York, NY: ACM, Oct. 2018, pp. 174–185.

[187] “Microsoft azure,” 2020, https://azure.microsoft.com.

[188] “Amazon web services,” 2020, https://aws.amazon.com/.

[189] S. R. Kleiman, “Vnodes: An architecture for multiple file system types in sun unix,” 1986, pp. 238–247.

[190] S. Boyd-Wickizer, M. F. Kaashoek, R. Morris, and N. Zeldovich, “Non-scalable locks are dangerous,” in *Ottawa Linux Symposium*, Ottawa, Canada, 2012.

[191] C. M. Kirsch, M. Lippautz, and H. Payer, "Fast and scalable, lock-free k-fifo queues," in *International Conference on Parallel Computing Technologies*, Sep. 2013, pp. 208–223.

[192] A. T. Clements, M. F. Kaashoek, N. Zeldovich, R. T. Morris, and E. Kohler, "The scalable commutativity rule: Designing scalable software for multicore processors," *ACM Transactions on Computer Systems*, vol. 32, no. 4, Jan. 2015.

[193] D. Jordan, "lru_lock scalability and smp list functions," Sep. 2018, https://lwn.net/Articles/764545/.

[194] P. E. McKenney, J. Fernandes, S. Boyd-Wickizer, and J. Walpole, "RCU Usage In the Linux Kernel: Eighteen Years Later," *SIGOPS Oper. Syst. Rev.*, vol. 54, no. 1, p. 47–63, Aug. 2020.

[195] D. Mazières, "A toolkit for user-level file systems," in *USENIX Annual Technical Conference*. Berkeley, CA: USENIX Association, 2001, pp. 261–274.

[196] A. Kantee, "Rump File Systems: Kernel Code Reborn," in *USENIX Annual Technical Conference*. San Diego, CA: USENIX Association, Jun. 2009, pp. 201–214.

[197] M. DeBergalis, P. Corbett, S. Kleiman, A. Lent, D. Noveck, T. Talpey, and M. Wittle, "The Direct Access File System," in *USENIX FAST*, San Francisco, CA, USA, 2003, pp. 175–188.

[198] H. Volos, S. Nalli, S. Panneerselvam, V. Varadarajan, P. Saxena, and M. M. Swift, "Aerie: Flexible File-system Interfaces to Storage-class Memory," in *ACM European Conference on Computer Systems*. New York, NY: ACM, Apr. 2014, pp. 14:1–14:14.

[199] C. Gruenwald, F. Sironi, M. F. Kaashoek, and N. Zeldovich, "Hare: a file system for non-cache-coherent multicores," in *ACM EuroSys*, 2015.

[200] D. Zahka, B. Kocoloski, and K. Keahey, "Reducing Kernel Surface Areas for Isolation and Scalability," in *International Conference on Parallel Processing*, Kyoto, Japan, 2019.

[201] J. Edge, "Measuring container security," *LWN.net*, Dec. 2018.

[202] E. G. Young, P. Zhu, T. Caraza-Harter, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, “The True Cost of Containing: A gVisor Case Study,” in *USENIX Workshop on Hot Topics in Cloud Computing*, Renton, WA, Jul. 2019.

[203] L. Lu, Y. Zhang, T. Do, S. Al-Kiswany, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, “Physical Disentanglement in a Container-Based File System,” in *USENIX Symposium on Operating Systems Design and Implementation*. Broomfield, CO: USENIX Association, Oct. 2014, pp. 81–96.

[204] Q. Xu, M. Awasthi, K. T. Malladi, J. Bhimani, J. Yang, and M. Annavaram, “Performance Analysis of Containerized Applications on Local and Remote Storage,” in *IEEE MSST*, 2017.

[205] D. Williams, R. Koller, M. Lucina, and N. Prakash, “Unikernels as Processes,” in *ACM SoCC*, Carlsbad, CA, Oct. 2018, pp. 199–211.

[206] V. Tarasov, E. Zadok, and S. Shepler, “Filebench: A Flexible Framework for File System Benchmarking,” *USENIX ;login:*, vol. 41, no. 1, pp. 6–12, 2016, https://github.com/filebench/filebench/wiki.

[207] “Stress-ng,” 2018, http://kernel.ubuntu.com/~cking/stress-ng/.

[208] “RocksDB: A persistent key-value store for fast storage environments,” 2020, rocksdb.org.

[209] “Use the BTRFS storage driver,” https://docs.docker.com/storage/storagedriver/btrfs-driver/.

[210] J. Corbet, “Unprivileged filesystem mounts, 2018 edition,” May 2018, https://lwn.net/Articles/755593/.

[211] “Linux kernel library,” 2019, https://github.com/lkl/linux.

[212] Y. Zhu, T. Wang, K. Mohror, A. Moody, K. Sato, M. Khan, and W. Yu, “Direct-FUSE: Removing the Middleman for High-Performance FUSE File System Support,” in *International Workshop on Runtime and Operating Systems for Supercomputers*. New York, NY: ACM, Jun. 2018.

