Fast and Efficient, Durable Storage in Local and Distributed Filesystems

Andromachi Hatzieleftheriou

Ph.D. Dissertation

—♦—

Ioannina, September 2015

ΤΜΗΜΑ ΜΗΧΑΝΙΚΩΝ Η/Υ & ΠΛΗΡΟΦΟΡΙΚΗΣ
ΠΑΝΕΠΙΣΤΗΜΙΟ ΙΩΑΝΝΙΝΩΝ

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
UNIVERSITY OF IOANNINA

Γρήγορη και Αποδοτική, Ανθεκτική Αποθήκευση σε Τοπικά και Κατανεμημένα Συστήματα Αρχείων

Η ΔΙΔΑΚΤΟΡΙΚΗ ΔΙΑΤΡΙΒΗ

υποβάλλεται στην

ορισθείσα από την Γενική Συνέλευση Ειδικής Σύνθεσης

Τμήματος Μηχανικών Η/Υ και Πληροφορικής
Εξεταστική Επιτροπή

από την

Ανδρομάχη Χατζηελευθερίου

ως μέρος των Υποχρεώσεων για τη λήψη του

ΔΙΔΑΚΤΟΡΙΚΟΥ ΔΙΠΛΩΜΑΤΟΣ ΣΤΗΝ ΠΛΗΡΟΦΟΡΙΚΗ

Σεπτέμβριος 2015

Τριμελής Συμβουλευτική Επιτροπή

- Στέργιος Αναστασιάδης, Αναπληρωτής Καθηγητής του Τμήματος Μηχανικών Η/Υ και Πληροφορικής του Πανεπιστημίου Ιωαννίνων
- Γεώργιος Μανής, Επίκουρος Καθηγητής του Τμήματος Μηχανικών Η/Υ και Πληροφορικής του Πανεπιστημίου Ιωαννίνων
- Ευαγγελία Πιτουρά, Καθηγήτρια του Τμήματος Μηχανικών Η/Υ και Πληροφορικής του Πανεπιστημίου Ιωαννίνων

Επταμελής Εξεταστική Επιτροπή

- Στέργιος Αναστασιάδης, Αναπληρωτής Καθηγητής του Τμήματος Μηχανικών Η/Υ και Πληροφορικής του Πανεπιστημίου Ιωαννίνων
- Γεώργιος Μανής, Επίκουρος Καθηγητής του Τμήματος Μηχανικών Η/Υ και Πληροφορικής του Πανεπιστημίου Ιωαννίνων
- Ευαγγελία Πιτουρά, Καθηγήτρια του Τμήματος Μηχανικών Η/Υ και Πληροφορικής του Πανεπιστημίου Ιωαννίνων
- Κωνσταντίνος Μαγκούτης, Επίκουρος Καθηγητής του Τμήματος Μηχανικών Η/Υ και Πληροφορικής του Πανεπιστημίου Ιωαννίνων
- Αλέξιος Δελής, Καθηγητής του Τμήματος Πληροφορικής και Τηλεπικοινωνιών του Εθνικού και Καποδιστριακού Πανεπιστημίου Αθηνών
- Antonio Cortes, Associate Professor Computer Architecture Department Universitat Politècnica de Catalunya, Barcelona, Spain
- Peter J. Varman, Professor Department of Electrical and Computer Engineering Rice University, Houston, Texas

Dedication

To my family and Nick,

for giving me the strength to tackle any challenge.

Acknowledgements

First of all, I would like to thank my advisor Prof. Stergios Anastasiadis, who inspired me to set high goals, and taught me how to achieve them. Through his systematic supervision and guidance, from the early stages to the last ones of the completion of this thesis, he gave me the opportunity to indulge in the scientific field of computer systems.

I would also like to thank the members of my examination committee, Prof. Evaggelia Pitoura, Prof. Konstantinos Magoutis, Prof. Georgios Manis, Prof. Alex Delis, Prof. Toni Cortes, and Prof. Peter J. Varman for their kind comments.

The deepest gratitude to my family for always believing in me. Especially, I would like to thank my parents, Thomas and Foteini, for their moral and financial support throughout these years. Also, sincere thanks to my brother Christos, my sister Alexandra, and little Thomas and Afroditi, for their love and understanding. Now at last I have plenty of time to share with them all!

Furthermore, I would like to thank my colleagues and friends Giorgos Margaritis, Andreas Vasilakis, Argyris Kalogeratos, and Dimitri Melissovas for sharing with me countless hours of interesting talks, scientific and non-scientific (i.e., politics, gossips, nagging, etc.). Special thanks to Thanasis Koufoulis, Christos Theodorakis, and Alek Papadogiannakis for their encouragement and the great time that we spent during the last year at Ioannina, having pleasant (and maybe some not so pleasant) conversations drinking tsipouro with Abdallah. I would not forget my close friends, Melisanthi Vakirtzi and Dimitra Tsintikidou, for the joyful moments that we had together during my short-term vacations in my home town, away from the terminal – indeed there were a few such times. Also, many thanks to Dimitris Karathanos for his 'exceptional' music choices. Last but definitely not least, I would like to thank Giorgos Kappes, Eirini Micheli, Vasileios Papadopoulos, all members of the Systems Research Group, for creating a pleasant and friendly environment at the office. Yet there was life beyond the walls of our cubicles!

Finally, I am especially grateful to Nikolaos Papanikos for his encouragement and unconditional support during this journey. This is also a good point to express my apologies to him for any 'unreasonable' nagging. The last ten years would have been less fascinating and much more difficult without him by my side.

This research was supported in part by the European Union (European Social Fund - ESF) and Greek national funds through the Operational Program "Education and Lifelong Learning" of the National Strategic Reference Framework (NSRF) - Research Funding Program: Thales. Investing in knowledge society through the European Social Fund (Project "Cloud9"). Finally, credit provided by AWS in Education Grant award to access the Amazon Web Services is gratefully acknowledged.

Table of Contents

List of Figures

List of Tables

List of Algorithms

Abstract

Andromachi Hatzieleftheriou T.
Phd, Department of Computer Science and Engineering, University of Ioannina, Greece.
September, 2015.
Fast and Efficient, Durable Storage in Local and Distributed Filesystems.
Thesis Supervisor: Stergios V. Anastasiadis.

The increasingly large amount of structured and unstructured data that needs to be constantly stored and processed, requires highly scalable storage system solutions. Typically, in a cloud environment the storage stack consists of multiple tiers. Front-end machines are mainly responsible for temporarily caching data, while back-end machines provide persistent storage. Additionally, data is redundantly stored among multiple servers for high availability in case of failures. However, the co-location of multiple workloads on top of a shared physical infrastructure introduces several new design challenges related to the *performance*, resource *efficiency* and *durability* of multi-tier cloud environments. In the present thesis, we argue that the inherent characteristics of multi-tier virtualized environments necessitate the fresh reconsideration of the I/O path. Across the different tiers of the storage stack, we investigate the tradeoff between consistency and resource efficiency, aiming to provide improved durability and high performance at moderate resource overhead.

In the first part of this thesis, we focus on the storage backend tier with the aim to combine improved local filesystem consistency with high performance and efficient storage bandwidth utilization. In general, synchronous small writes are commonly used to safely log recent state modifications for fast crash recovery. Demanding systems usually dedicate separate devices to logging for adequate performance during normal operation and redundancy during state reconstruction. Nevertheless, storage stacks enforce page-sized

granularity in data transfers from memory to disk. As a result, they consume excessive storage bandwidth to handle small writes, which hurts performance. The problem worsens, as filesystems often handle multiple concurrent streams, which effectively generate random I/O traffic.

In a local filesystem, we rely on journaling of both data and metadata blocks in order to achieve their safe transfer to disk at sequential disk throughput and low latency. We propose the design of two new mount modes, *wasteless journaling* and *selective journaling*. Wasteless journaling coalesces multiple concurrent subpage writes into page-sized journal blocks. Instead, selective journaling selectively journals data updates below a write threshold, and transfers the rest directly to the filesystem. We implement a functional prototype of our design over a widely-used local filesystem. Across a wide range of microbenchmarks and application-level workloads over standalone servers and a multi-tier networked system, we demonstrate that the proposed modes preserve filesystem consistency, and provide improved operation throughput along with reduced write latency and recovery time, at low storage bandwidth overhead.

In the second part of the present thesis, we focus on the frontend layer of a multi-tier environment. In particular, we examine the implications among performance, resource efficiency, and durability in scalable storage systems. Hardware consolidation in the datacenter occasionally leads to scalability bottlenecks due to the heavy utilization of critical resources, such as the shared network bandwidth. Host-side caching on durable media is already applied at the block level in order to reduce the load of the storage backend. However, block-level caching is often criticized for added overhead, and restricted data sharing across different hosts. During client crashes, writeback caching can also lead to unrecoverable loss of written data that was previously acknowledged as stable.

We improve the durability of shared storage in the datacenter by supporting journaling at the kernel-level client of a well-known object-based distributed filesystem. Storage virtualization at the file interface allows us to achieve clear consistency semantics across data and metadata blocks, support native file sharing between clients over the same or different hosts, and provide flexible configuration of the time period during which the data is durably staged at the host side. Over a prototype implementation, we demonstrate improved operation throughput at reduced disk and network bandwidth utilization for specific durability, across multiple microbenchmarks, application-level workloads, and

real-world applications on top of a local cluster setup and a large-scale public cloud environment.

Εκτεταμενη Περιληψη στα Ελληνικα

Ανδρομάχη Χατζηελευθερίου του Θωμά και της Φωτεινής.
PhD, Τμήμα Μηχανικών Η/Υ και Πληροφορικής, Πανεπιστήμιο Ιωαννίνων, Σεπτέμβριος, 2015.
Γρήγορη και Αποδοτική, Ανθεκτική Αποθήκευση σε Τοπικά και Κατανεμημένα Συστήματα Αρχείων.
Επιβλέπων: Στέργιος Β. Αναστασιάδης.

Στην εποχή της ψηφιακής πληροφορίας, τα κλιμακώσιμα συστήματα αποθήκευσης είναι απαραίτητα για τη διαχείριση του τεράστιου όγκου δομημένων και αδόμητων δεδομένων που απαιτούν οι υπηρεσίες διαδικτύου. Το αποθηκευτικό σύστημα σε ένα περιβάλλον υπολογιστικού νέφους αποτελείται συνήθως από πολλαπλά επίπεδα. Το πρώτο επίπεδο είναι υπεύθυνο για την προσωρινή αποθήκευση των δεδομένων, ενώ το τελευταίο επίπεδο παρέχει μόνιμη αποθήκευση. Επιπρόσθετα, τα συστήματα αποθήκευσης μεγάλης κλίμακας διατηρούν πολλαπλά αντίγραφα των δεδομένων σε διαφορετικούς κόμβους ώστε να πετύχουν υψηλή διαθεσιμότητα σε περίπτωση κάποιας αποτυχίας. Ωστόσο, η συνύπαρξη πολλαπλών ροών εργασιών πάνω σε μία κοινή υποδομή στο κέντρο δεδομένων (*datacenter*), εισάγει μία σειρά από σχεδιαστικά ζητήματα που αφορούν την απόδοση (*performance*), την αποδοτικότητα (*efficiency*) και την ανθεκτικότητα (*durability*) των δεδομένων στο σύστημα. Στην παρούσα διατριβή επισημαίνουμε την αναγκαιότητα για συνολική αναθεώρηση του μονοπατιού αποθήκευσης σε ένα πολυστρωματικό σύστημα μεγάλης κλίμακας. Προς αυτή την κατεύθυνση, σε διαφορετικά επίπεδα του συστήματος αποθήκευσης, εξετάζουμε την επίδραση της συνέπειας των δεδομένων στην αποδοτικότητα του συστήματος. Θέτουμε ως βασικό στόχο την εξασφάλιση της ανθεκτικότητας των δεδομένων σε συνδυασμό με υψηλή απόδοση και χαμηλές απαιτήσεις σε φυσικούς πόρους.

Στο πρώτο τμήμα της παρούσας διατριβής, μελετάμε στο κατώτερο επίπεδο της στοίβας

αποθήκευσης, και εξετάζουμε το τοπικό σύστημα αρχείων με στόχο να προσφέρουμε υψηλή αξιοπιστία, βελτιωμένη απόδοση και αποδοτική χρήση του εύρους ζώνης δίσκου. Σε ένα μεγάλο εύρος συστημάτων, οι μικρές σύγχρονες εγγραφές παίζουν κρίσιμο ρόλο στην αξιοπιστία και τη διαθεσιμότητα των συστημάτων αποθήκευσης, καθώς χρησιμοποιούνται για την ασφαλή καταγραφή των μεταβολών στην κατάσταση του συστήματος και τη μετέπειτα ανάκαμψη από πιθανές αποτυχίες. Τυπικά παραδείγματα τέτοιων συστημάτων αποτελούν τα παραδοσιακά συστήματα αρχείων, οι σχεσιακές βάσεις δεδομένων και τα συστήματα αποθήκευσης κλειδιού-τιμής. Το γεγονός ότι τα σύγχρονα συστήματα αποθήκευσης μεταφέρουν τα δεδομένα από τη μνήμη στο δίσκο σε ολόκληρα μπλοκ, έχει σαν αποτέλεσμα την άσκοπη χρήση εύρους ζώνης δίσκου που επιφέρει αυξημένη καθυστέρηση εγγραφής.

Προκειμένου να ελαττώσουμε τις απαιτήσεις σε εύρος ζώνης δίσκου, σχεδιάσαμε δύο νέες μεθόδους καταγραφής ενημερώσεων (*journaling*). Πιο συγκεκριμένα, βασιζόμαστε στην τεχνική καταγραφής ενημερώσεων στα μεταδεδομένα και τα δεδομένα, ώστε να πετύχουμε στην ασφαλή εγγραφή τους στο δίσκο με χαμηλή καθυστέρηση, αξιοποιώντας την ακολουθιακή προσπέλαση του αρχείου καταγραφής. Η πρώτη μέθοδος αποθηκεύει μόνο την πραγματική μεταβολή στα δεδομένα ως αποτέλεσμα των αιτήσεων εγγραφής του χρήστη, συγκεντρώνοντας πολλαπλές μικρές αιτήσεις σε ένα πλήρες μπλοκ. Στη συνέχεια προτείνουμε μία εναλλακτική μέθοδο καταγραφής ενημερώσεων, η οποία έχει ως στόχο να μειώσει την κίνηση στο αρχείο καταγραφής στην περίπτωση μεγάλων αιτήσεων ακολουθιακής προσπέλασης. Το προτεινόμενο σύστημα διαχωρίζει τις αιτήσεις με βάση κάποιο κατώφλι εγγραφής (*write threshold*). Σύμφωνα με το μέγεθος της εκάστοτε αίτησης, τα δεδομένα της αποθηκεύονται είτε στο αρχείο καταγραφής, είτε στην τελική τους θέση στο δίσκο. Υλοποιήσαμε τις προτεινόμενες μεθόδους στο ευρέως διαδεδομένο σύστημα αρχείων ext3 του Linux. Πραγματοποιήσαμε την αξιολόγηση του προτεινόμενου συστήματος χρησιμοποιώντας μια σειρά εκτενών πειραματικών μετρήσεων που περιλαμβάνουν συστήματα ροών δεδομένων, διακομιστές ηλεκτρονικής αλληλογραφίας, συστήματα επεξεργασίας συναλλαγών πραγματικού χρόνου και παράλληλα συστήματα αρχείων. Συγκρίναμε το σύστημά μας με τις υπάρχουσες προσεγγίσεις και αποδείξαμε ότι επιτυγχάνει χαμηλή καθυστέρηση εγγραφής και επαναφοράς, υψηλό ρυθμό εξυπηρέτησης συναλλαγών, ενώ έχει χαμηλές απαιτήσεις σε εύρος ζώνης δίσκου.

Στο δεύτερο τμήμα της παρούσας διατριβής, επικεντρωνόμαστε στο πρώτο επίπεδο ενός πολυστρωματικού συστήματος αποθήκευσης. Ειδικότερα, μελετάμε ζητήματα που

αφορούν την απόδοση, την αποδοτική αξιοποίηση των πόρων και την ανθεκτικότητα των δεδομένων σε ένα κλιμακώσιμο σύστημα αποθήκευσης. Στο κέντρο δεδομένων, η εκτέλεση πολλαπλών ροών εργασίας πάνω στην ίδια υποδομή μπορεί να οδηγήσει σε περιορισμένη κλιμακωσιμότητα εξαιτίας της αυξημένης χρήσης κρίσιμων πόρων, όπως το εύρος ζώνης δίσκου και δικτύου. Η μεγάλη επιβάρυνση του δικτύου επιβάλλει στον πελάτη να διατηρήσει για κάποιο χρονικό διάστημα τα ενημερωμένα δεδομένα στην κρυφή του μνήμη (*cache*) για λόγους απόδοσης. Η κρυφή αποθήκευση στην πλευρά του πελάτη πάνω από ανθεκτικά μέσα μπορεί να βελτιώσει την ανθεκτικότητα των δεδομένων, ενώ εφαρμόζεται ήδη σε επίπεδο μπλοκ (*block-based*) προκειμένου να μειωθεί το φορτίο του αποθηκευτικού συστήματος που βρίσκεται από πίσω. Ωστόσο, η κρυφή αποθήκευση επιπέδου μπλοκ κρίνεται ανεπαρκής λόγω της πρόσθετης επιβάρυνσης στην απόδοση του συστήματος, και της περιορισμένης δυνατότητας κοινοχρησίας δεδομένων που προσφέρει μεταξύ διαφορετικών διακομιστών. Επιπρόσθετα, σε περίπτωση αποτυχίας της λειτουργίας του πελάτη, η κρυφή αποθήκευση με περιοδική εγγραφή (*writeback caching*) μπορεί να οδηγήσει σε απώλεια των δεδομένων τα οποία προηγουμένως επιβεβαιώθηκαν ως μόνιμα αποθηκευμένα στις εκτελούμενες εφαρμογές.