[213] A. Aghayev, S. Weil, M. Kuchnik, M. Nelson, G. R. Ganger, and G. Amvrosiadis, “File Systems Unfit as Distributed Storage Backends: Lessons from 10 Years of

Ceph Evolution," in *ACM Symposium on Operating Systems Principles*, 2019, pp. 353–369.

[214] L. Rizzo, "netmap: a novel framework for fast packet I/O," in *USENIX Annual Technical Conference*, Boston, MA, Jun. 2012, pp. 101–112.

[215] "OpenZFS," 2020, https://github.com/openzfs.

[216] "FUSE implementation for overlayfs," 2020, https://github.com/containers/fuse-overlayfs.

[217] "Redis," 2020, https://github.com/redis.

[218] "Gluster," https://www.gluster.org/.

[219] "aio - posix asynchronous i/o overview," 2020, http://man7.org/linux/man-pages/man7/aio.7.html.

[220] "musl libc," 2020, https://musl.libc.org/.

[221] H. Franke, R. Russell, and M. Kirkwood, "Fuss, Futexes and Furwocks: Fast Userlevel Locking in Linux," in *Ottawa Linux Symposium*, Ottawa, Canada, Jun. 2002, pp. 479–495.

[222] "Amd opteron 6378," 2012, https://www.amd.com/en/products/cpu/6378.

[223] C. B. Sears, "The elements of cache programming style," in *Annual Linux Showcase and Conference*. Atlanta, GA: USENIX Association, Oct. 2000, pp. 283–298.

[224] "Vaapi support for xine lib," 2012, https://github.com/huceke/xine-lib-vaapi.

[225] "Intel® 64 and IA-32 Architectures Software Developer's Manual: Volume 2 Instruction Set Reference, Intel Corporation," Sep. 2016.

[226] M. M. Michael and M. L. Scott, "Simple, Fast and Practical Non-Blocking and Blocking Concurrent Queue Algorithms," in *ACM Symposium on Principles of Distributed Computing*. New York, NY: ACM, May 1996, pp. 267–275.

[227] W. N. Scherer, D. Lea, and M. L. Scott, "Scalable Synchronous Queues," in *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, New York, NY, Mar. 2006, pp. 147–156.

[228] B. Kerbl, M. Kenzel, J. H. Mueller, D. Schmalstieg, and M. Steinberger, “The Broker Queue: A Fast, Linearizable FIFO Queue for Fine-Granular Work Distribution on the GPU,” in *International Conference on Supercomputing*. New York, NY: ACM, Jun. 2018, pp. 76–85.

[229] A. Morrison and Y. Afek, “Fast Concurrent Queues for x86 Processors,” in *ACM Symposium on Principles and Practice of Parallel Programming*. Shenzhen, China: ACM, Feb. 2013, pp. 103–112.

[230] J. Kopinsky, “Relaxed Concurrent Ordering Structures,” Ph.D. dissertation, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, Jun. 2018.

[231] C. Yang and J. Mellor-Crummey, “A Wait-free Queue as Fast as Fetch-and-Add,” in *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. New York, NY: ACM, Mar. 2016, pp. 16:1–16:13.

[232] 2019, https://github.com/containers/fuse-overlayfs/issues/141.

[233] “Cloc - Count Lines of Code,” https://github.com/AlDanial/cloc.

[234] “Mercury,” 2018, https://mercury-hpc.github.io/.

[235] J. Soumagne, D. Kimpe, J. Zounmevo, M. Chaarawi, Q. Koziol, A. Afsahi, and R. Ross, “Mercury: Enabling remote procedure call for high-performance computing,” in *IEEE International Conference on Cluster Computing*. Piscataway, NJ: IEEE, Sep. 2013, pp. 1–8.

[236] V. Tarasov, D. Hildebrand, G. Kuenning, and E. Zadok, “Virtual machine workloads: The case for new benchmarks for NAS,” in *USENIX Conf. File and Storage Technologies*. Berkeley, CA: USENIX Association, Feb. 2013, pp. 307–320.

[237] B. C. Kuszmaul, M. Frigo, J. M. Paluska, and A. S. Sandler, “Everyone Loves File: File Storage Service (FSS) in Oracle Cloud Infrastructure,” in *USENIX ATC*, 2019.

[238] R. Podgorny, “unionfs-fuse,” 2017, https://github.com/rpodgorny/unionfs-fuse.

[239] “Ceph Client Config Reference,” http://docs.ceph.com/docs/master/cephfs/client-config-ref/.

[240] "Documentation for /proc/sys/vm/*," https://www.kernel.org/doc/Documentation/sysctl/vm.txt.

[241] Y. Sun, D. Safford, M. Zohar, D. Pendarakis, Z. Gu, and T. Jaeger, "Security Namespace: Making Linux Security Frameworks Available to Containers," in *USENIX Security Symposium*, 2018.

[242] "Scriptable database and system performance benchmark," https://github.com/akopytov/sysbench.

[243] "Lighttpd fly light," https://www.lighttpd.net/.

[244] "wrk - a HTTP benchmarking tool," 2019, https://github.com/wg/wrk.

[245] "GAP Benchmark Suite," 2019, https://github.com/sbeamer/gapbs.

[246] "AUFS," https://en.wikipedia.org/wiki/Aufs.

[247] P. Donnelly, "break client_lock," 2018, https://tracker.ceph.com/issues/23844.

[248] R. Santana, R. Rangaswami, V. Tarasov, and D. Hildebrand, "A Fast and Slippery Slope for File Systems," in *INFLOW*, no. 15, Monterey, CA, Oct. 2015.

[249] Y. Afek, G. Korland, and E. Yanovsky, "Quasi-linearizability: Relaxed consistency for improved concurrency," in *Principles of Distributed Systems*, 2010, pp. 395–410.

[250] A. Haas, M. Lippautz, T. A. Henzinger, H. Payer, A. Sokolova, C. M. Kirsch, and A. Sezgin, "Distributed queues in shared memory: Multicore performance and scalability through quantitative relaxation," in *ACM International Conference on Computing Frontiers*, May 2013.

[251] A. Haas, T. A. Henzinger, A. Holzer, C. M. Kirsch, M. Lippautz, H. Payer, A. Sezgin, A. Sokolova, and H. Veith, "Local linearizability for concurrent container-type data structures," in *International Conference on Concurrency Theory*, Aug. 2016, pp. 6:1–6:15.

[252] A. Rukundo, A. Atalar, and P. Tsigas, "Monotonically relaxing concurrent data-structure semantics for increasing performance: An efficient 2d design framework," in *International Symposium on Distributed Computing*, Oct. 2019, pp. 31:1–31:15.

[253] J. D. McCalpin, “Notes on ”non-temporal” (aka ”streaming”) stores,” Jan. 2018, https://sites.utexas.edu/jdm4372.

[254] *Intel©64 and IA-32 Architectures Software Developer’s Manual*. Intel Corp., May 2020, vol. 1, order No: 325462-072US.

[255] G. Hager, “The McCalpin STREAM benchmark: How to do it right and interpret the results,” Mar. 2019, https://blogs.fau.de/hager/archives/8263.

[256] J. D. McCalpin, “How does WC-buffer relate to LFB?” Jan. 2020, https://community.intel.com/.

[257] N. Kurz, “Does software prefetching allocate a Line Fill Buffer (LFB)?” Oct. 2013, https://stackoverflow.com/q/19472036.

[258] J. D. McCalpin, “Optimizing AMD Opteron Memory Bandwidth, Parts 1-5,” Nov. 2010, https://sites.utexas.edu/jdm4372/.

[259] *Intel©64 and IA-32 Architectures Optimization Reference Manual*. Intel Corporation, Sep. 2019, order No. 248966-042b.

[260] *Software Optimization Guide for AMD Family 19h Processors*. Advanced Micro Devices, Inc., Nov. 2020, publication No. 56665.

[261] P. Sewel, S. Sarkar, S. Owens, F. Z. Nardelli, and M. O. Myreen, “x86-TSO: A Rigorous and Usable Programmer’s Model for x86 Multiprocessors,” *Communications of the ACM*, pp. 87–97, May 2010.

[262] W. N. Scherer and M. L. Scott, “Nonblocking concurrent data structures with condition synchronization,” in *Intl. Conf. Distributed Computing*, 2004, pp. 174–187.

[263] D. Dice, “Ptlqueue : a scalable bounded-capacity mpmc queue,” Jun. 2014, https://blogs.oracle.com/dave/ptlqueue-:-a-scalable-bounded-capacity-mpmc-queue.

[264] “Atomic operations library (memory_order),” 2017, https://en.cppreference.com/w/c/atomic/memory_order.

[265] “Intel® 64 and IA-32 Architectures Software Developer’s Manual: Volume 2 Instruction Set Reference, Intel Corporation,” Sep. 2016.

[266] H. Golestani, A. Mirhosseini, and T. F. Wenisch, "Software data planes: You can't always spin to win," in *ACM Symposium on Cloud Computing*, Nov. 2019, pp. 337–350.

[267] "Intel® 64 Architecture Memory Ordering White Paper," Aug. 2007.

[268] P. Tsigas and Y. Zhang, "A simple, fast and scalable non-blocking concurrent fifo queue for shared memory multiprocessor systems," in *ACM Symposium on Parallel Algorithms and Architectures*, Crete Island, Greece, Jan. 2001, pp. 134–143.

[269] N. Shafiei, "Non-blocking array-based algorithms for stacks and queues," in *International Conference on Distributed Computing and Networking*, Hyderabad, India, Jan. 2009, pp. 55–66.