Στην παρούσα διατριβή, βελτιώνουμε την ανθεκτικότητα της κοινόχρηστης αποθήκευσης στο κέντρο δεδομένων με την υποστήριξη καταγραφής ενημερώσεων στον πελάτη επιπέδου πυρήνα ενός κατανεμημένου συστήματος αρχείων μεγάλης κλίμακας. Η επιλογή της διεπαφής αρχείων (*file-based interface*) απλοποιεί το μονοπάτι προς το τελικό αποθηκευτικό μέσο προσφέροντας πολλαπλά οφέλη. Αρχικά, βελτιώνει την απόδοση των εφαρμογών και πετυχαίνει ξεκάθαρη σημασιολογία συνέπειας μεταξύ δεδομένων και μεταδεδομένων. Επιπλέον, υποστηρίζει την εγγενή κοινοχρησία αρχείων μεταξύ πελατών που τρέχουν είτε στον ίδιο, είτε σε διαφορετικούς διακομιστές. Τέλος, η διεπαφή αρχείων επιτρέπει την ευέλικτη ρύθμιση της χρονικής περιόδου κατά την οποία τα δεδομένα είναι αξιόπιστα αποθηκευμένα στην πλευρά του διακομιστή. Προκειμένου να δώσουμε τη δυνατότητα σε πολλαπλούς πελάτες να προσπελάσουν ταυτόχρονα κοινά δεδομένα, εξασφαλίζουμε ότι το σύστημα διατηρείται συνεχώς σε μία συνεπή κατάσταση. Επίσης, παρέχουμε έναν ειδικό μηχανισμό για την επαναφορά των δεδομένων που βρίσκονται αποθηκευμένα στο αρχείο καταγραφής ενημερώσεων του πελάτη κατά την ανάκαμψη του συστήματος από κάποια αποτυχία. Αξιολογούμε πειραματικά την πρωτότυπη υλοποίηση που αναπτύξαμε στο κατανεμημένο σύστημα αρχείων του Ceph στο Linux, χρησιμοποιώντας μια τοπική συστοιχία υπολογιστών,

και το μεγάλης κλίμακας περιβάλλον υπολογιστικού νέφους της Amazon. Συνολικά το προτεινόμενο σύστημα επιτυγχάνει σημαντική βελτίωση στην απόδοση για συγκεκριμένες εγγυήσεις ανθεκτικότητας, αξιοποιώντας μειωμένο εύρος ζώνης δικτύου και δίσκου στους διακομιστές αποθήκευσης.

Chapter 1

Introduction

1.1 Thesis Context

The continuous increase in volume, variety and velocity of data has led to the evolution of data storage. In the era of Big Data, large amounts of constantly generated information needs to be efficiently stored, analysed and processed. For instance, in 2010 Facebook reported the actual need to store and load in daily basis, 15PB and 60TB of data accordingly, with a total of one billion new photos per week [184, 18]. Today, Twitter monthly serves 316 million of active users, posting 500 million tweets per day [187]. According to a recent study, data is doubling every two years, and is expected to reach 44 trillion gigabytes by 2020 [61]. Additionally, the emergence of cloud computing has given rise to a new class of Internet-scale applications, including online serving, analytics and bulk processing, that manage the increasing amount of data and serve thousands or even millions of concurrent users.

The large data volume, in combination with the rapidly growing number of concurrent users, imposes the requirement for high scalability at the underlying cloud storage infras-

tructure. This observation has led system designers to move from centralized to scalable distributed solutions. The storage stack typically consists of several tiers with multiple servers per tier. Frontend machines temporarily cache data locally, while backend machines provide persistent storage. A common multi-tier storage architecture consists of a distributed database running on top of a cloud-scale filesystem with typical examples including Bigtable over GFS and HBase over HDFS [37, 74].

Furthermore, the advent of cloud computing creates an increasing tendency of migrating traditional desktop and server applications to the cloud. Server consolidation is attractive because it enables operational cost reduction, and power efficiency in the datacenter. Virtualization is the key technology that enables the consolidation of multiple virtual desktop or server machines on top of a shared physical infrastructure. In a virtualization environment, the physical resources are multiplexed among multiple isolated virtual machines. Today, the majority of cloud providers, such as Amazon through Elastic Compute Cloud (EC2) and Microsoft through Azure, provide computing, network and storage resources on demand through virtual machines [3, 32].

Nevertheless, hardware consolidation also introduces several design challenges. First, the co-location of heterogeneous workloads over a shared infrastructure often leads to contention of critical resources, such as disk and network bandwidth, resulting in reduced system scalability and performance [110, 155, 200, 69, 16]. Furthermore, typical assumptions of several large-scale distributed applications about resource homogeneity are weakened, which further complicates the efficient resource management of tasks and virtual machines [200]. Despite the physical resources, the co-located applications may also need to share data, for instance across jobs cooperating to solve a particular task [83]. Indeed, fine-grained data sharing is commonly enabled through a distributed filesystem interface [199].

The storage interface typically defines the semantic level of information between the application and the storage system. In a virtualized datacenter, storage is provided through protocols operating at the block, file or object level. The block-based interface in the form of virtual disks has prevailed in the storage management of virtualization environments. Virtual disks are stored as flat files usually over a network fileserver, or a storage array network. However, storage consolidation at the block-level introduces multiple layers of abstraction. This results in redundant translations between different

layers of the storage and the network stack in the course of a request from the guest to the remote storage backend [80, 176, 177, 9, 106]. Alternatively, through the file interface, the guest can directly access the remote fileserver without unnecessary performance overhead. Moreover, the file interface facilitates semantical awareness, which further enables several performance optimizations and strengthens consistency [120, 111, 189, 91, 1, 43]. Native support for fine-grained file sharing is also a desirable feature [145, 120, 172, 191, 23]. Overall, although the block-based interface provides virtualization flexibility and wide system compatibility, a file-based interface is attractive for its performance, controlled sharing properties, and clear consistency semantics. As a third option a guest can use an object-based interface to directly access multiple object based servers for improved scalability [189].

On another design dimension, service failures have been surveyed and analyzed extensively across different online providers [136, 84]. Operator errors in the form of misconfigurations or buggy custom-written software running in the machines or the network, are recognized as the leading causes of service failures. In order to tolerate network and machine failures, large scale systems usually replicate data across multiple nodes. However, the frontend layer is usually kept stateless for reduced communication overhead [25]. In particular, the frontend only stores locally soft state that does not survive crash failures or reboots, which hurts the durability of the system.

In general, the filesystem consistency introduces a tradeoff between performance and durability. The low latency and high throughput of directly-attached storage allows a local filesystem to periodically flush data and metadata updates to stable storage. In contrast, the potential contention over the network or the shared servers mandates that the client of a distributed filesystem preferably keeps dirty data unflushed in volatile memory arbitrarily long for improved performance and resource efficiency. In this case, the client participates in the system failure model; if the client fails, it loses recently updated blocks in the volatile memory which have not reached the server yet. Indeed, several designers of local filesystems, large-scale storage systems, and flash-based caching, decide to trade consistency for improved performance by decoupling the ordering of write requests from their durability [42, 101, 121].

At the storage backend, the crash consistency of local filesystems has been studied extensively [42, 146]. Write-ahead logging is a technique commonly used to improve

system reliability by preserving recent updates from failures; it also increases system availability by substantially reducing the subsequent recovery time. Synchronous small writes play critical role in the availability of a wide range of systems, including traditional filesystems, relational databases, and key-value stores, because they safely log recent state modifications for fast recovery from crashes [72, 66, 37, 104, 19, 169, 63, 75, 140]. Nevertheless, the storage stack enforces page-sized granularity in data transfers from memory to the storage backend, resulting in inefficient storage bandwidth utilization and reduced performance. In fact, the resource waste is exacerbated due to multiple concurrent streams that generate random I/O traffic [32, 13].

Despite the technological advances in cloud computing over the last decade, cloud storage systems face several limitations related to their efficiency, performance, durability, or file sharing properties [23]. In this thesis, we propose the fresh reconsideration of the I/O path in multi-tier virtualized environments. We show that the resource efficiency at each storage tier contributes to the overall system performance improvement. We also recognize the importance of investigating the implications of the consistency semantics to the resource efficiency and the performance, at each tier separately. Overall, we combine improved filesystem consistency with high performance and efficient resource utilization, across different layers of the storage stack.

Our initial goal is to improve the consistency and the bandwidth efficiency of the local filesystem at the storage backend. We rely on journaling of data updates in order to ensure their safe transfer to disk at low latency and high operation throughput without excessive resource overhead. We design and implement a new journaling method which merges concurrent subpage writes into page-sized blocks to the journal. We also develop an additional journaling method which only logs updates below a write threshold and transfers the rest directly to the filesystem.

Subsequently, we focus on the frontend layer aiming to improve the performance, resource efficiency, and durability of the shared storage system. Especially, we rely on storage virtualization at the file level for its clear consistency semantics, native file sharing support, and performance characteristics. We improve the durability of shared storage in the datacenter by supporting local disk-based journaling at the kernel-level client of a scalable distributed filesystem. Our approach enables frequent flushes of dirty pages to the local journal without crossing the network and hitting the disks of the backend

storage.

1.2 Thesis Contributions

According to the examined storage layer of a multi-tier environment, the contributions of this thesis can be classified into two categories.

At the storage backend layer:

- We measure bandwidth inefficiencies in journaled filesystems and examine ways to combine filesystem consistency with high performance at moderate cost.
- We design and implement two new mount modes, wasteless and selective journaling, in a widely-used local filesystem.
- We discuss the effects of alternative journaling optimizations to the consistency semantics of the filesystem in the context of different storage configurations.
- We experimentally show the performance improvement of the proposed modes, at low journal bandwidth requirements, across a wide range of realistic workloads over standalone servers, and a multi-tier networked system.

At the frontend layer:

- We propose to improve the durability of frontend memory caching with a local disk-based journal at the client of a distributed filesystem.
- We design and implement a prototype of the proposed storage layer at the client of a commonly-used object-based filesystem.
- We carefully investigate the consistency semantics of the proposed storage protocol under normal system operation, and in case of client failures, such as network disconnection and reboot.

- We experimentally demonstrate that the proposed design improves the application performance, and reduces the disk and network bandwidth utilization for specific durability guarantees over a local clustered storage backend and a large scale cloud environment.

1.3 Thesis Organization

The remainder of the present thesis is organized as follows:

In **Chapter 2**, we provide the background required to understand the challenges that arise in multi-tier storage environments with regard to the implications of crash consistency on the resource efficiency and the performance of the system.

In **Chapter 3**, we motivate our work on improving the bandwidth efficiency for consistent multistream storage at the storage backend by experimentally revealing bandwidth waste in a widely known journaling filesystem. Then, we present the design of the two new journaling modes that we propose, and discuss the consistency semantics provided by the Okeanos design.

In **Chapter 4**, we describe the implementation details of the Okeanos prototype, and investigate the implications of alternative journaling optimizations to the consistency semantics of the filesystem in the context of different storage configurations.

In **Chapter 5**, we present the experimentation environment that we used, and provide the experimental evaluation of the Okeanos prototype through microbenchmarks and application-level workloads on standalone servers, and a multi-tier networked system.

In **Chapter 6**, we motivate our work on improving the durability of shared storage by experimentally measuring the amount of dirty data that remains vulnerable at the client memory of a commonly used large-scale filesystem over time. We outline the proposed design goals and describe the architecture of Arion. We also investigate the consistency semantics of our storage protocol along with its implications to the efficiency.

In **Chapter 7**, we initially provide some necessary background information, and then present the implementation details of the Arion prototype.

In **Chapter 8**, we describe our experimentation environment, and present the experimental evaluation of Arion using microbenchmarks, application-level workloads, and

real-world applications over a local cluster and a public-cloud setup. Additionally, we examine an alternative storage device setup.

In **Chapter 9**, we compare the proposed systems with previous research related to the storage management in virtualization and cloud environments. Furthermore, we present a detailed study of the previous research on distributed filesystems, transaction processing, flash-based caching and filesystem logging, and discuss several device and application-specific reliability issues.

In **Chapter 10**, we investigate some promising directions for future work, and describe our ongoing work on related open research issues.

In **Chapter 11**, we provide an overview of the contributions of this thesis, and summarize the basic conclusions.

Chapter 2

Background

Infrastructure virtualization in the datacenter consolidates desktop and server machines over the same hardware. The physical resources are typically shared among multiple heterogeneous applications with different I/O characteristics, resulting in resource contention and limited performance scalability. The storage interface plays a critical role in the performance, efficiency, consistency and sharing properties of the co-located virtual machines. However, providing crash consistency in order to ensure the correctness of the filesystem structure after a failure, further complicates the design of a large-scale storage system, and introduces a tradeoff between performance and durability. As highly concurrent streams of data travel across the multi-tier storage stack of a virtualized cloud environment, it is important to investigate the implications of the consistency semantics to the resource efficiency and the overall system performance at each layer separately.

In this chapter we initially describe the storage properties of the alternative storage interfaces in virtualization environments. Then, we discuss the design issues and the challenges that arise in a multi-tier cloud environment. We also examine some interesting topics in the reliable storage management of highly-concurrent streams of data over a wide range of applications. Finally, we focus on the crash consistency semantics of local filesystems.

2.1 Storage in Virtualization Environments

The increasing power of modern systems in combination with flexible virtualization software, encourage service providers to consolidate multiple virtual servers on a single physical machine. Virtualization enables the multiplexing of physical resources, such as disk capacity and network bandwidth, across different guest operating systems. A thin layer, namely the *Virtual Machine Monitor*(VMM) or *hypervisor*, lies between the hardware and the hosted *virtual machines*. The VMM is responsible for scheduling and managing the allocation of physical system resources among the co-located guest machines. The individual workflows from multiple guest operating systems result in highly-concurrent streams of data to the underlying storage system. Moreover, the submitted I/O requests are increasingly latency-sensitive.

In virtualization environments the performance interference among multiple concurrently-running virtual machines leads to unpredictable performance and suboptimal resource utilization [70, 170, 110]. For example, a single badly-behaving application is able to saturate the storage system due to random I/O requests. As a result, the contention in the storage system has been extensively studied over the last years [70, 170, 110]. Another significant difficulty is the *semantic gap* introduced between the hosted machines and the VMM [41, 189]. In particular, the resource scheduling decisions of the VMM are agnostic of application-specific information available at the operating system level. Additionally, the semantic gap raises questions about data consistency because the hypervisor operates transparently to the guest filesystem [1, 43]. Overall, it remains difficult to efficiently provide crash consistency guarantees at low latency and high throughput in virtualization environments.

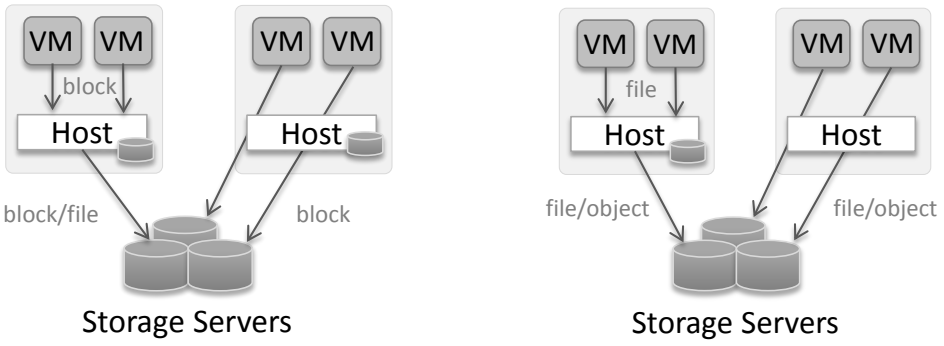


Figure 2.1: (a) Block, (b) file and object storage interfaces in virtualization environments.

Application data is typically served in the form of *virtual disks* stored as flat files over a network fileserver, or volumes over either directly-attached storage or a storage-area network. A guest accesses a virtual disk as a local device through a *block-based* interface as depicted in Figure 2.1a. Alternatively, a *file-based* interface can be used at the guest side to directly access remote file servers over the network (Figure 2.1b).