[270] M. P. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Trans. Program. Lang. Syst.*, vol. 12, no. 3, pp. 463–492, Jul. 1990.

[271] "tcmalloc," 2020, https://google.github.io/tcmalloc/.

[272] A. Haas, T. Hütter, C. M. Kirsch, M. Lippautz, M. Preishuber, and A. Sokolova, "Scal: A benchmarking suite for concurrent data structures," in *International Conference on Networked Systems*, May 2015, pp. 1–14.

[273] C. Yang, "A benchmark framework for concurrent queue implementations," 2020, https://github.com/chaoran/fast-wait-free-queue.

[274] Nathan_K_3, "Single threaded memory bandwidth on Sandy Bridge," Oct. 2013, https://community.intel.com/.

[275] J. Anick, "Writing fast memcpy() functions on x86 platforms," https://joryanick.com/.

[276] "Copying accelerated video decode frame buffers," Sep. 2009, https://software.intel.com/.

[277] M. Popoloski, "Demystifying SSE move instructions," Jun. 2011, https://www.gamedev.net/blog/.

[278] K. Viswanathan, “Intel©memory latency checker v3.9,” Jun. 2020, https://software.intel.com/content/www/us/en/develop/articles/intelr-memory-latency-checker.html.

[279] “Linux v5.10.12,” https://elixir.bootlin.com/linux/v5.10.12/.

[280] J. Tunney, “cosmopolitan libc: your build-once run-anywhere c library,” https://justine.lol/cosmopolitan/.

[281] S. Pal, A. K. Mandal, and A. Sarkar, “Application multi-tenancy for software as a service,” *ACM SIGSOFT Softw. Eng. Notes*, vol. 40, no. 2, Apr. 2015.

[282] C. J. Guo, W. Sun, Y. Huang, Z. H. Wang, and B. Gao, “A framework for Native Multi-tenancy Application Development and Management,” in *IEEE Int. Conf. on E-Commerce Technology and Int. Conf. on Enterprise Computing, E-Commerce and E-Services*, Broomfield, CO, USA, 2014, pp. 1–16.

[283] I. Odun-Ayo, S. Misra, O. Abayomi-Alli, and O. Ajayi, “Cloud Multi-Tenancy: Issues and Developments,” in *Companion Proceedings of the 10th International Conference on Utility and Cloud Computing*, ser. IEEE, ACM Compainion, Int. conf on Utility and Cloud Computing, Austin, TX, USA, 2017, pp. 209–214.

[284] L. Popa, M. Yu, S. Y. Ko, S. Ratnasamy, and I. Stoica, “CloudPolice: Taking Access Control out of the Network,” in *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, ser. Hotnets-IX, Monterey, CA, USA, 2010, pp. 7:1–7:6.

[285] Y. Mundada, A. Ramachandran, and N. Feamster, “SilverLine: Data and network isolation for cloud services,” in *HotCloud ’11: Proceedings of the 2011 USENIX HotCloud Workshop*, Portland, OR, USA, Jun. 2011.

[286] T. Wood, A. Gerber, K. K. Ramakrishnan, P. Shenoy, and J. V. der Merwe, “The case for enterprise-ready virtual private clouds,” in *Usenix HotCloud*, 2009.

[287] E. J. Jackson, M. Walls, A. Panda, J. Pettit, B. Pfaff, J. Rajahalme, T. Koponen, and S. Shenker, “SoftFlow: A Middlebox Architecture for Open vSwitch,” in *Usenix HotCloud*, Denver, CO, USA, Jun. 2016.

[288] D. Muthukumaran, D. O'Keeffe, C. Priebe, D. Eyers, B. Shand, and P. Pietzuch, "FlowWatcher: defending against data disclosure vulnerabilities in web applications," in *ACM Conf. on Computer and Communications Security*, Denver, CO, USA, Oct. 1995, pp. 603–615.

[289] N. H. Walfield, P. T. Stanton, J. L. Griffin, and R. Burns, "Practical protection for personal storage in the cloud," in *EuroSec Security Workshop*, Paris, France, Apr. 2010, pp. 8–14.

[290] https://aws.amazon.com/efs/.

[291] J. Darcy, "HekaFS," http://hekafs.org.

[292] https://github.com/kozyraki/phoenix.

[293] https://wiki.openstack.org/wiki/Manila.

[294] http://docs.openstack.org/developer/keystone/.

[295] https://aws.amazon.com/documentation/iam/.

[296] D. Thain, "Identity boxing: A new technique for consistent global identity," in *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, Seattle, WA, USA, Nov. 2005.

[297] D. Thain, C. Moretti, P. Madrid, P. Snowberger, and J. Hemmes, "The consequences of decentralized security in a cooperative storage system," in *SISW '05: Proceedings of the Third IEEE International Security in Storage Workshop*, San Francisco, CA, USA, Dec. 2005, pp. 71–82.

[298] A. Madhavapeddy, T. Leonard, M. Skjegstad, T. Gazagnaire, D. Sheets, D. Scott, R. Mortier, A. Chaudhry, B. Singh, J. Ludlam, J. Crowcroft, and I. Leslie, "Jitsu: just-in-time summoning of unikernels," in *USENIX Symp. on Networked Systems Design and Implementation*, Oakland, CA, USA, May 2015, pp. 559–573.

[299] C. Chen, H. Raj, S. Saroiu, and A. Wolman, "cTPM: a cloud TPM for cross-device trusted applications," in *USENIX Symp. on Networked Systems Design and Implementation*, Seattle, WA, USA, Apr. 2014, pp. 187–201.

[300] M. Mihailescu, G. Soundararajan, and C. Amza, "Mixapart: Decoupled analytics for shared storage systems," in *USENIX Conf. on File and Storage Technologies*, San Jose, CA, USA, Feb. 2013, pp. 133–146.

[301] W. Arthur and D. Challener, *A Practical Guide to TPM 2.0 Using the Trusted Platform Module in the New Age of Security*. Apress, Jan. 2015.

[302] S. Berger, R. Cáceres, K. A. Goldman, R. Perez, R. Sailer, and L. van Doorn, "vTPM: virtualizing the trusted platform module," in *USENIX Security Symp.*, Vancouver, Canada, Jul. 2006, pp. 305–320.

[303] A. W. Leung, E. L. Miller, and S. Jones, "Scalable Security for Petascale Parallel File Systems," in *ACM/IEEE Conf. Supercomputing*, Nov. 2007, pp. 16:1–16:12.

[304] Y. Li, N. S. Dhotre, Y. Ohara, T. M. Kroeger, E. L. Miller, and D. D. E. Long, "Horus: Fine-grained encryption-based security for large-scale storage," in *USENIX Conf. on File and Storage Technologies*, San Jose, CA, USA, Feb. 2013, pp. 147–160.

[305] N. Pustchi and R. Sandhu, "MT-ABAC: a multi-tenant attribute-based access control model with tenant trust," in *Intl Conf. on Network and System Security*, New York, NY, USA, Nov. 2015, Springer LNCS 9408.

[306] R. Yeluri and E. Castro-Leon, *Building the Infrastructure for Cloud Security A Solutions View*. Apress, Mar. 2014.

[307] J. Mace, P. Bodik, R. Fonseca, and M. Musuvathi, "Retro: targeted resource management in multi-tenant distributed systems," in *USENIX Symp. on Networked Systems Design and Implementation*, Oakland, CA, USA, May 2015, pp. 589–603.

[308] A. Jøsan, M. A. Zomai, and S. Suriadi, "Usability and Privacy in Identity Management Architectures," in *Australasian Information Security Workshop: Privacy Enhancing Technologies*, Ballarat, Australia, Jan. 2007, pp. 143–152.

[309] R. Alfieri, R. Cecchini, V. Ciaschini, L. dell' Agnello, A. Frohner, K. Lorentey, and F. Spataro, "From gridmap-file to VOMS: managing authorization in a grid environment," *Future Generation Computer Systems (Elsevier)*, vol. 21, pp. 549–558, 2005.

[310] M. Kaminsky, G. Savvides, D. Mazières, and M. F. Kaashoek, “Decentralized user authenication in a global file system,” in *ACM Symp. Operating Systems Principles*, Bolton Landing, NY, USA, Oct. 2003, pp. 60–73.

[311] B. C. Neuman and T. Ts’o, “Kerberos: An authentication service for computer networks,” *IEEE Communications Magazine*, vol. 32, no. 9, pp. 33–38, Sep. 1994.

[312] D. K. Smetters and N. Good, “How users use access control,” in *Symp. on Usable Privacy and Security*, Mountain View, CA, USA, Jul. 2009.

[313] S. Bouchenak, G. Chockler, H. Chockler, G. Gheorghe, N. Santos, and A. Shraer, “Verifying cloud services: Present and future,” *SIGOPS Oper. Syst. Rev.*, vol. 47, no. 2, pp. 6–19, Jul. 2013.

[314] http://sourceforge.net/projects/mdtest/.

[315] Porter, Donald E. and Boyd-Wickizer, Silas and Howell, Jon and Olinsky, Reuben and Hunt, Galen C., “Rethinking the Library OS from the Top Down,” in *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVI, 2011, p. 291–304.

[316] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift, “Peeking Behind the Curtains of Serverless Platforms,” in *USENIX ATC*, 2018.

[317] J. Khalid, E. Rozner, W. Felter, C. Xu, K. Rajamani, A. Ferreira, and A. Akella, “Iron: Isolating Network-based CPU in Container Environments,” in *USENIX Symposium on Networked Systems Design and Implementation*, Renton, WA, Apr. 2018, pp. 313–328.