Despite the actual storage backend (SAN or NAS), a virtual disk is also formatted with a local filesystem within the guest. The resulting multi-layered storage stack may lead to performance degradation due to multiple file-to-block transformations of the I/O requests [80, 176, 177, 9]. Similarly, filesystem nesting through a block-based interface has been reported to incur significant performance overhead [106]. On the contrary, a file-based interface reduces redundant translations between different layers of the storage and the network stack as a request travels from the guest to the remote file server. Therefore, remote storage access through a file-based interface has been advocated to improve the performance of virtual machines in comparison to block-based access.

Another significant advantage of the file interface is its ability to preserve file-level semantics in order to improve the performance and strengthen consistency [120, 111, 189, 91, 1, 43]. In particular, the file interface provides valuable semantical information about the consistency dependencies of modified the data and metadata blocks. Instead, the block-based interface treats metadata as data below the guest filesystem, which can lead to inconsistencies in case of a guest failure [43, 106, 176, 101, 91, 1]. Additionally, a file-

Benefits	Interface		References
	Block	File	
sharing		✓	[145, 120, 172, 191, 23]
semantical awareness			
consistency		✓	[43, 106, 101, 189, 91, 1]
performance		✓	[176, 80, 106, 9, 91]
manageability			
disaster recovery	✓	✓	[50, 141]
migration	✓	✓	[143, 57]
thin provisioning	✓	✓	[57, 106]
searchability		✓	[145, 120]
snapshotting	✓	✓	[193, 141, 106]
versioning	✓	✓	[145, 57]
systems compatibility	✓		[176, 9]
isolation	✓	✓	[145, 103, 112]

Table 2.1: Storage properties respectively facilitated by the block-based and file-based interface of a virtual machine.

based interface natively supports controlled file-sharing among different virtual machines [94, 51, 120, 145, 172, 191, 23, 9].

On the contrary, the file interface has been criticized for reduced virtualization flexibility and limited isolation of the guest machines [80, 176, 9, 145]. In contrast to the file-based interface, the block interface also provides wide compatibility across different backend storage systems and frontend guest operating systems [176, 189, 9]. However, the block interface limits content searching within the virtual machines [145]. Finally, both interfaces provide flexible manageability through versioning, migration, thin provisioning, snapshotting, and disaster recovery [145, 57, 106, 193, 141, 50, 141]. In Table 2.1 we summarize the comparative benefits of block-based and file-based networked storage as reported in current literature. Alternatively, for improved scalability, an *object-based* interface allows the guest to directly connect to multiple object servers (Figure 2.1b) [193, 189].

2.2 Cloud Storage

The storage management in a cloud environment is challenging due to the increasingly large amount of data and the constantly growing number of concurrent users. The cloud storage solutions can be categorized as *structured* and *unstructured*. Unstructured data, such as text and multimedia files, has no predefined data model, while structured data depends on a particular well-defined schema. Unstructured data management is provided over the traditional file-based interface by large-scale filesystems, such as the Google File System (GFS) and the Hadoop Distributed File System (HDFS) [65, 171]. Instead, key-value stores are used for the storage management of structured data [37, 74]. Common multi-tier cloud storage examples include Bigtable over GFS and HBase over HDFS [37, 74]. Nevertheless, such layering has been reported to result in reduced consistency, while the write-dominated disk I/O negatively impacts the overall system performance [74].

Generally, the compute and storage resources of physical machines in the datacenter are multiplexed among multiple *heterogeneous* applications. According to recent publicly available traces from Google, the variability concerns both the executed tasks and the resource types of the running virtual machines, in case of virtualized environments [155]. For instance, server and desktop workloads have different I/O patterns from large-scale distributed applications [121]. More specifically, big data workloads (such as MapReduce) issue large sequential I/O, while POSIX applications (such as databases) usually result in small, random I/O requests [109]. The co-location of such heterogeneous workloads on top of a shared infrastructure through virtualization can lead to significant resource contention, impacting the application performance [110, 155, 200, 69, 16]. Heterogeneity is also observed in the consistency requirements of different applications [178]. For example, POSIX applications typically require stricter consistency semantics. However, POSIX semantics can be provided at the cost of limited performance, especially when multiple machines access shared storage [78]. Instead, several system designers typically prefer to trade strong consistency for improved performance [121].

The consistency and availability of large-scale storage systems in the event of machine and network failures have been extensively studied over the last decade. In general, the availability and performance of cloud services depend on the ordering, durability and membership properties of replication consistency [25]. In a multi-tier system, data is

replicated at the frontend application, an intermediate caching layer, and the backend persistent storage. Replication across multiple machines allows the system to tolerate machine crashes and network partitions. For reduced cross-layer communication, the frontend can be stateless and lose recently written data during a crash or reboot. Recognition of this risk has urged the designers of local filesystems, flash-based caches and distributed storage systems to emphasize the ordering guarantees of crash consistency at the expense of weaker durability [42, 101, 121, 46].

On another design dimension, in order to improve their performance and reduce the respective network and server load, I/O-intensive workloads in a distributed filesystem can take advantage of *writeback caching* at the client side. Writeback caching allows the application data to be acknowledged as soon as it reaches the client-side cache, while the actual transfer to the backend storage is performed at a later time. Unfortunately, current scalable filesystems can natively support only in-memory caching at the client side, resulting in limited durability [193, 50, 171]. This deficiency has been partially addressed by having the filesystem client running in the hypervisor (or another proxy node) and enforcing the guests to mount disk images as plain files through a block interface that enables block-based caching at the hypervisor [31, 191]. Nevertheless, this approach has been criticized for the increased overheads from the semantic gap that it introduces and the unnecessary multiple translations between the file and block interface [80, 106, 176, 23]. Indeed, data sharing is a useful feature of the file interface, and it is desirable either among independent web services co-located within the same cloud, or across jobs cooperating to solve a particular task [64, 83].

2.3 Reliable Multistream Storage

Synchronous small writes lie in the critical path of several systems that target fast recovery from failures with low performance loss during normal operation [72, 66, 82, 37, 104, 5, 19, 113, 169, 63, 75, 140]. Before modifying the system state, updates are recorded to a sequential file (*write-ahead log*). Periodically the entire system state (*checkpoint*) is copied to permanent storage. After a transient failure, the lost state is reconstructed by replaying recent logged updates against the latest checkpoint [190].

Write-ahead logging improves system availability by preserving state from failures and substantially reducing recovery time. It is a method widely applied in general-purpose file systems [156, 82, 163, 148], relational databases [66], key-value stores [37, 113], event processing engines [104, 29], and other mission-critical systems [132, 32, 27]. Logging is also one technique applied during the checkpointing of parallel applications to avoid discarding the processing of multiple hours or days after an application or system crash [147, 19, 139]. Logging incurs synchronous small writes, which are likely to create performance bottleneck on disk [66, 192, 132, 10, 124]. Thus, the logging bandwidth is typically over-provisioned by placing the log file on a dedicated disk separately from the devices that store the system state (e.g. relational databases [127], Azure [32]). In general, asynchronous writes also behave as synchronous if an I/O-intensive application modifies pages at the flushing rate of the underlying disk [17].

Furthermore, a distributed service is likely to maintain numerous independent log files at each server (RVM [159], Megastore [13], Azure [32]). For instance, multiple logs facilitate the balanced load redistribution after a server failure in a storage system. If the logs are concurrently accessed on the same device, random I/O is effectively generated leading to long queues and respective delays. This inefficiency remains even if the logs are stored over a distributed filesystem across multiple servers. One solution is to manage the multiple logs of each server as a single file (e.g. Bigtable over GFS [37], HBase over HDFS [27, 74]). In case of recovery, individual logs have to be separated from each other at the cost of extra software complexity and processing delay during recovery.

For the needs of high-performance computing, special file formats and interposition software layers have been developed to efficiently store the data streams generated by multiple parallel processes [60, 79, 147, 19]. In structures optimized for multi-core key-value storage, the server thread running on each core maintains its own separate log file [116]. For higher total log throughput it is recommended that different logs are stored on different magnetic or solid-state drives. However, fully replacing hard disks with flash-based solid-state drives is currently not considered a cost-effective option for several storage-intensive workloads [130, 67]. Also, while the storage density of flash memory continues to improve, important metrics such as reliability, endurance, and performance of flash memory are currently declining [68].

2.4 Local Filesystem Consistency

In a local filesystem, inconsistencies can arise due to a hardware error, memory corruption, or a system crash. Write ordering is essential to preserve filesystem consistency in the presence of system crashes. Typically write ordering is imposed by explicit cache flushes. However, a cache flush request is expensive because it forces all dirty data to disk, even if only a subset actually needs to be persisted [42, 112]. The durability of written data can be improved if one alternatively disables the on-disk cache, or uses controllers with battery-backed cache [152, 169].

According to the guarantees provided for data and metadata blocks, a recent study distinguishes the following levels of filesystem consistency [43]:

- Metadata consistency ensures that the metadata structures are consistent, with no guarantees about data blocks.
- Data consistency provides guarantees about both metadata and the corresponding data blocks.
- Version consistency additionally associates metadata with data of the matching version, resulting in the highest consistency level.

A number of techniques have been proposed to maintain filesystem consistency in the face of system crashes. *Soft updates* track and enforce metadata update dependencies so that the filesystem can safely delay writes for most file operations [163]. *Copy-on-write* techniques write filesystem blocks to any location on disk without updating the original disk blocks in-place [82, 156, 202]. Filesystem *journaling* applies writes of metadata, and sometimes data, blocks to a write-ahead log before updating their final location in the filesystem [72, 148, 186].

Soft updates ensure both data and metadata consistency, whereas copy-on-write provides all the above levels of consistency [43]. Instead, journaling supports three alternative modes with different consistency semantics. In *writeback journaling mode* only metadata blocks are logged without any constraints in the relative order at which data and metadata blocks update the filesystem. It is considered the fastest mode, but also the weakest in terms of consistency. In *ordered journaling mode* only metadata writes are logged. However, the data blocks are written to their final location right before the journal writes

of the metadata, reducing the risk of corrupting data during recovery. This order ensures that a file structure points to valid data blocks on disk. In *data journaling mode* both metadata and data blocks are logged. As a result, data journaling minimizes the risk of losing file updates and, is considered to provide the strongest consistency guarantees. In particular, it achieves data, metadata and version consistency by correctly associating metadata with data of the corresponding version.

Despite the strong consistency guarantees, data journaling incurs additional disk accesses because each block is typically transferred to disk twice; once to the journal and then later to its final location. Several studies have tried to improve the journaling performance by relaxing the ordering constraints [42, 43]. Nevertheless, the use of data journaling is commonly explicitly discouraged in several cases [148, 203, 5, 152, 43].

2.5 Summary

In this chapter, we provided the background required to understand the challenges that arise in multi-tier storage systems regarding the implications of crash consistency on the resource efficiency and performance of the system across different tiers. Overall, we underlined the importance to improve the efficiency of the resource consumption across all layers of the storage stack in a multi-tier environment in order to enhance the overall system performance. Efficiency is important because hardware consolidation can sometimes lead to scalability bottlenecks due to heavy utilization of critical resources. The semantics of the storage interface is also crucial for the performance, efficiency, and consistency properties of virtualized environments. Finally, it is essential for designers of reliable storage systems to ensure the consistency of committed updates in case of failures. However, filesystem consistency typically leads to a tradeoff between performance and durability. In the present thesis, we investigate the performance and consistency implications of critical resource consumption across different layers of the I/O path in the datacenter storage.

Chapter 3

Improving the Journaling Efficiency

In a large-scale multi-tier environment the efficiency of the local filesystem at the storage backend plays a critical role in the overall system performance. We begin our study by focusing on the storage efficiency of local filesystems. In local filesystems, journaling is a technique commonly used to ensure their fast recovery in case of system failures. However, storage stacks enforce page-sized granularity in data transfers from memory to disk resulting in inefficient bandwidth utilization in case of small write requests. In this thesis, we consider the reduction of journal bandwidth in current systems as a means to improve the performance of reliable storage at low cost. In particular, we rely on journaling of data updates in order to ensure their safe transfer to disk at low latency and high operation throughput without storage bandwidth waste. In this chapter, we present the architectural aspects of the proposed design with particular emphasis on the provided consistency semantics.

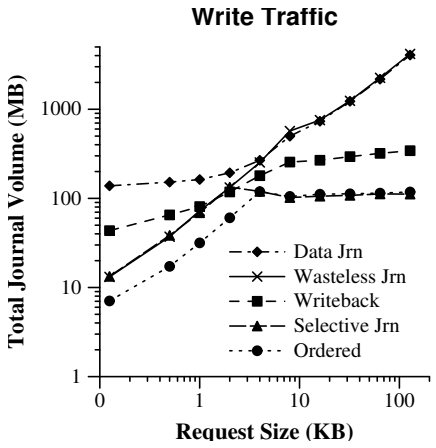


Figure 3.1: For a duration of 5min, we use 100 threads over the Linux ext3 filesystem to do periodic synchronous writes at fixed request size (each thread writes 1req/s). We measure the total write traffic to the journal across different mount modes (fully explained in Sections 3.2 and 4.1).

3.1 Motivation

Journaled filesystems copy data and/or metadata from memory to a write-ahead log (*journal*) on disk [81, 164, 186, 203, 148]. Update records are safely appended to the journal at sequential throughput, and costly filesystem modifications are postponed without penalizing the write latency perceived by the user. A page cache temporarily stores recently accessed data and metadata; it receives byte-range requests from applications and forwards them to disk in the form of page-sized requests [28, 33]. The page granularity of disk accesses is prevalent across all storage transfers, including data and metadata writes to the filesystem and the journal. In the case of asynchronous small writes, the disk efficiency is improved as multiple consecutive requests are coalesced into a page before being flushed to disk. In contrast, synchronous small writes are flushed to disk individually causing costly random I/O of data and metadata page transfers.

In Figure 3.1, we measure the amount of data written to the journal of the ext3 filesystem (described in Section 4.1). We run a synthetic workload of 100 concurrent threads for 5min. Each thread generates periodic synchronous writes of fixed request size at rate 1req/s. We include the ordered and writeback modes along with the data journaling mode. The ordered and writeback modes incur lower traffic because they only write to the journal the blocks that contain modified metadata. Instead, data journaling

writes to the journal the entire modified data and metadata blocks. In Figure 3.1, as the request size drops from 4KB to 128 bytes (by a factor of 32), the total journal traffic of data journaling only decreases from 267MB to 138MB (by about a factor of 2). Thus, data journaling incurs a relatively high amount of journal traffic at subpage requests. By tracing the block transfer activity of ext3 in the Linux kernel we found that metadata and data modifications are journaled in granularity of entire 4KB pages regardless of how many bytes are actually modified in a file page.

According to Chidambaram et al. [43], after a system crash, the data journaling mode correctly associates metadata with data of the matching version (version consistency). This type of crash consistency is stronger than only keeping the metadata structures consistent with each other (metadata consistency), or correctly associating the data blocks with the file they belong to (data consistency). However, data journaling requires each data update to be written twice, first in the journal and later in the filesystem. If it involves a large amount of written data or numerous small writes, double writes lead to excessive utilization of storage bandwidth that may hurt performance. Consequently, the use of data journaling is explicitly discouraged in several cases, while production systems activate only metadata journaling by default [148, 203, 5, 152, 43].

Today, journaling of both data and metadata is applied in a production distributed filesystem to ensure consistency of on-disk state [193]. Additionally, data journaling is desirable for increased consistency across several production environments, including the native filesystem of I/O-intensive high-performance computing systems [137], or the host filesystem holding the disk images of virtual machines running write-dominated workloads [106]. In general, write-optimized filesystems that improve random access performance are increasingly important for networked environments [109].

In this chapter, we investigate the performance and consistency implications of storage bandwidth consumption in journaled and other filesystems. In the case of data journaling, we find that the excessive disk traffic of synchronous small writes is primarily a result of the page granularity enforced by the storage stack and less a consequence of writes to both the journal and the filesystem. In fact, journaling may actually improve performance because it safely copies updates to disk at sequential throughput. The bandwidth inefficiency of small writes is not trivially overcome by reducing the granularity of disk writes to a single sector because smaller writes would cause higher I/O overhead in the system. Instead,

we propose to accumulate the modifications from multiple subpage updates from different threads into a single page, and only pay once the disk I/O cost of the page write. This approach cannot be directly applied to writes that modify the filesystem in-place because each write corresponds to a different block on disk. However, it is applicable to the updates appended into the journal.