[318] L. Shalev, J. Satran, E. Borovik, and M. Ben-Yehuda, “IsoStack - Highly Efficient Network Processing on Dedidated Cores,” in *USENIX Annual Technical Conference*, Boston, MA, USA, Jun. 2010, pp. 61–74.

[319] R. Nikolaev and G. Back, “VirtuOS: An Operating System with Kernel Virtualization,” in *ACM SOSP*, Farminton, PA, USA, 2013, pp. 116–132.

[320] S. Chen, C. Delimitrou, and J. F. Martínez, “PARTIES: QoS-Aware Resource Partitioning for Multiple Interactive Services,” in *International Conference on Ar-*

chitectural Support for Programming Languages and Operating Systems. Providence, RI: ACM, Apr. 2019, pp. 107–120.

[321] Z. Shen, Z. Sun, G.-E. Sela, E. Bagdasaryan, C. Delimitrou, R. Van Renesse, and H. Weatherspoon, "X-Containers: Breaking Down Barriers to Improve Performance and Isolation of Cloud-Native Containers," in *International Conference on Architectural Support for Programming Languages and Operating Systems*. New York, NY: ACM, Apr. 2019, pp. 121–135.

[322] W. Cao, Z. Liu, P. Wang, S. Chen, C. Zhu, S. Zheng, Y. Wang, and G. Ma, "PolarFS: An Ultra-low Latency and Failure Resilient Distributed File System for Shared Storage Cloud Database," in *Proc. VLDB Endowment*, 2018, pp. 1849–1862.

[323] A. Bijlani and U. Ramachandran, "Extension framework for file systems in user space," in *USENIX Annual Technical Conference*. Berkeley, CA: USENIX Association, Jul. 2019, pp. 121–134.

[324] E. Zadok and J. Nieh, "Fist: A language for stackable file systems," in *USENIX Annual Technical Conference*. Berkeley, CA: USENIX Association, Jun. 2000, pp. 55–77.

[325] D. Zhuo, K. Zhang, Y. Zhu, H. H. Liu, M. Rockett, A. Krishnamurthy, and T. Anderson, "Slim: OS kernel support for a low-overhead container overlay network," in *USENIX NSDI*, 2019, pp. 331–344.

[326] D. Kim, T. Yu, H. H. Liu, Y. Zhu, J. Padhye, S. Raindel, C. Guo, V. Sekar, and S. Seshan, "Freeflow: Software-based virtual RDMA networking for containerized clouds," in *USENIX NSDI*, 2019.

[327] J. Axboe, "Efficient IO with io_uring," Oct. 2019, version 0.4.

[328] L. Vilanova, M. Jordà, N. Navarro, Y. Etsion, and M. Valero, "Direct interprocess communication (dipc): Repurposing the codoms architecture to accelerate ipc," in *ACM European Conference on Computer Systems*. New York, NY: ACM, Apr. 2017, pp. 16–31.

[329] A. Kalia, M. Kaminsky, and D. Andersen, “Datacenter rpcs can be general and fast,” in *USENIX Symposium on Networked Systems Design and Implementation*. Berkeley, CA: USENIX Association, Feb. 2019, pp. 1–16.

[330] A. Menon, A. L. Cox, and W. Zwaenepoel, “Optimizing network virtualization in xen,” in *USENIX Annual Technical Conference*. Berkeley, CA: USENIX Association, May 2006, pp. 15–28.

[331] D. S. Ritchie and G. W. Neufeld, “User level ipc and device management in the raven kernel,” in *USENIX Microkernels and Other Kernel Architectures Symposium*. Berkeley, CA: USENIX Association, Sep. 1993, pp. 111–126.

[332] O. Ostrovsky and A. Morrison, “Scaling concurrent queues by using htm to profit from failed atomic operations,” in *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, San Diego, CA, 2020, pp. 89–101.

[333] C.-H. Shann, T.-L. Huang, and C. Chen, “A practical nonblocking queue algorithm using compare-and-swap,” in *International Conference on Parallel and Distributed Systems*, Iwate, Japan, Jul. 2000, pp. 470–476.

[334] C. Evequoz, “Non-Blocking Concurrent FIFO Queues With Single Word Synhronization Primitives,” in *International Conference on Parallel Processing*, Portland, OR, Sep. 2008, pp. 397–405.

[335] P. Fatourou and N. D. Kallimanis, “Revisiting the Combining Synchronization Technique,” in *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, New Orleans, LA, Feb. 2012, pp. 257–266.

[336] T. A. Henzinger, C. M. Kirsch, H. Payer, A. Sezgin, and A. Sokolova, “Quantitative relaxation of concurrent data structures,” in *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2013, pp. 317–28.

[337] M. Wimmer, J. Gruber, J. L. Träff, and P. Tsigas, “The lock-free k-lsm relaxed priority queue,” in *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Feb. 2015, pp. 277–278.

[338] H. Rihani, P. Sanders, and R. Dementiev, “Brief announcement: Multiqueues: Simple relaxed concurrent priority queues,” in *ACM Symposium on Parallelism in Algorithms and Architectures*, Jun. 2015, pp. 80–82.

[339] D. Alistarh, J. Kopinsky, J. Li, and N. Shavit, “The spraylist: A scalable relaxed priority queue,” in *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. New York, NY: ACM, Feb. 2015, pp. 11–20.

[340] N. Shavit and G. Taubenfeld, “The computability of relaxed data structures: queues and stacks as examples,” *Distributed Computing*, vol. 29, no. 5, pp. 395–407, Oct. 2016.

[341] T. Zhou, M. Michael, and M. Spear, “A Practical, Scalable, Relaxed Priority Queue,” in *International Conference on Parallel Processing*, Kyoto, Japan, Aug. 2019, pp. 57:1–57:10.

[342] Y. Afek, G. Korland, M. Natanzon, and N. Shavit, “Scalable producer-consumer pools based on elimination-diffraction trees,” in *Intl Euro-Par Conference*, Ischia, Italy, Aug. 2010, pp. 151–162.

[343] H. Sundell, A. Gidenstam, M. Papatriantafilou, and P. Tsigas, “A lock-free algorithm for concurrent bags,” in *ACM Symposium on Parallelism in Algorithms and Architectures*, San Jose, CA, Jun. 2011, pp. 335–344.

[344] D. Dice and O. Otenko, “Brief announcement: Multilane - a concurrent blocking multiset,” in *ACM Symposium on Parallelism in Algorithms and Architectures*. New York, NY: ACM, Jun. 2011, pp. 313–314.

[345] J. Gruber, J. L. Träff, and M. Wimmer, “Brief announcement: Benchmarking concurrent priority queues,” in *ACM Symposium on Parallelism in Algorithms and Architectures*, Jul. 2016, pp. 361–362.

[346] C. L. Alapat, J. Hofmann, G. Hager, H. Fehske, A. R. Bishop, and G. Wellein, “Understanding HPC benchmark performance on Intel Broadwell and Cascade Lake processors,” in *International Conference on High Performance Computing*. Frankfurt/Main, Germany: Springer, Cham, Jun. 2020, pp. 412–433, lNCS, vol. 12151.

[347] M. Gottschlag, T. Schmidt, and F. Bellosa, “Avx overhead profiling: How much does your fast code slow you down?” in *ACM SIGOPS Asia-Pacific Workshop on Systems*, Tsukuba, Japan, Aug. 2020, pp. 59–66.

[348] G. A. Gibson, D. F. Nagle, K. Amiri, F. W. Chang, E. M. Feinberg, H. Gobioff, C. Lee, B. Ozceri, E. Riedel, D. Rochberg, and J. Zelenka, "File server scaling with network-attached secure disks," in *ACM SIGMETRICS Conf.*, Seattle, WA, USA, 1997, pp. 272–284.

[349] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu, "Plutus: Scalable secure file sharing on untrusted storage," in *USENIX Conf. on File and Storage Technologies*, San Francisco, CA, USA, 2003, pp. 29–42.

[350] J. Walgenbach, S. C. Simms, J. P. Miller, and K. Westneat, "Enabling Lustre WAN for Production Use on the TeraGrid: A Lightweight UID Mapping Scheme," in *TeraGrid Conf.*, Pittsburgh, PA, Aug. 2010.

[351] G. Margaritis, A. Hatzieleftheriou, and S. V. Anastasiadis, "Nepheli: Scalable access control for federated file services," *J. Grid Comp.*, vol. 11, no. 1, pp. 83–102, Mar. 2013.

[352] C. Olson and E. L. Miller, "Secure capabilities for a petabyte-scale object-based distributed file system," in *ACM Workshop on Storage Security and Survivability*, Fairfax, VA, USA, Nov. 2005, pp. 64–73.

[353] Z. Niu, K. Zhou, H. Jiang, D. Feng, and T. Yang, "IDEAS: an identity-based security architecture for large-scale and high-performance storage systems," University of Nebraska-Lincoln, Tech. Rep., Nov. 2008, tR-UNL-CSE-2008-0013.

[354] J. Kelley, R. Tamassia, and N. Triandopoulos, "Hardening access control and data protection in GFS-like file systems," in *ESORICS Symp.*, Pisa, Italy, Sep. 2012, pp. 19–36, Springer LNCS 7459.

[355] J. Bethencourt, A. Sahai, and B. Waters, "Ciphertext-Policy Attribute-Based Encryption," in *IEEE Symp. on Security and Privacy*, Berkeley, CA, USA, May 2007, pp. 321–334.

[356] C. Ngo, Y. Demchenko, and C. de Laat, "Multi-tenant attribute-based access control for cloud infrastructure services," *Journal of Information Security and Applications*, vol. 27-28, pp. 65–84, Apr. 2016.