We set as objective to achieve filesystem crash consistency at high I/O performance with efficient bandwidth utilization. We introduce, design, and fully implement two new mount modes, *wasteless journaling* and *selective journaling*. We are mainly concerned about highly concurrent multi-threaded workloads that synchronously apply small writes over the same storage devices [132, 37, 5, 104, 19, 27, 32, 169]. Unnecessary writes of unmodified data and writes of high positioning overhead occupy valuable disk access time. Thus they waste disk bandwidth, which should preferably be spent on useful data transfers. To achieve our objective, we transform multiple random small writes into a single block append to the journal. With microbenchmarks and application-level workloads, we show that our two modes can considerably reduce the journal (and filesystem) traffic. More importantly, they improve operation throughput and substantially reduce the response time in comparison to alternative mount options and filesystems.

To the best of our knowledge, the present work is the first to comprehensively investigate the general benefits of subpage data journaling using a prototype implementation in a fully operational filesystem. We summarize our contributions as follows:

1. Measure bandwidth inefficiencies in journaled filesystems and examine ways to combine filesystem consistency with high performance at moderate cost.

2. Design and fully implement wasteless and selective journaling as optional mount modes in a widely-used filesystem.

3. Discuss the implications of alternative journaling optimizations to the consistency semantics of the filesystem in the context of different storage configurations.

4. Apply micro-benchmarks, storage workloads and database logging traces over a journal spindle to demonstrate performance improvements up to several orders of magnitude across different metrics.

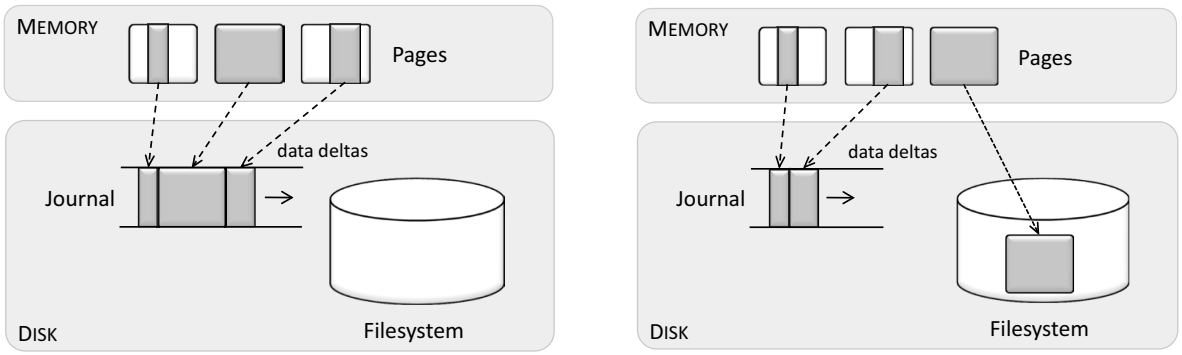
5. With a parallel filesystem, we show that wasteless journaling doubles the throughput of parallel checkpointing over small writes, while it reduces the total traffic to disk.

3.2 System Design

In the present section, we describe the basic assumptions and objectives of our journaling architecture. In a general-purpose filesystem, we aim to safely store recent state updates on disk and ensure their consistent recovery in case of failure. We also strive to serve synchronous small writes and subsequent reads fast, with low bandwidth requirements. The consistency of *metadata* updates has already been studied previously [72, 163]. Additionally, subpage journaling of metadata updates is made widely available today through popular commercial filesystems, such as the IBM JFS and MS NTFS [148]. On the contrary, data journaling is only supported in fewer filesystems (e.g. ext3/4, ReiserFS) and its use is generally avoided because it is considered harmful for performance [43]. Moreover, the subpage journaling of *data* updates is not supported in current filesystems.

3.2.1 Wasteless Journaling

Historically, journaling was only applied to the metadata of a filesystem with specific goal to ensure fast structural recovery after a system failure [72, 186]. Today, support for data journaling is provided in few filesystems to preserve from a system crash the latest data updates and keep them accessible [43]. As a side effect of the journal sequential access, data journaling can improve the throughput of random I/O operations. However, this benefit is realized at the cost of excessive bandwidth consumption due to the page granularity of the storage traffic [148, 10]. In order to overcome this limitation, we designed and implemented a new mount mode that we call *wasteless journaling*. In synchronous writes, we transform partially modified data blocks into descriptor records, which we subsequently accumulate into special journal blocks (Figure 3.2.a). For data blocks that have been fully modified by write operations, we synchronously copy the entire blocks from memory to the journal. After timeout expiration or due to shortage of journal space, we copy the partially or fully modified data blocks from memory to their final location in the filesystem. Subsequently, we clean the respective records from the journal device.



(a) (b)

Figure 3.2: (a) In wasteless journaling, we journal data updates at subpage granularity. (b) In selective journaling, while we treat small requests approximately as in wasteless mode, we transfer large requests directly to the final location without prior journaling of the data.

3.2.2 Selective Journaling

Data journaling adds extra I/O cost because it writes data to both the journal and the filesystem. In the particular case of sequential writes, the benefit from sequential appends to the journal device is not that significant, because the data writes to the filesystem are most likely sequential. In this case, the journal device may actually become a bottleneck and harm performance. With goal to reduce the journal I/O activity of sequential writes, we evolved wasteless journaling into an alternative mount mode that we call *selective journaling*. In this mode, the system automatically differentiates the write requests based on a fixed size threshold that we call *write threshold*. Depending on whether the write size is below the write threshold or not, we respectively transfer the synchronous writes to either the journal or the final disk location directly (Figure 3.2b). The rationale of this approach is to apply data journaling only when multiple small writes can be coalesced into a single journal block, or different data blocks are fully modified and scattered across multiple locations in the filesystem.

3.2.3 Crash Consistency

Since the synchronous write from a single thread must be transferred to disk immediately, it only makes sense to accumulate into a journal block the writes that originate from different concurrent threads. Therefore, we expect wasteless and selective journaling to be mostly beneficial in environments that consist of multiple writing streams with frequent small writes. In the case of wasteless journaling, we only consider a write operation effectively completed after we log both data and metadata into the journal device. Synchronous writes from the same thread are added to the journal sequentially. In case of failure, a prefix of the operation sequence is recovered through the replay of the data modifications that have been successfully logged into the journal. Thus, the structure of the filesystem remains consistent across system failures and the filesystem metadata refers to the latest data that has been safely stored on disk (version consistency [43]).

Selective journaling allows a series of synchronous writes to have a subset of the modified data added to the journal and the rest of the modified data directly transferred to the final location in the filesystem. During a recovery from crash, a write operation is fully aborted if the corresponding journal appends were interrupted halfway. However, if the write is large enough to be directly transferred to the final location, it may be only partially completed at the instance of the failure. Consequently, selective journaling provides the consistency of mount modes that journal the metadata only after the respective data is saved to disk (data consistency [43]). Such modes update the data in place and add metadata modifications to the journal, while selective journaling applies large data updates in place but adds to the journal both metadata modifications and small data updates. If the write traffic is dominated by request sizes below the write threshold, the consistency of selective journaling approaches the version consistency of wasteless journaling.

Arguably, the accumulation of multiple small updates into a single journal block leaves open the possibility of losing multiple updates if the block does not safely reach the journal device. However, the wasteless and selective journaling do not defer in any way the operation of buffer flushing regardless of whether it is periodically invoked by the filesystem or explicitly requested through synchronous writes. Instead, the two modes merely flush the buffer updates faster because they reduce the amount of I/O involved. This

efficiency allows applications to achieve decreased write latency and shorter vulnerability window during which requested updates remain outstanding. We provide additional explanations about the system consistency, when we describe the atomicity guarantees of our implementation in Section 4.1.4, while we experimentally demonstrate the reduced flushing latency of our modes in chapter 5.

3.2.4 Update Sequences

In selective journaling, we call *update sequence* a series of multiple incoming updates applied to the buffer of a single data block. The update sequence terminates when the modified data block along with the respective metadata blocks are safely transferred from memory to the filesystem. In our definition, the updates are not necessarily back-to-back, but there should be no in-between transfer of the respective data and metadata blocks to the final disk location. For presentation simplicity, but without loss of generality, we assume that the write threshold and the block size are both set equal to the size of a system page.

If the first update in such a sequence has subpage size, we mark the corresponding buffer as *journaled*. Then, we log to the journal the entire update sequence of the block as the individual updates are flushed to disk. It would be a straightforward approach to turn off the journaling of the block as soon as the subpage write switched into a full overwrite along an update sequence. As a result, the initial updates of the block would be journaled and the rest would be directly flushed to the filesystem. During journal replay at a subsequent recovery from a failure, the subpage writes would erroneously corrupt the block whose latest update fully overwrote it. We deliberately avoid this situation by journaling the block throughout the update sequence at the cost of paying data journaling I/O for the entire update sequence.

Instead, if the first update is page-sized, we skip journaling for the entire update sequence of the block. This implies that flushing any subsequent subpage data write to disk causes the entire data block to be flushed to the filesystem and the respective metadata blocks written to the journal. If we trivially did not do that, then only the subpage writes could be recorded in the journal. Our approach in this case sacrifices the sequential I/O of journaling in order to avoid block corruption due to only replaying the

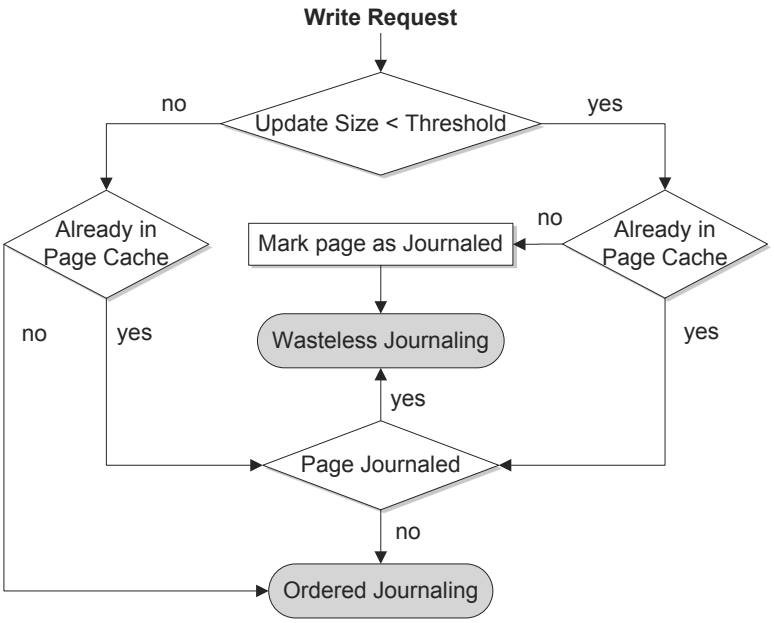


Figure 3.3: Alternative execution paths of a write request in the selective journaling mode.

subpage writes of the update sequence after a failure.

We prefer to preserve clean recovery semantics in selective journaling at the cost of lower performance gain. In our experience, the two above transitions in write size along an update sequence are not common in practice. Also, an update sequence has limited lifetime due to the periodic flushing of dirty data by the system. According to our experiments, selective journaling maintains significant performance gains across different representative workloads that we examined. In Figure 3.3, we use a flowchart to summarize the possible execution paths of a write request through selective journaling.

3.3 Summary

Motivated by the measured inefficiencies in journaled filesystems, we examined ways to combine filesystem consistency with high performance at moderate cost. In a journaled filesystem, we introduced the design of wasteless and selective journaling as alternative mount modes. Wasteless journaling coalesces synchronous concurrent small writes of data

into full page-sized journal blocks. Instead, selective journaling automatically activates wasteless journaling on data writes with size below a fixed threshold. Finally, we included a detailed discussion about the consistency semantics provided by the proposed journaling modes.

Chapter 4

The Okeanos Prototype

In this chapter we provide the implementation details of the proposed journaling modes. Additionally, we discuss the implications of alternative journaling optimizations to the consistency semantics of the filesystem in the context of different storage configurations. We implement wasteless and selective journaling in the Okeanos prototype system over the Linux ext3 filesystem.

4.1 Prototype Implementation

At a high level, the original ext3 filesystem implements journaling of updates in two steps. First, it copies the modified blocks into the journal with a a commit block at the end. Then, it updates the modified blocks in-place at the filesystem and discards the journal blocks.

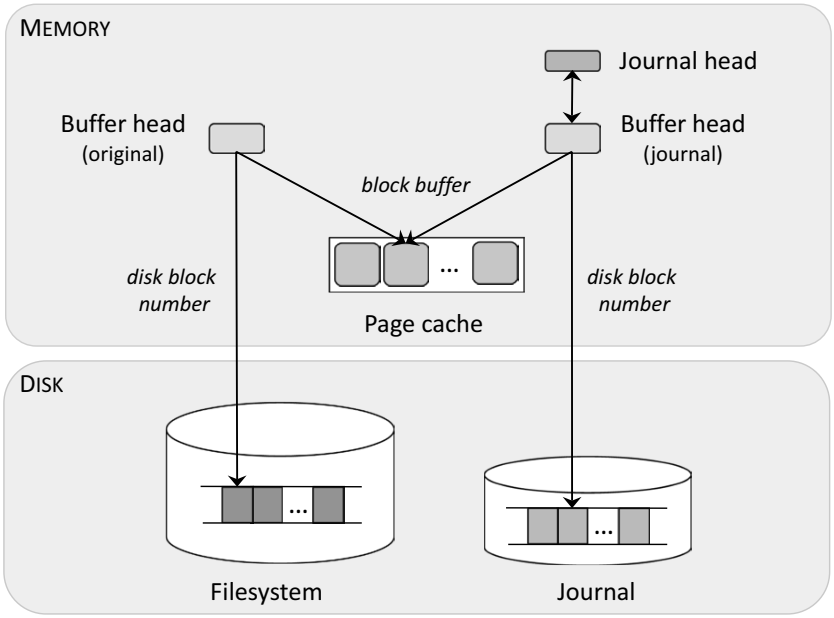
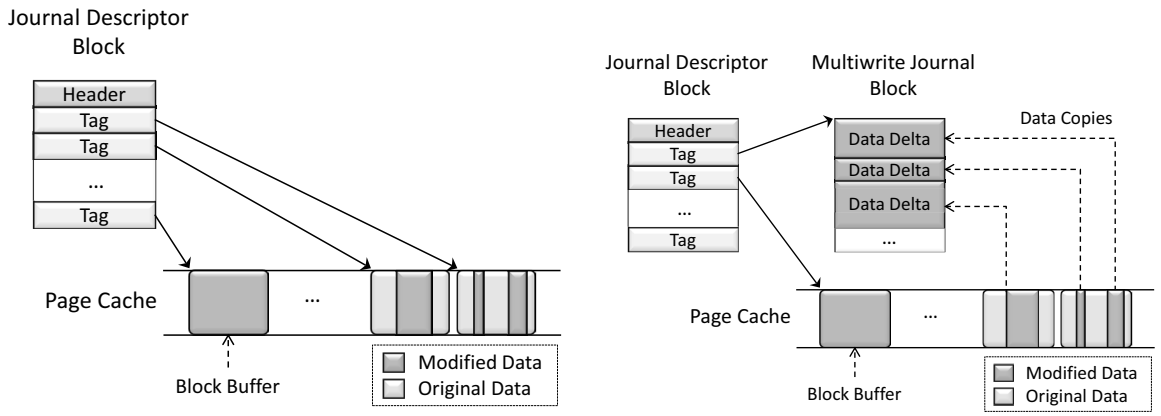


Figure 4.1: For each journaled block (i) A dedicated buffer head in memory specifies the respective disk block in the journal device and, (ii) A journal head in memory links the block with the journal transaction to which it belongs.

4.1.1 Buffers

The Linux kernel uses the *page cache* to keep the data and metadata of recently accessed disk files in memory [28]. For every cached disk block, a *block buffer* in memory stores the respective data, while a *buffer head* stores the related bookkeeping information (Figure 4.1). The page cache manages disk blocks in page-sized groups called *buffer pages*. Since the block and page typically have the same size, we use these two terms interchangeably from now on. A number of *pdflush* kernel threads periodically flush dirty pages to their final disk location. The threads systematically scan the page cache every *writeback period*; a dirty page is due for flushing after an *expiration period* has passed since it was last modified. Additionally, applications can synchronously flush the data and metadata blocks of an open file, for instance, through the *fsync* call or after opening the file with the `O_SYNC` option enabled. The *journaling block device* is a special kernel layer used by ext3 to implement the journal as a hidden file in the filesystem, or a separate disk partition. In the journal, each *log record* corresponds to an update of one disk block in the filesystem. The log record contains the entire modified block instead of the byte range



(a) (b)

Figure 4.2: (a). In the original design of data journaling, the system copies to the journal the entire blocks modified by write operations. (b) In wasteless journaling, we use multiwrite journal blocks to accumulate the data modifications from multiple writes.

actually overwritten. This wastes disk bandwidth and space but makes straightforward the restoration of modified blocks after a crash. The degree of waste depends on the fraction of the block that is left unmodified by the write operation.