[357] D. Hilderbrand, A. Povzner, R. Tewari, and V. Tarasov, "Revisiting the storage stack in virtualized nas environments," in *USENIX Workshop on I/O Virtualization*, Portland, OR, USA, Jun. 2011.

[358] E. G. Sirer, W. de Bruijn, P. Reynolds, A. Shieh, K. Walsh, D. Williams, and F. B. Schneider, "Logical Attestation: an authorization architecture for trustworthy computing," in *ACM Symp. on Operating Systems Principles*, Cascais, Portugal, Oct. 2011, pp. 249–264.

[359] R. A. Popa, J. R. Lorch, D. Molnar, H. J. Wang, and L. Zhuang, "Enabling Security in Cloud Storage SLAs with CloudProof," in *USENIX Annual Technical Conf.*, Portland, OR, USA, Jun. 2011, pp. 355–368.

[360] N. Santos, R. Rodrigues, K. P. Gummadi, and S. Saroiu, "Policy-Sealed Data: A New Abstraction for Building Trusted Cloud Services," in *USENIX Security Symp.*, Bellevue, WA, USA, Aug. 2012, pp. 175–188.

[361] J. R. Lorch, B. Parno, J. Mickens, M. Raykova, and J. Schiffman, "Shroud: ensuring access to large-scale data in the data center," in *USENIX Conf. on File and Storage Technologies*, San Jose, CA, USA, Feb. 2013, pp. 199–213.

[362] J. Weiner, "PSI - Pressure Stall Information," Apr. 2018, https://www.kernel.org/doc/html/latest/accounting/psi.html/.

[363] H. S. Gunawi, M. Hao, T. Leesatapornwongsa, T. Patana-anake, T. Do, J. Adityatama, K. J. Eliazar, A. Laksono, J. F. Lukman, V. Martin, and A. D. Satria, "What bugs live in the cloud? a study of 3000+ issues in cloud systems," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SOCC '14. New York, NY, USA: ACM, 2014, pp. 1–14.

[364] O. A. Wahab, J. Bentahar, H. Otrok, and A. Mourad, "I know you are watching me: Stackelberg-based adaptive intrusion detection strategy for insider attacks in the cloud," in *2017 IEEE International Conference on Web Services (ICWS)*, 2017, pp. 728–735.

[365] J. Aikat, A. Akella, J. S. Chase, A. Juels, M. K. Reiter, T. Ristenpart, V. Sekar, and M. Swift, "Rethinking security in the era of cloud computing," *IEEE Security Privacy*, vol. 15, no. 3, pp. 60–69, 2017.

[366] X. Gao, Z. Gu, M. Kayaalp, D. Pendarakis, and H. Wang, “Containerleaks: Emerging security threats of information leakages in container clouds,” in *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2017, pp. 237–248.

[367] Intel, “Intel Software Guard Extensions,” https://software.intel.com/content/www/us/en/develop/topics/software-guard-extensions.html, accessed: 2021-04-10.

Author's Publications

1. Giorgos Kappes, Stergios V. Anastasiadis, A Lock-free Relaxed Concurrent Queue for Fast Work Distribution, ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), Virtual Event, Republic of Korea, February 2021, pp. 454-456.

2. Giorgos Kappes, Stergios V. Anastasiadis, A user-level Toolkit for Storage I/O Isolation on Multitenant Hosts, ACM Symposium on Cloud Computing (SoCC), Virtual Event, USA, October 2020, pp. 74-89.

3. Giorgos Kappes, Stergios V. Anastasiadis, Libservices: Dynamic Storage Provisioning for Multitenant I/O Isolation, ACM SIGOPS Asia-Pacific Workshop on Systems (APSys), Tsukuba, Japan, August 2020, pp. 33-41.

4. Giorgos Kappes, Andromachi Hatzieleftheriou, Stergios V. Anastasiadis, Multitenant Access Control for Cloud-Aware Distributed Filesystems, IEEE Transactions on Dependable and Secure Computing (TDSC), Nov-Dec 2019, pg. 1070-1085.

5. Giorgos Kappes, Andromachi Hatzieleftheriou, Stergios V. Anastasiadis, Virtualization-aware Access Control for Multitenant Filesystems, IEEE International Conference on Massive Storage Systems and Technology (MSST), Santa Clara, CA, USA, June 2014, pp. 1-6.

Short Biography

Giorgos Kappes was born in Ioanina, Greece, in 1987. He holds an M.Sc degree in Computer Systems and a B.Sc degree from the Department of Computer Science and Engineering of the University of Ioannina, Greece. Since 2014, he has been a Ph.D candidate in the same Deparment under the supervision of Prof. Stergios Anastasiadis. His research includes the design, development, and evaluation of system software for multitenant cloud environments. His research interests include operating systems, data storage, and systems security.