At the minimum, the system only needs to log the modified part of each buffer and merge it into the original block to recover the latest block version. Thus, we introduce a new type of journal block that we call *multiwrite block* (Figure 4.2b). We only use multiwrite blocks to accumulate the updates from data writes that partially modify block buffers. If a buffer contains metadata or is fully modified by a write operation, we can send it directly to the journal without creation of an extra copy in the page cache. We call such a journal block a *regular block*. When a write request of arbitrary size enters the kernel, the request is broken into variable-sized updates of individual block buffers. In wasteless journaling, for buffer updates smaller than the block size, we copy the corresponding data modification into a multiwrite block. Otherwise, we link the update to the entire modified block in the page cache. In selective journaling, we set the *write threshold* equal to the page size of 4KB. If a buffer update is smaller than the write threshold, we mark the corresponding block as *journaled* by setting a special flag that we added in the page descriptor of the buffer. Then, we copy the modification to the multiwrite block. If the update modifies the entire block, we prepare the corresponding modified buffer for transfer

to the filesystem without prior journaling. We clear the journaled flag after we complete the block transfer to the filesystem.

In a straightforward way, our current prototype can also support arbitrary write thresholds below the page size. In contrast, support for write thresholds above the page size requires additional implementation intervention at the system path of write requests as described recently in a more general context [118]. The additional modification is necessary in order to keep track of the write size across the buffers in the page cache and treat them differently based on the write threshold.

4.1.2 Transactions

A system call may consist of multiple low-level operations that atomically manipulate disk data structures of the filesystem. For improved efficiency, the system assigns to one *transaction* the records of multiple calls. Before the records of a transaction are transferred to the journal, the kernel allocates a *journal descriptor block* with a list of *tags*. A tag maps a buffer to the respective block in the filesystem (Figure 4.2a). When a journal-descriptor block fills up with tags, the kernel moves it to the journal together with the associated block buffers. For each block buffer that will be written to the journal, the kernel allocates an extra buffer head specifically for the needs of journaling I/O. Additionally, it creates a *journal head* structure to associate the block buffer with the respective transaction (Figure 4.1). After all the log records of a transaction have been safely transferred to the journal, the system appends to the journal a final commit block.

For writes that only modify part of a block, we expanded the journal head with two extra fields, the offset and the length of the partially modified block pointed to by the buffer head. When we start a new transaction, we allocate a buffer for the journal descriptor block. The journal descriptor block contains a list of fixed-length tags, where each tag corresponds to one block update (Figure 4.2.b). Originally, each tag contained the filesystem location of the modified block and one flag for the journal-specific properties of the block. In our design, we introduce three new fields in each tag: (i) A flag to indicate the use of a multiwrite block, (ii) The length of the write in the multiwrite block, and (iii) The starting offset of the modification in the filesystem data block. These fields are required during recovery to allow the extraction of the update from the multiwrite block

and the overwrite of the respective filesystem block at the right offset.

4.1.3 Recovery

We consider a transaction *committed* if it has flushed all its records to the journal and has been marked as finished. A transaction is automatically committed by the *kjournald* kernel thread after a fixed amount of time has elapsed since the transaction started. Subsequently, we regard the transaction as *checkpointed* if all the blocks of a committed transaction have been moved to their final location in the filesystem and the corresponding log records have been removed from the journal. If the journal contains log records after a crash, the system assumes that the unmount was unsuccessful and initiates a recovery procedure in three phases. In the *scan* phase, it looks for the last record in the journal that corresponds to a committed transaction. During the *revoke* phase, the kernel marks as revoked those blocks that have been obsoleted (overwritten or deleted) by later operations. In the *replay* phase, the system writes to the filesystem the remaining (unrevoked) blocks that occur in committed transactions.

During the recovery process, we retrieve the modified blocks from the journal. In the case of multiwrite blocks, we apply the updates to blocks that we read from the corresponding filesystem locations. Since the data of consecutive writes are placed next to each other in the multiwrite block, we can deduce their corresponding starting offsets from the length field in the tags. As soon as the length field of a tag exceeds the end of the current multiwrite block, we read the next block from the journal and treat it as another multiwrite block from the same transaction. We read into memory and update the appropriate block as specified by the filesystem location and the starting offset in the tag. However, if the multiwrite flag is not set, then we read the next block of the journal and treat it as a regular block. We write every regular block directly to the filesystem without need to read first its older version from the disk.

4.1.4 Atomicity

Disk drives can guarantee the atomic update of a 512-byte sector through an attached checksum calculated over the sector data [160]. For a page that consists of multiple sectors, incomplete page updates can be detected (*torn page detection*) through additional bits

calculated over the entire page [162, 43]. Accordingly, we assume that the disk supports atomic page updates. One misbehaviour not covered by the page atomicity is the case that only a subset of the pages in a transaction actually reaches the filesystem. This is possible because the disk internally uses a write cache to temporarily store incoming data. The on-disk write cache is typically set to operate in write-back mode, which may reorder writes for better performance. Then, the disk is possible to acknowledge a synchronous page write before the data is safely stored.

The integrity of a journaled transaction can be verified with a checksum calculated over the contents of the transaction [149]. However, a journaled filesystem may silently end up in inconsistent state if the system crashes after a transaction partially updates the filesystem but before the transaction is safely stored in the journal. Such inconsistency can be avoided if the filesystem explicitly controls the on-disk ordering of journal commits. For that purpose, the Linux ext3 provides the `barrier` mount option while the SCSI specification offers the `SYNCHRONIZE CACHE` command [160]. Similarly, SATA provides the `FLUSH CACHE` command [157]. If the device does not support write barriers, a flush workload can be used to flush the on-disk write cache instead [152]. Alternatively, we can disable the write cache and have the disk only acknowledge a write after that really reaches the medium [169].

Assuming page atomicity on disk, wasteless journaling provides the consistency of data journaling. If additionally the disk barrier is used or the write cache is disabled, both wasteless and data journaling guarantee the idempotence of write operations. If a transaction replay is interrupted halfway through, from page atomicity it follows that each affected page in the filesystem will carry either the new value or the old value. A safely committed transaction can be repeatedly applied to the filesystem until it completes successfully. At this point, all the affected pages in the filesystem will have the new value.

Selective journaling marks as journaled the buffer of an update sequence based on the size of the first update to the respective data page. If a data page is prepared for direct transfer to the filesystem, there is no journal head to associate this data page with a transaction. It is possible that the system crashes right after a dirty data page is directly transferred to the filesystem. The respective metadata updates will not make it to the journal if we use write barriers or disable the on-disk write cache. After the crash, the above data update can become visible to the user if the update overwrote an existing file

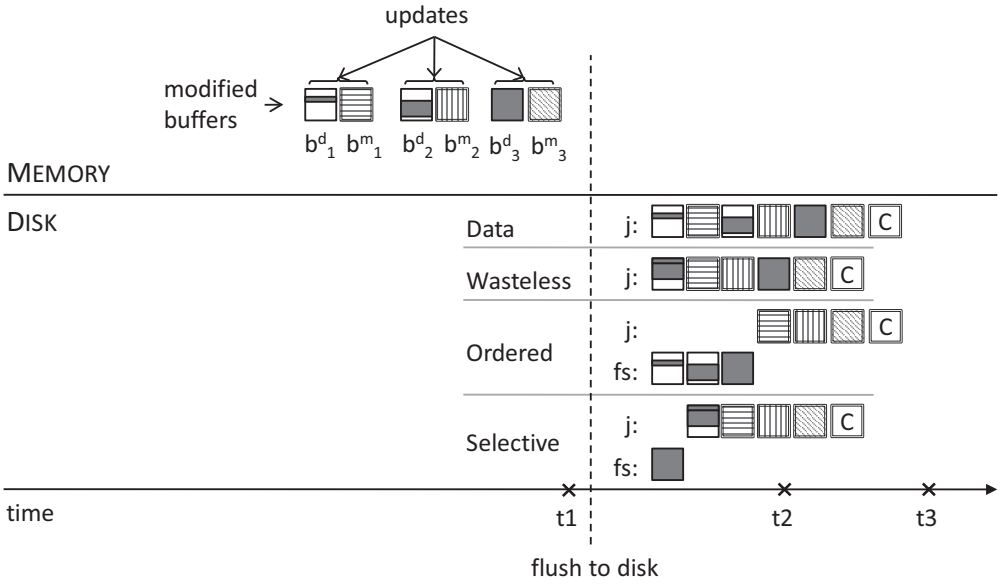


Figure 4.3: We consider three different pairs of data and metadata blocks whose respective buffers are updated in memory. From left to right, we show a possible timing of block transfers to the journal (j) and the filesystem (fs) across four different filesystem modes. The superscripts d and m of the blocks refer to data and metadata, respectively, while t_1, t_2 and t_3 refer to three time instances of system crash that we examine. The square containing c refers to the commit block.

page. Essentially, the consistency of selective journaling degenerates to that of ordered mode if the update sequence is not journaled. With journaled update sequence, the respective filesystem page is only modified if it belongs to a safely committed transaction. If all the update sequences of a transaction are journaled, the consistency of selective journaling is that of wasteless journaling. As part of our experimentation, we confirm the comparative benefits of our journaling modes across different settings of the on-disk cache and the write barrier (Section 5.2.6).

Example In Figure 4.3, we examine the potential effect of the updates applied to three different pairs of data and metadata blocks whose buffers are already located in system memory. Assuming that time increases from left to right, we refer to the data and metadata block of updates 1, 2, and 3, respectively, with b^d_1 and b^m_1, b^d_2 and b^m_2, b^d_3 and b^m_3. The squares that contain the character c symbolize the commit block of the transaction. We assume that the updates applied to b^d_1 and b^d_2 are partial, while b^d_3 is fully overwritten.

With the journal and filesystem on disk, the example indicates that wasteless and selective journaling are likely to require less I/O time to safely flush the updates from memory to disk in comparison to the ordered and data journaling modes.

If the system crashes at instance t_1, then all the updates applied in memory are lost. At the other extreme, if the system crashes at instance t_3, then all the updates can be safely recovered from disk. Although both data and wasteless journaling record all the block updates to the journal, wasteless journaling transfers one less block. The ordered mode transfers all the data blocks to the filesystem, before it appends the three metadata blocks to the journal. Selective journaling only transfers block b_3^d to the filesystem, but it copies to the journal the updates of b_1^d and b_2^d through a multiwrite block.

In this example, if the system crashes at instance t_2, then selective journaling has already modified block b_3^d in the filesystem, while the ordered mode has modified b_1^d, b_2^d, b_3^d in the filesystem. As a result both the ordered and selective journaling modes leave the filesystem in an inconsistent state after the crash. Additionally, given that the commit block has not been safely stored on disk before the crash, all four modes fail to recover the three updates. The example indicates that the multiwrite block helps reduce the I/O traffic to the journal, while any in-place updates directly applied to the filesystem may lead to inconsistencies during a crash.

4.2 Summary

To summarize, we provided a detailed description of the Okeanos prototype implementation. In particular, we implemented a method that we call wasteless journaling to merge concurrent subpage writes to the journal into page-sized blocks. Additionally, we developed the selective journaling method that only logs updates below a write threshold and transfers the rest directly to the filesystem. In this chapter we also examined the implications of alternative journaling optimizations to the consistency semantics of the filesystem.

Chapter 5

Performance Evaluation of the Okeanos System

In this chapter, we provide an extensive experimental evaluation of the Okeanos prototype. We compare the proposed journaling modes against existing methods using microbenchmarks and application-level workloads on standalone servers, and a multi-tier networked system. In our experiments we examine both synchronous and asynchronous write-intensive workloads.

5.1 Experimentation Environment

We implemented wasteless and selective journaling in the Linux kernel version 2.6.18. Newer Linux releases still lack the functionality that we propose (e.g. ext4 [102]). In order to add the proposed functions into ext3, we modified 684 lines of code across 19 files of the original Linux kernel. Members of our team used the modified system as their

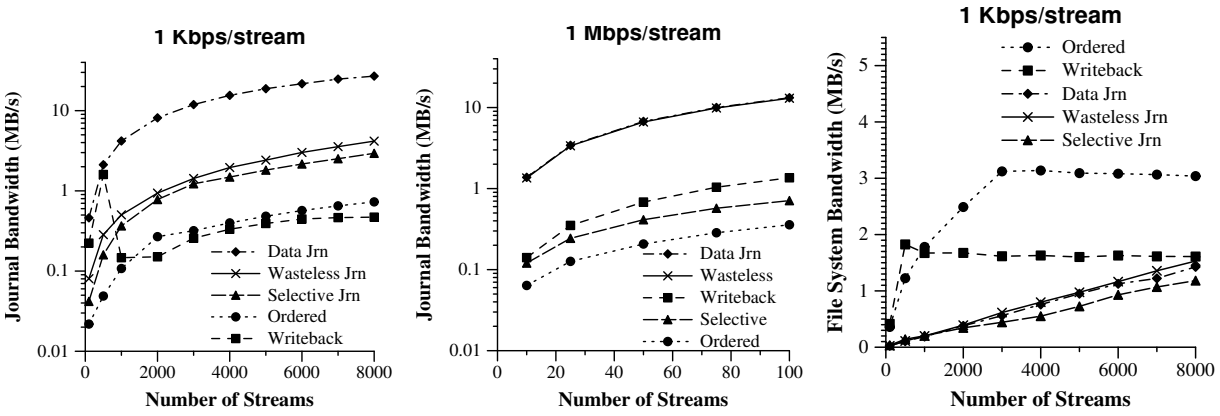
working environment for several months. We evaluated our prototype over a sixteen-node cluster using x86-based servers running the Debian Linux distribution and connected through gigabit Ethernet

In most experiments we use nodes with one quad-core 2.66GHz processor, 3GB RAM, and two SAS 15KRPM disks (Seagate Cheetah ST3300655SS [38]). Each disk has 300GB storage capacity, multi-segmented 16MB cache, 3.4/3.9ms average read/write seek time and 122-204MB/s sustained transfer rate. We have the journal and the data partition on two separate disks, unless we mention otherwise. Our conclusions were similar in several experiments that we did (not shown) with two SATA 7.2KRPM disks of 250GB capacity and 16MB cache. We keep the page and block sizes equal to 4KB, while we leave the journal size at the default value 128MB. In our measurements, we assume synchronous write operations, unless we specify differently. We keep the default parameters of periodic page flushing: writeback period equal to 5s and expiration period 30s. Between successive repetitions, we flushed the page cache by unmounting the journal device and writing the value 3 to the /proc/sys/vm/drop_caches. On otherwise idle machines, with up to fifteen experiment repetitions, we ensure that our results have half-length of 90% confidence interval within 10% of the reported average.

5.2 Performance Evaluation

We study the performance of microbenchmarks, application-level workloads and traces from database logs directly running on the modified filesystem. We also evaluate a stable Linux port of the Log-structured File System, where the entire filesystem is structured as a log [156]. Additionally, over a multi-tier configuration based on the PVFS2 distributed filesystem, we examine the impact of the server filesystem to the parallel workload running across multiple clients. Finally, we measure the recovery time after a crash.

The default disk settings typically increase performance by allowing a synchronous write to return when the data reaches the on-disk cache rather than the storage surface. The durability of written data can be improved if one alternatively disables the on-disk cache, applies flush workloads to the cache, or uses controllers with battery-backed cache [132, 152, 169]. In most of our experiments we kept enabled the on-disk



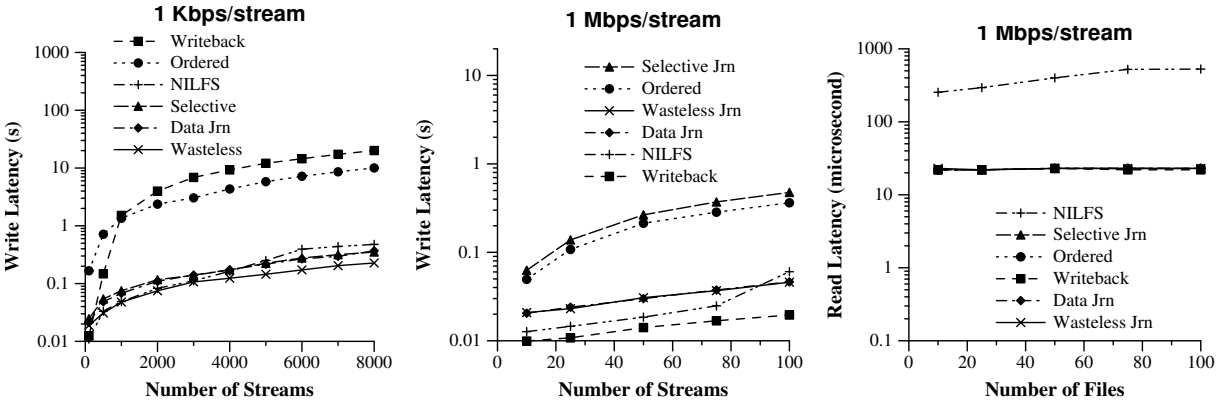
(a) (b) (c)

Figure 5.1: (a) At 1Kbps, the journal bandwidth (lower is better) of both selective and wasteless journaling approaches that of ordered and writeback modes, unlike data journaling which is several factors higher. (b) At 1Mbps, wasteless and data journaling have the same journal bandwidth, while selective journaling lies between writeback and ordered. (c) In comparison to ordered and writeback at 1Kbps, the other three modes incur lower filesystem bandwidth (lower is better), because they batch multiple writes into fewer page flushes.

caches, but in Section 5.2.6 we report the sensitivity of our results to alternative cache configurations.

5.2.1 Microbenchmarks

For a time period of 5min, we run a number of threads on the local filesystem. Each thread appends data to a separate file by calling one synchronous write per second. The generated aggregate traffic effectively consists of random I/O operations. As metric of inefficiency, we use the average consumed bandwidth (the lower the better) on the journal device across the different mount modes of ext3. With 1Kbps streams in Figure 5.1a, we observe that as the number of streams increases from one hundred to several thousand, the journal bandwidth of data journaling reaches 27MB/s. On the contrary, selective and wasteless journaling limit the journal traffic up to 2.9MB/s and 4.2MB/s, respectively. The higher storage bandwidth of data journaling is expected because it writes to the journal the entire modified data blocks instead of just the subpage modifications.



(a) (b) (c)

Figure 5.2: (a) With low rates, the write latency (lower is better) of ordered and writeback mode appears orders of magnitude longer than the other modes. (b) At higher rates, the selective and ordered modes experience much higher latency. (c) As we read sequentially multiple files that we previously wrote concurrently, read requests of 4KB size with NILFS complete in order of magnitude longer time than the different modes of ext3.

At stream rate 1Mbps, wasteless and data journaling are comparable in terms of journal bandwidth (Figure 5.1b). Instead, the selective and ordered modes transfer data updates directly to the filesystem, which reduces their journal traffic by an order of magnitude or more with respect to wasteless and data journaling. We additionally examined (not shown) streams of 10Kbps and 100Kbps, and mixed workloads with multiple stream rates at different ratios. Without surprise, the journal bandwidth of wasteless journaling varied between the values reported in Figures 5.1a,b according to the fraction of requests that correspond to each stream rate.

In Figure 5.1c, we measure the consumed bandwidth of the filesystem device for 1Kbps streams. The ordered mode synchronously transfers the data updates directly to the filesystem with costly random I/Os before moving the corresponding metadata to the journal. Instead, wasteless, selective and data journaling synchronously transfer the updates to the journal and only periodically flush the dirty pages to the filesystem. Thus, multiple writes to the same data block are automatically coalesced into fewer page flushes leading to lower traffic at the filesystem. Respectively, we also measured the processor utilization (not shown) and found it relatively higher for wasteless, selective and data journaling. Nevertheless, processor utilization in these experiments always remained low,

up to 5%.

Ultimately, the journaling of data is expected to reduce the latency of synchronous writes. As they serve multiple streams of 1Kbps, in Figure 5.2a, the ordered and writeback modes incur orders of magnitude higher latency with respect to the other modes. Multiple concurrent synchronous requests in ordered mode result in random accesses to the filesystem device. Thus, data journaling completes a write operation in tens of milliseconds, but the ordered mode takes several seconds instead. Selective journaling follows wasteless at low rates, and approaches the ordered mode at high rates (Figures 5.2a,b).

In Figure 5.2, we also consider a stable Linux port of the Log-structured File System, where all data and metadata updates are written sequentially as a continuous stream (NILFS) [197]. We find that the write latency of NILFS is comparable to that of wasteless and data journaling at both 1Kbps and 1Mbps streams. Overall, the sequential throughput of the journal improves significantly the ability of the system to store fast the incoming data. In Figure 5.2c, we use a thread to read sequentially one after the other different numbers of files that we previously created concurrently at 1Mbps each, using NILFS or ext3. In this experiment, we measure the average time to read one 4KB block. We observe that NILFS is an order of magnitude slower with respect to ext3. We attribute this behaviour to the fact that NILFS interleaves the writes from different files on disk, which may lead to poor storage locality during sequential reads. Our results with 1Kbps streams were similar; NILFS along with the ordered and writeback modes incur higher read latencies than the other three modes.

In order to examine the generality of our conclusions, we also considered streams with asynchronous writes. In I/O-intensive workloads, we anticipate that recent updates are flushed to the filesystem as a result of memory pressure, before the page cleaning daemon is periodically activated over the cache. In Figure 5.3, with several low-rate streams, we notice that the ordered mode leads to write latency that is considerably longer and highly variable in comparison to selective journaling. Essentially, selective and wasteless journaling move recent updates to the journal device at sequential throughput, which reduces the latency of ordered and data journaling up to several orders of magnitude. Correspondingly, we confirm previous reports that asynchronous workloads may behave as synchronous under conditions of high update rate [17].

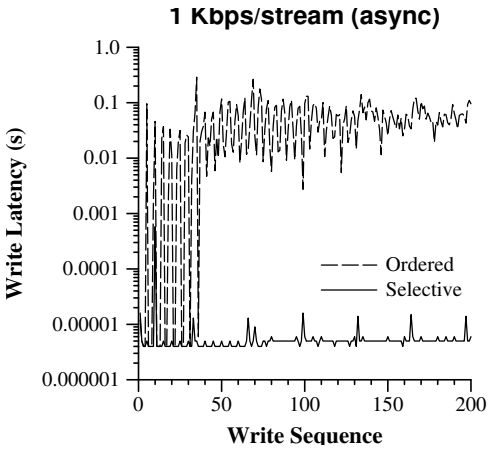
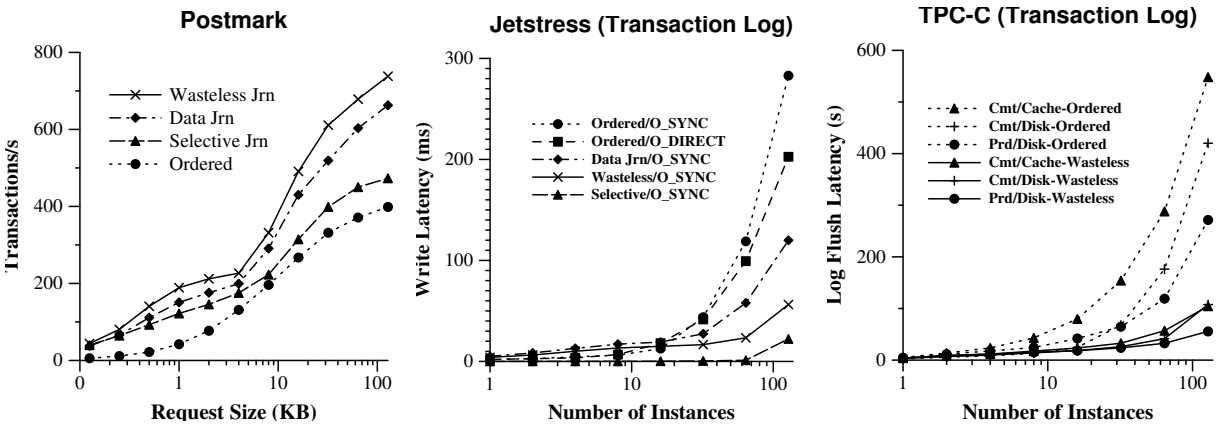


Figure 5.3: We depict the average latency of 1000 streams along a sequence of 200 disk writes. Each stream *asynchronously* writes once per second 125 bytes (1Kbps). In comparison to selective journaling, the write latency of ordered mode tends to be highly variable and orders of magnitude longer.

5.2.2 Postmark and Filebench

We use the Postmark benchmark to examine the performance of small writes as seen in electronic mail, netnews and web-based commerce [97]. We apply version 1.5 with the option of synchronous writes added by FSL of Stony Brook Univ. The experiment duration varies depending on the efficiency of the requested operations. In order to keep the runtime reasonable, we assume an initial set of 500 files and use 100 threads to apply a total workload of 10,000 mixed transactions with file read, append, create and delete operations. We set equal to 5 the ratio of read/append operations and equal to 9 the ratio of create/delete. We draw the file sizes from the default range between 500 bytes and 97.66KB, while I/O request sizes lie in the range between 128 bytes and 128KB. In Figure 5.4a, we observe that the transaction rate (higher is better) of wasteless journaling gets as high as 738 transactions/s. Wasteless journaling combines the sequential throughput of journaling with the reduced amount of written data to the journal and the filesystem during small updates. Across different request sizes between 128 bytes and 128KB, wasteless journaling consistently remains faster than the other modes, including data journaling (max rate 663 transactions/s). It is notable that wasteless journaling improves by 85% the performance of ordered mode (max rate 399 transactions/s). Instead, selective journaling with max rate 473 transactions/s lies between the data journaling and



(a) (b) (c)

Figure 5.4: (a) With the Postmark benchmark, wasteless journaling consistently outperforms the other modes in terms of operation transaction rate (higher is better). (b) We consider up to 128 concurrent Jetstress instances. In comparison to the other modes, selective journaling maintains the latency of log writes lower up to several orders of magnitude. (c) We examine the three flushing methods of MySQL/InnoDB. With respect to the ordered mode, wasteless journaling reduces up to an order of magnitude the latency required to flush the transaction log to the disk.

ordered modes.

As application-level workloads with asynchronous writes, we used the `fileserver` and `oltp` personalities of Filebench v.1.4.9.1 [62]. Similarly to SPECsfs, the `fileserver` emulates the I/O activity of a simple fileserver using an operation mix of file create, delete, append, read, write and attribute accesses. By default the number of threads is set to 50 and the mean size of appends is 16KB. We let the tool automatically configure the number of files to 250K based on the memory size of the server. From Table 5.1 it follows that the operation throughput (higher is better) of ordered mode is improved by 12.6% with data journaling and 17.5% with wasteless, respectively. Subsequently, we configured the mean append size of `fileserver` to 4KB. Then, the respective improvement became 21.4% with data journaling and 23.3% with wasteless. Selective journaling splits the data writes between the journal and the filesystem leading to operation throughput below that of ordered.

In the case of the `oltp` personality, Filebench performs the file system operations

	Fileserver (16KB)		Fileserver (4KB)		OLTP	
Mount Mode	**Thput (Ops/s)**	**Latency (ms)**	**Thput (Ops/s)**	**Latency (ms)**	**Thput (Ops/s)**	**Latency (ms)**
Ordered	579.2	314.6	576.8	315.5	779.8	182.2
Selective Jrn	493.8	368.9	559.2	326.1	810.2	156.3
Data Jrn	652.2	278.2	700.0	260.1	826.8	146.6
Wasteless Jrn	680.4	266.9	711.0	255.5	825.2	146.7

Table 5.1: We measure the performance of the `fileserver` and `oltp` personalities in Filebench. In `fileserver` we alternatively examine mean append size equal to 16KB (default) or 4KB.

of the Oracle 9i I/O model. By default, it uses 200 reader processes, 10 processes for asynchronous writing and a synchronous log writer. The tool automatically configures the file size to 600MB. The workload involves small random reads and writes, and it is sensitive to the latency of the moderate-sized (128KB+) writes to the log involved. Data and wasteless journaling achieve a limited throughput improvement (6%) with respect to the ordered mode, while selective journaling lies between the wasteless and ordered.

Overall, wasteless journaling improves the operation throughput of the ordered mode at improved bandwidth efficiency. The performance of selective journaling lies between the ordered and data journaling modes, while it reduces the respective bandwidth waste by transferring data updates either to the journal, or to the filesystem device. In the following section, we further examine the logging latency of databases by considering multiple concurrent workloads [32, 116].

5.2.3 Groupware and Database Logging

System administrators prefer to devote a separate device for the logs of I/O-intensive applications for efficiency [127]. Distributed systems place multiple log files locally at each machine for improved performance and autonomy [66, 32]. Also, database engines optimized for multi-core hardware maintain multiple log files on the same host [116]. Given the high cost of maintaining extra spindles in a machine, we investigate the possibility of serving multiple log files efficiently over a single disk with appropriate filesystem support.

In the present section, we measure the latency to serve the I/O traffic of log traces that we gathered from groupware and database workloads.

Jetstress

We consider the Jetstress Tool that emulates the disk I/O load of the Microsoft Exchange messaging and collaboration server [92]. We run Jetstress for two hours in a Windows Server 2003 system with 1GB RAM and two SATA disks in mirrored mode. We used 50 mailboxes with 100MB each and 1 operation per second for each mailbox. With these parameter values, we stress the hardware but also keep the reported measurements within acceptable levels to successfully pass the Jetstress test. The tool fixes the database cache to 256MB. Using the MS Process Monitor, we recorded a system-call trace of the Jetstress I/O activity. The I/O traffic of the database log contains appends of size from 512 bytes to tens of KB. The writes are tagged as *uncached*, i.e., they are configured to bypass the buffer cache and directly reach the disk.

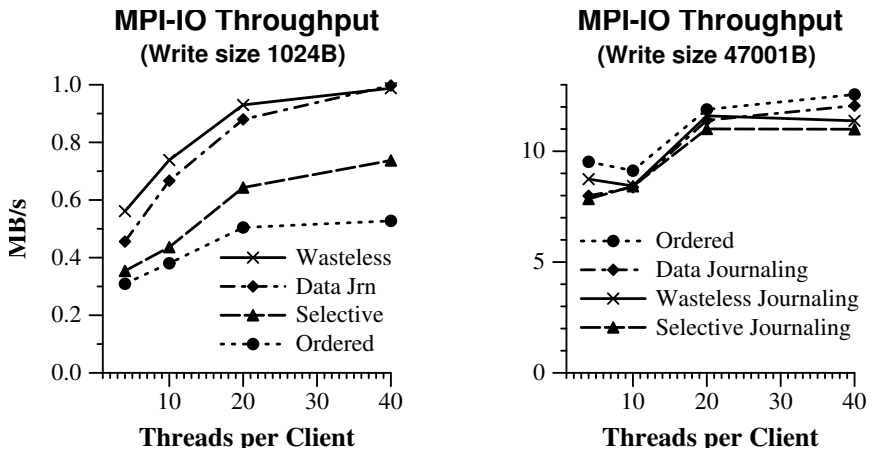
Over Linux, we use the original inter-arrival times to replay a 15min extract from the middle of the log trace. We consider different ext3 modes with the `O_SYNC` option enabled at file open for synchronous access. Additionally, we consider the ordered mode with the `O_DIRECT` option at file open to bypass the page cache. In order to study different loads and serve multiple logs from the same device, we varied the number of concurrent replays from 1 to 128. In Figure 5.4b, both selective and wasteless journaling keep write latency up to tens of milliseconds even at high load. Unlike wasteless journaling that writes to the journal all the affected data modifications, selective distributes across both spindles —of the journal and filesystem— the incoming appends. As a result, selective journaling achieves logging latency that is half that of wasteless or less. At high load, data journaling and ordered mode incur write latency that reaches hundreds of milliseconds, an order of magnitude longer than our two modes. These results indicate that the default uncached writes of Jetstress can be outperformed with appropriate filesystem support. We assume that the durability of synchronous writes is similar to that of bypassing the page cache.

TPC-C

We also examine the logging activity of the OLTP performance benchmark TPC-C [185] as implemented in Test 2 of the Database Test Suite [52]. We used the MySQL open-source database system with the default InnoDB storage engine [129]. After consideration of our hardware capacity, we tested a configuration with 20 warehouses and 20 connections, 10 terminals per warehouse and 500s duration. Running the benchmark led to insignificant differences of the measured transaction throughput among ordered mode, wasteless and selective journaling. This is reasonable because most updates in the workload have size above the write threshold; as a result, the disk operations are sequential regardless of whether they update the journal or the filesystem.

The InnoDB storage engine supports three different methods for flushing to disk the transaction log of the database. In default method 1 (*Cmt/Disk*), the log is flushed directly to disk at each transaction commit. It is considered the safest to avoid transaction loss in case of database, operating system or hardware failure. In method 0 (*Prd/Disk*), a performance improvement is expected by having the transaction log written to the page cache and flushed to disk periodically. Finally, in method 2 (*Cmt/Cache*), the transaction log is written to the page cache at each transaction commit and periodically flushed to disk. A transaction loss is probable in case of operating system or hardware failure.

During an execution of TPC-C, we collect a system-call trace of the MySQL transaction log. Subsequently, we replay a number of concurrent instances of the log trace over the ordered and wasteless journaling. We measure the average latency to flush the transaction log to disk. In Figure 5.4c, we see that wasteless journaling takes up to tens of seconds to complete each log flush across the three methods of InnoDB at high load. Instead, at 64 or 128 instances, ordered mode takes hundreds of seconds. We also experimented with selective journaling (not shown) and found it close to wasteless journaling and well below ordered. The reported behaviour is anticipated because wasteless and selective journaling sequentially store the small appends of the database log into the system journal.



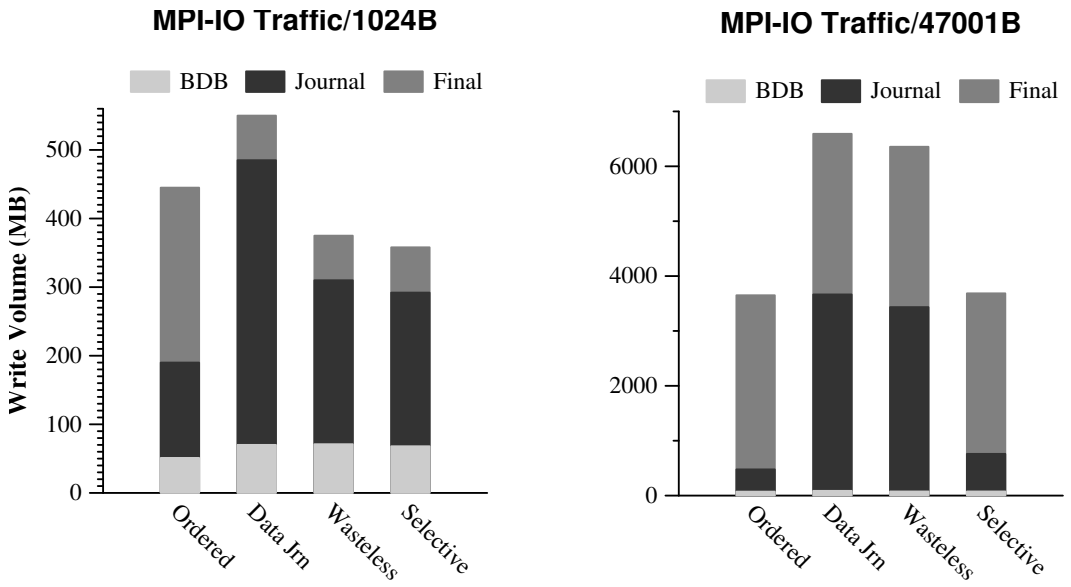
(a) (b)

Figure 5.5: We measure the data throughput (higher is better) of MPI-IO as client of PVFS2. (a) At 1KB writes, wasteless journaling almost doubles the performance of the default ordered mode. (b) At request size 47001 bytes, the prevalence of writes above the write threshold keeps similar the relative performance of the mount modes.

5.2.4 MPI-IO over PVFS2

Workload characterization of parallel applications shows the need for improved performance in small I/O requests over small and large files that arise due to normal execution and checkpointing activity [79, 34]. Especially small requests of 1KB are known to be problematic because they incur high rotational overhead even after they are transformed into sequential [147]. Writes of 47001 bytes also appear often in parallel applications and lead to poor performance due to alignment misfit [19]. In the present section, we examine the performance gain of a parallel multi-tier configuration with our mount modes running directly in the kernel-based filesystem of the storage server.

We chose the PVFS2 as an open-source scalable parallel file system [150]. We configured a networked cluster of fifteen quad-core machines with thirteen clients, one PVFS2 *data server* and one PVFS2 *metadata server*. By default, each server uses a local BerkeleyDB database to maintain local metadata. Through system-call tracing, we observed that the data server uses a single thread for local metadata updates and multiple threads for data updates. To focus our study on multi-stream workloads, at the data server we placed the BerkeleyDB on one partition of the root disk, and dedicated the entire sec-



(a) (b)

Figure 5.6: We measure the disk traffic (lower is better) of BerkeleyDB (BDB), the journal (Journal) and the filesystem (Final) over a PVFS2 data server. (a) At 1KB writes, selective and wasteless reduce the journal traffic of data journaling and the filesystem traffic of ordered. (b) At 47001 bytes, wasteless is similar to data journaling, and selective comparable to ordered mode, in terms of total disk traffic.

ond disk to the user data (filesystem and journal). We fixed the BerkeleyDB partition to ordered mode and tried alternative mount modes at the data disk. We used the default thread-based asynchronous I/O of PVFS2. Also, we enabled data and metadata synchronization, as suggested in the system guide to avoid write losses at server failures.

We used the LANL MPI-IO Test to generate a synthetic parallel I/O workload on top of PVFS2 [126]. In our configuration each process writes to a separate unique file ("N processors to N files"). According to previous studies, this is the write pattern suggested to application developers for best performance [19]. We varied between 4 and 40 the number of processes on each of the thirteen quad-core clients leading to total processes between 52 and 520. We tried 65,000 writes with alternative write sizes of 1024 and 47001 bytes. In Figure 5.5, we compare the data throughput of MPI-IO across different write sizes and loads. With 1KB writes, wasteless journaling almost doubles the throughput of ordered mode, while data journaling and selective lie between the other two. With writes

of 47001 bytes, the write throughput remains about the same across the different modes.

In Figure 5.6, we depict the total volume of write traffic across the BerkeleyDB, the journal and the filesystem. At 1KB requests, data journaling transfers to the journal 415MB, while wasteless and selective journaling reduce this amount by 42% (Figure 5.6a). The ordered mode writes to the journal 139MB, but transfers to the filesystem a total of 255MB. This amount is at least a factor of four higher with respect to the other three modes, which accumulate multiple small writes in memory before transferring them coalesced into the filesystem. At requests of 47001 bytes, selective journaling closely tracks the ordered mode in terms of total write volume. In contrast, data and wasteless journaling almost double the total disk traffic by double-writing the updated data blocks (Figure 5.6b).

In summary, wasteless and selective journaling at small writes improve substantially the performance of ordered mode, while they avoid the excessive journal traffic of data journaling. At larger write sizes, performance remains similar across the mount modes, but the journal traffic is higher for the data and wasteless journaling as they enforce stricter consistently between the data and metadata updates.

5.2.5 Recovery Time

In a different experiment, we evaluate the ability of the system to recover quickly after a system crash, which leaves the journal with log records before the respective updates are checkpointed to the filesystem, when the free journal space lies between $\frac{1}{4}$ and $\frac{1}{2}$ of the journal size, the original ext3 system automatically checkpoints the updates to the final location [148]. In order to do a fair comparison across the different modes, we use writes that are small enough to prevent checkpointing before the crash, but also useful for some application classes, e.g., event stream processing [29]. Thus, we start 100 threads each doing 100 synchronous writes of request size 8 bytes. Then we cut the power of the system. At the subsequent reboot, we verify that all modes fully and correctly recover the unique written data, while in the kernel we measure the duration of filesystem recovery.

In Figure 5.7, we breakdown the total recovery across the three passes that scan the transactions, revoke blocks, and replay the committed transactions, respectively. In comparison to data journaling, the *scan* pass of selective and wasteless journaling is an

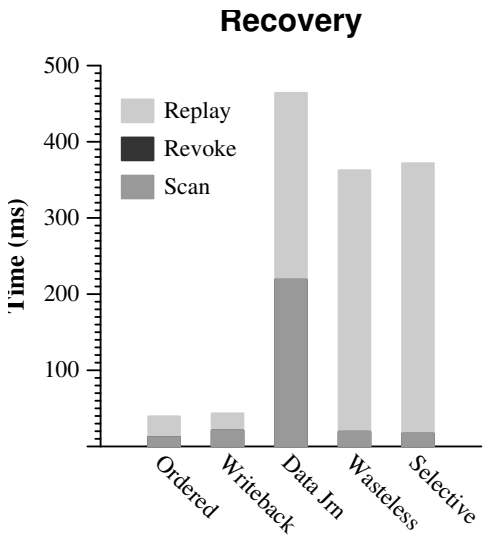
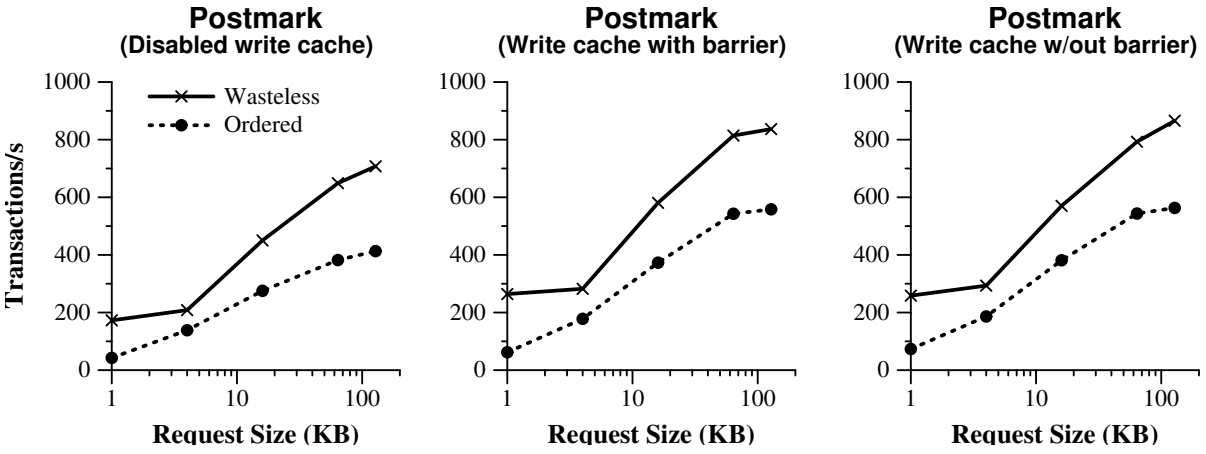


Figure 5.7: In comparison to data journaling, wasteless and selective journaling reduce the scan time of recovery by an order of magnitude but increase the replay time by about 40%. In total, they reduce the recovery time of data journaling by 20-22%.

order of magnitude shorter. This difference arises from journaling entire data blocks by data journaling, which significantly increases the amount of scanned data. The *replay* pass of selective and wasteless journaling takes about 40% more time than the ordered and writeback modes due to the extra block reads involved. Overall, selective and wasteless journaling reduce by 20-22% the recovery time of data journaling. In comparison to these modes, the recovery time of ordered and writeback is an order of magnitude lower at the cost of weaker consistency guarantees across the stored data and metadata.

5.2.6 Device Issues

We examine the sensitivity of our performance results to the settings of the on-disk cache and the use of write barriers (Section 4.1.4). The disk we experimented with (ST3300655SS) organizes the cache into multiple logical segments. It supports the `SYNCHRONIZE CACHE` command to force the transfer of all cached write data to the medium, and the `FORCE UNIT ACCESS` bit to enforce medium access on the basis of individual reads and writes [38, 160]. We kept the read cache always activated, and used the `sdparm` utility to configure the write cache. In Figure 5.8a we disable the on-disk write caches at both the filesystem and the journal, while we mount the filesystem with the option `barrier`=0



(a) (b) (c)

Figure 5.8: (a) With disabled the on-disk write caches, wasteless journaling improves the performance of ordered mode by a factor of 4 at 1KB requests and 73% at 128KB size. (b) We enable the on-disk write caches and mount the ext3 filesystem with `barrier`=1. Wasteless journaling improves the performance of ordered mode by a factor of 4.3 at 1KB requests. (c) If we enable the on-disk write caches with mount option `barrier`=0 (default ext3), the performance of ordered mode improves up to 18% at 1KB. However, the relative advantage of wasteless journaling with respect to the ordered mode remains significant (e.g. 3.52 times at 1KB).

(default ext3). We run the Postmark workload with the configuration of Section 5.2.1. It is not surprising that, for requests of subpage size 1KB, wasteless journaling maintains a performance advantage of four times in comparison to the ordered mode. The relative improvement drops to 50% at 4KB requests, and becomes 73% at 128KB requests.

In Figure 5.8b, we enable the write caches of the disks and mount the filesystem with `barrier`=1. Write barriers ensure that the write cache of the journal device is flushed before the commit block is written and also flushed to the medium. With enabled the write caches, the two mount modes improve their performance by 21-53% with respect to (a). In comparison to the ordered mode, wasteless journaling maintains performance advantage up to a factor of 4.25 at 1KB requests. In Figure 5.8c, we enable the on-disk write caches and mount the filesystem with `barrier`=0. In comparison to (b), the performance of ordered mode increases from 62 transactions/s to 73 transactions/s at 1KB, and up to 4.6% at larger request sizes. The performance of wasteless journaling without write

barriers (c) remains within 3.5% of that achieved with write barriers (b). Also, wasteless journaling improves the performance of ordered mode up to a factor of 3.52 at 1KB. We conclude that enabling the write caches improves the benchmark performance, while the use of write barriers incurs a relatively low cost, mostly noticeable in the ordered mode. In all the other experiments, we kept the write caches enabled on our disks and used the default ext3 mount option of `barrier=0`.

Arguably, wasteless journaling takes advantage of the two spindles that store the journal and the filesystem, respectively. Instead, the ordered mode mostly uses the spindle of the filesystem and less the spindle of the journal. To address this asymmetry, we also run our stream microbenchmarks over two SAS disks in RAID0 configuration with hardware controller support. We examine the two modes with the journal instantiated as a hidden file rather than a separate partition. With 1Kbps streams over RAID0, the write latency of ordered mode drops to half, while the write latency of wasteless does not change. Nevertheless, wasteless journaling remains one to two orders of magnitude faster than ordered mode across different numbers of streams. Also, wasteless journaling is up to an order of magnitude faster than ordered mode with 1Mbps streams.

5.3 Summary

Our experimental results include measurements of streaming microbenchmarks, application-level workloads, database logging traces and multi-stream I/O over a parallel filesystem in the local network. Across different cases, we demonstrated reduced write latency and recovery time along with improved transaction throughput with low journal bandwidth requirements. Especially we noticed that coalescing small data updates to the journal sequentially preserves filesystem consistency, but it reduces consumed bandwidth up to several factors, decreases recovery time up to 22%, and lowers write latency up to orders of magnitude. Furthermore, with a parallel filesystem, we showed that wasteless journaling doubles the throughput of parallel checkpointing over small writes, while it reduces the total traffic to disk.

Chapter 6

Improving the Durability of Distributed Filesystems

In the cloud infrastructure, a file-based storage interface is desirable because it provides improved performance and native file sharing support among different virtual machines. In this thesis, we set as main objective to improve the performance and durability of shared storage access in the datacenter. We recognize that a stateful filesystem client is part of the system failure model, since recently updated blocks that reside in the volatile memory of the client can be lost upon a crash. In order to improve the durability and efficiency of shared data storage, we increase the statefulness of each filesystem client with a local journal. Next, we specify the proposed design goals, describe the Arion architecture, and investigate the consistency semantics of the storage protocol along with its implications to the efficiency. Especially, we explain our design decisions regarding the event ordering and the resolution of conflicting client accesses in case of client failures, such as network disconnection and reboot.

6.1 Motivation

In a virtualization environment, network storage is often provided by scalable server clusters through protocols operating at the file, block or object level. The file interface is attractive for its sharing and efficiency properties [145, 80, 120, 191, 23, 112, 9]; the block interface provides convenient virtualization flexibility but incurs undesirable translation overheads [119, 106, 176, 177, 121]; and the object interface is scalable and efficient because it carries semantical information for specialized storage management [145, 189, 193].

Loss or corruption of committed updates to critical data is recognized as a particularly damaging class of failure [84]. This observation is highly relevant in a large-scale multi-tier environment, with mean time between failures inversely proportional to the number of machines. Several studies conclude that hardware failures contribute much less to service-level failures in comparison to causes related to software bugs and faults from operator or maintenance tasks [136, 84]. In particular, a recent study from Google reveals that misconfiguration and software errors account for the 29% and 34% respectively of the failures, while the hardware failures are below 10% [84]. Similar statistics from Facebook show that host failures are typically rare, with an observed Annualized Failure Rate around 1% of their disks [128]. Additionally, only a small percentage (0.5%-1%) of the nodes in a datacenter has been reported to not come back to life after power outages [45].

Another design dimension in datacenter storage applies client-side caching for improved performance and durability at reduced network and server load. Existing solutions often apply block-level caching at the client-side host, and they adopt write-through or writeback policy according to the application and hardware characteristics. A write-through policy is preferred for read caching without data loss at device failure. Instead, a writeback policy improves the resource efficiency and application performance but makes the cache device part of the failure model [119, 166, 31, 101, 151, 89].

The Arion system is a new design point that we introduce in cloud storage to improve the durability of the file interface at the client side (Figure 6.1). We integrate the client software of a distributed filesystem with persistent host-based storage over a journal device. We enhance the flushing functionality of the filesystem client with tunable control of both the amount of dirty pages that are staged at the host, and the time period taken by dirtied pages to reach the backend servers. At increased flushing frequency to the journal

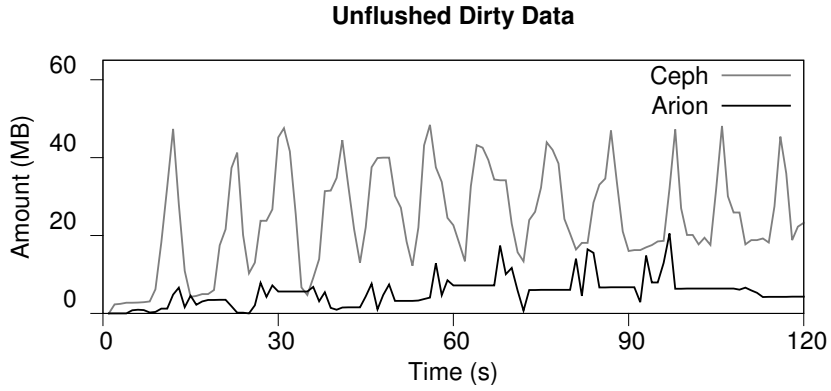


Figure 6.1: Dirty data that remains unflushed in the volatile memory of the client in Ceph and the proposed Arion system.

device, we practically minimize the recovery point objective (RPO) close to zero under the following condition: the dominant cause of a client crash is operator or software bug rather than permanent hardware loss, such as that of the local storage device [136, 84].

In traditional Unix, written data is acknowledged asynchronously to the application but only flushed periodically to the local disk. This approach has been adopted by several distributed filesystems in the form of asynchronous data transfer from the volatile memory of the client to the servers [131, 115]. Although durable caching at the client side can reduce the network load of the servers, it complicates the maintenance of replication consistency among different clients or between the clients and the servers [85].

In Figure 6.1, we measure the amount of dirty data that remains unflushed at the client memory over time. We compare Ceph [193] under default flushing parameters with the proposed Arion system (Section 6.2). In the environment of Section 8.1, we used the fileserver mode of Filebench [62] running for 2min over 10000 files. The Linux pdflush daemon wakes up every 5s and transfers dirty data older than 30s from the client to the servers [28]. Additionally, the Arion client every 1s flushes dirty data to the local journal of the host. On average over time, the Ceph client keeps 24.3MB of dirty data solely in volatile memory, i.e., unrecoverable from a crash. Instead, the Arion host-side journaling reduces to 5.4MB the vulnerable data in the volatile memory of the client.

Our main contributions are the following:

1. We improve the durability of frontend memory caching by integrating disk-based

journaling into the client of a distributed filesystem.

2. We implement a prototype of the proposed storage layer in the kernel-level client of the Ceph object-based filesystem.

3. We carefully investigate the consistency semantics of the proposed storage protocol.

4. We experiment with several application-level benchmarks over a virtualized host and a clustered storage backend.

Overall, in a host machine with reliable local storage, we approximate the consistency ordering and durability of write-through caching with the configurable efficiency of periodic writeback.

6.2 System Architecture

Next we outline our assumptions and goals before we describe the main design ideas of Arion.

6.2.1 Assumptions and Goals

We aim to improve the durability and performance of shared storage in the datacenter at reduced utilization of the server resources. User is the application-level entity that initiates I/O requests to the filesystem, and client is the host-based software that provides filesystem access to users. We target host hardware with reliability characteristics on par with those of the server machines. The host provides directly-attached storage with sufficient redundancy to tolerate the occasional failure of a single device. Appropriate storage technologies include hard disks, solid-state drives, or non-volatile memory. In the proposed storage architecture we aim to support the following properties:

i) **Interface** Stored data is directly accessible for regular use and maintenance tasks over the network with a POSIX-like file-based interface [181].

ii) **Sharing** Heterogeneous clients on the same or different hosts can natively share data at the storage level but may also apply synchronizations at the application level.

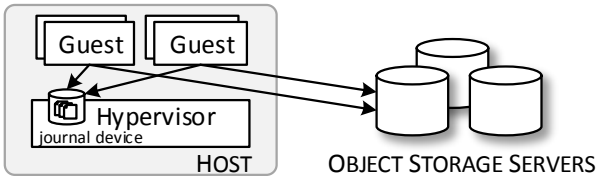


Figure 6.2: Host-side journaling in the Arion architecture.

iii) **Durability** Most recent writes survive client reboots but require redundant hardware support to tolerate permanent failures of individual storage devices at the host.

iv) **Performance** Client writes are safely stored at sequential disk throughput, but the read performance depends on the efficiency of the client memory cache.

v) **Scalability** The storage backend linearly scales out to efficiently hold increasing amounts of data.

6.2.2 Design

We rely on an object-based scale-out backend of multiple data and metadata servers (Figure 6.2). The client runs over either a guest system on virtualized hardware or a standalone system on bare metal. A read operation synchronously returns the latest version of the requested state. A synchronous write reaches a configurable number of durable replicas before it returns. An asynchronous write returns as soon it updates the buffer cache of the client system, but the modified blocks have to reach a configurable number of durable replicas before they are considered safely stored.

We regard the frontend logging to a persistent storage medium as a complementary form of replication. Unlike the traditional replication that is homogeneously applied across functionally equivalent backend servers, the frontend logging adds *heterogeneity* with respect to the storage format, the logical layer and the time duration of the replica[1].

The metadata server (MDS) enables shared filesystem access at *file* granularity through different types of tokens leased to the clients. Supported access types include exclusive write (cached) by a single client, and concurrent read (cached) or concurrent write (uncached) by multiple clients. A client can only cache the writes of data and metadata

[1]The Coda filesystem previously introduced the concept of two-tier replication in the context of disconnected operation [100](see also Section 9.3).

accessed with exclusive permission. The file interface provides valuable semantical information about the consistency dependencies of modified data and metadata. When a client transfers the file updates to the servers, the metadata is written only after the referenced data blocks have safely reached the server state.

The key innovation of our design is the integration of a local journal with the kernel-level client of a distributed filesystem (Figure 6.2). The host-side journal is distinct for each guest in virtualized hosts. The client inserts into the journal *both* the data and metadata modified by an I/O request. Thus we ensure that a metadata version matches the version of the data it refers to (version consistency [43]). We only keep one transaction active to accept all the (redo) records of low-level I/O operations corresponding to an atomic filesystem request. An active transaction closes as a result of timeout expiration, explicit flush request, or reclamation of journal space [28].

A journaled block remains cached in the client memory until it is safely written to the servers. If an MDS revokes the write token from a client due to some conflict (e.g., concurrent writes to the same file), the client is forced to write (checkpoint) the conflicting writes to the servers and invalidate the respective journal records. On client disconnection from the servers, the leased tokens may expire and the client will no longer be able to access the files locally [115]. At network reconnection, the client writes to the servers the mutated blocks of each file whose token has been refreshed and whose metadata cached at the client is newer than the file metadata at the MDS, but it discards the remaining blocks.

The primary benefit from host-side journaling in a distributed filesystem is the reduced vulnerability of outstanding writes in the volatile memory of the client. If a client crashes and reboots without hardware failure at the host, then the client replays the completed transactions and transfers the recorded updates to the filesystem servers. We update a file only if the replaying client confirms token ownership, and the journaled metadata is newer than the file metadata at the MDS. In case of client crash during the recovery, the replay is repeated until the client journal is fully checkpointed.

The durable storage of recent writes over the host-side journal improves the server writeback efficiency with respect to the utilized network and disk bandwidth. The consumed shared resources are reduced through batching applied to repetitive writes over the same blocks, or to small writes. At synchronous writes, we journal the updates lo-

cally and postpone the server writeback as permitted by the flushing parameter settings. Thus, performance improves depending on the pressure over the shared resources and the resulting queuing delays in the I/O path of the Arion networked storage.

6.2.3 Consistency

We strengthen the durability of memory-based caching in clients that provide native support for file sharing. The file interface differentiates the data blocks from the metadata. Thus, our system cleanly addresses issues of vertical (client-server) and horizontal (client-client) consistency across different replicas [31]. In the order imposed by their arrival time and structural dependencies, the data and metadata updates are first journaled at the local host and subsequently persisted at the backend servers. Additionally, the filesystem arbitrates the conflicts among different clients through lease-based tokens. In contrast, block-based schemes typically operate transparently to the filesystem, and as a result explicitly track the order and *relax* the durability of block updates [42, 101, 121].

Traditionally, the filesystem state comprises three different types of entities: the external namespace, the internal structure and the user data. The state of each entity type has to meet specific conditions in order to remain consistent over time. Respectively, we refer to the consistency of each entity type as *name*, *structural* and *data* consistency. For brevity, we use the *metadata* consistency for referring to both the name and structural consistency.

The system consists of client and server nodes. A client node runs applications along with the software that makes the filesystem visible to the applications. The server nodes undertake all the functions —except for the client functionality— related to the storage of the namespace, the structure and the data. From the filesystem point of view, the servers are replicated and the clients are not. A client failure results into network disconnection from the servers, or complete termination of operation. The network disconnection is temporary; instead, the operation termination can be temporary, such as a common reboot, or permanent, such as a malfunction of the local storage hardware. We assume that the probability of permanent client failure can be substantially reduced through appropriate redundancy in the local storage hardware.

We strengthen the role of the client because it can partly undertake responsibilities of

the server. Although the client does not vote for the persistence of an update as regular servers do in quorum-based consistency [182], it maintains a copy of the update in durable medium and is responsible to assign the timestamp that will order the update with respect to the other activity of the system. In that sense, our storage architecture is asymmetric and the replication heterogeneous.

Durability semantics

Namespace updates are transferred synchronously from a client to the servers for ensuring consistent maintenance of the namespace state. Instead, other metadata and the user data can be cached at the client for improved performance but somewhat weakened consistency. We regard the system call of a filesystem operation as the system unit of atomicity. Operation atomicity is ensured by forcing to the servers the data updates before the respective metadata updates. As a result, an interrupted data update is not visible to subsequent readers unless it is successfully retried by the client.

All namespace operations are synchronous. In our design this means that the namespace updates are transferred to the servers and subsequently acknowledged to the client. The updates of data and other metadata are either synchronous or asynchronous, depending on the type of the requested operation and whether there are other clients concurrently accessing the same object. In case of concurrent client accesses, an update has to be synchronously transferred to the server. Otherwise, we transfer the updates of a synchronous operation to the journal before we acknowledge it to the client. Instead, we acknowledge an asynchronous operation as soon as the modified blocks are copied to the kernel memory, but we only periodically append them to the local journal.

During normal operation, the lease token granted to a client allows the client to locally cache the updates and periodically transfer them to the server. In case of synchronous writes, the client transfers to the server the prior synchronous writes that are already locally journaled, before it appends to the journal the latest synchronous writes. If a client receives a lease revocation for an object, it disables the caching of the affected object and transfers all the locally journalled updates to the server. The above operation ensures that incoming updates are made durable according to the order at which the operations arrive to the client and the respective writeback period expires for each of them. The system achieves version consistency because the version of the data matches

that of the metadata as a result of the metadata updates propagating to the server after the respective data updates [43]. This is held assuming that the journaled updates can be replayed in case of a temporary client crash, otherwise data consistency is offered.

Network disconnection

In case of network disconnection, we have to consider the duration of lost connectivity. If the client remains disconnected after the expiration of a lease, the server can grant the lease to a different client. This means that the first client has not necessarily transferred to the server all the updates that were temporarily copied into memory and/or appended to the journal. The clients that subsequently access the same object are given the right to read or write the object.

We aim to approximate the POSIX semantics in a distributed filesystem to the degree that we can also provide a reasonable performance and efficiency. According to the POSIX semantics a write should be visible to any subsequent read in the system [181]. In normal operation, this semantics is offered by the fact that clients either alternate exclusive access to an object, or concurrently read the object with local caching, or synchronously update the object without local caching. This behavior approximately allows the updates to become visible across the system through the appropriate arbitration by the server. In the case of disconnection with expired lease, some updates remain at the client memory. An intervening access by a different client is recorded through the updated object timestamp at the server.

Event ordering

We need a total order of the reads and writes to an object by the clients and servers of the system. When a client c modifies the object o, it assigns a timestamp $T_c^w(o) = T_c^l$ of the local modification time T_c^l. When the server s receives an updated object o from a client c, the server assigns the write timestamp $T_s^w(o) = T_c^w(o)$. Accordingly, when a server s transfers a copy of an object o to a client, the server assigns to the object the timestamp $T_s^r(o) = T_s^l$ of the local read time T_s^l. Similarly, if a client c obtains a local copy of an object o, it assigns a local timestamp $T_c^r(o) = T_s^r(o)$.

In the case of concurrent writes to the same object, clients have to synchronously

transfer the updates to the server before confirming them as completed to the applications. It is possible that a received update from client c has write timestamp $T_c^w \leq T_s^w$. This is possible as a result of the variance in the delay of transferring the update from different clients to the server. One possibility for handling this case is to define a configurable time window W within which we accept incoming updates from the clients that have been delayed with respect to the current server time, e.g., in the time interval $[T_s^l - W, T_s^l]$.

If multiple clients obtain shared read access to the same object, each of the clients creates a local copy of it and maintains it as long as there is no conflicting operation at the server, i.e., object write. If a client obtains a lease token for exclusive write access to an object, then the client creates a local copy on which it can apply updates as long as the lease remains valid. Such an object carries the write timestamp of the most recent local update. Thus, if the system operates normally, the system makes visible to the clients the latest state of each object.

Throughout the duration of a write lease token, the client is allowed to apply updates and accompany them with a locally-generated timestamp. This is possible because the write token guarantees that there is no concurrent access to the same object by a different client. Additionally, the token has been granted sufficiently recently to ensure that the client has interacted with the server and synchronized its clock. The lease model provides mutual exclusion that allows different clients to alternate write access to the same object at concurrency granularity in the order of the lease duration. Therefore, we should be able to order the accesses of different clients according to the relative order of the leases to which they belong. In our model the unit of atomicity is an individual operation that affects a small number of objects in a predefined way.

Our model places less strict requirements of concurrency control in comparison to traditional transaction processing. The main source of complexity in transaction processing arises from the existence of concurrent operations to different objects and the grouping of operations into atomic transactions that should be serializable [22] (or linearizable [77]) for reasons of consistency. Transaction processing prevents deadlocks typically by assigning to each transaction a timestamp used to resolve conflicts among the concurrent operations from different transactions. The parallelism at different parts of the system requires a centralized entity (called oracle) to grant monotonic timestamps, or a complex infrastructure for high synchronization accuracy.

The requirements of total event ordering across the different system nodes can be achieved by Lamport's logical clocks [105]. Alternatively, the real-time clocks of different nodes can be synchronized at sufficient accuracy in the order of a millisecond (NTP [123]) or even microsecond (PTP [59]). We assume that a timestamp is derived from the logical or real-time clock concatenated with the client identifier in order to resolve ties among the clients [182, 154].

Conflict resolution

At network disconnection, the client will not be able to obtain new lease tokens. Additionally, it will be impossible for the client to renew tokens that it already holds or receive revocation notifications for conflicting requests from other clients. Consequently, it is possible that a token will expire or be unilaterally revoked by the server before the respective client is able to transfer the latest updates to the server. Subsequently, a different client may obtain the token and read or write the object. If the first client is reconnected later, it may obtain the lease for the same object. At this point, the client has to decide how to treat the locally cached updates that were not previously transferred to the server. We assume that the updates of the client carry the local timestamp of the update event.

We treat each read or write operation as a separate atomic event. The writes that have been locally cached at the client, are already acknowledged to the application. Therefore, we have to make them durable to the server at the respective timestamp. If before the reconnection, a different client has already read the same object, then it missed the cached update. In this case, we decided to completely skip the respective update and notify the local client accordingly. A similar approach is followed by early work on atomic actions to preserve the correctness of object reads [154], and in the handling of read-write conflicts in multi-version timestamping [21]. More recently, Tango rejects a write whose reads are found obsolete at commit time [15].

We disregarded the alternative option of transferring the update to the server and attaching to it the current server timestamp. This approach could be confusing for the application, especially if the disconnection period is longer than a few seconds. The possibility of network disconnection exists in all distributed systems. Typically, writes are asynchronous for reasons of performance but with open the possibility that an acknowledged write will be lost in the case of client failure.

In another case, one or more different clients update the object before the first client manages to reconnect to the server. Inspection of the update timestamp at the server informs the client about this event. The client cannot transfer the locally cached updates to the server, for several reasons. The updates can be based on object value potentially read before the recent update by different clients. The update of the client bears a timestamp that is earlier than the timestamp of the latest updates. Therefore, the updates of the client for the same object are no longer valid, and have to be discarded. This approach is similar to Thomas' write rule, previously proposed for the synchronization of database replicas [182]. We presume that the order at which leases for an object are granted by the server to different clients allows to overcome reported concerns about the accuracy at which writes from different clients are ordered [167].

In the case of large objects —such as the files in a distributed filesystem— it is likely that the conflicting writes refer to different offsets of the object. Therefore, the conflict is a result of false sharing rather than true offset overlap in the modified bytes. We believe that the probability of false sharing can be substantially reduced if the leases refer to byte ranges rather entire objects. This is similar to the notion of range locking that was proposed in early work on distributed filesystems [99].

The disconnection case results into dropped updates at the affected client even though these updates were previously acknowledged as completed to the user. Depending on the nature of the application, this can be avoided if the client can request synchronous write with disabled local journaling. In practice, asynchronous writes are the norm due to their improved performance and the rarity of the network failures. In our approach, we cannot follow approaches developed in transaction processing, because the filesystem does not inherently support transactions over arbitrary groups of operations across different objects. Similarly, we cannot rely on methods of *eventual consistency*, because eventual consistency by definition requires that all updates are eventually reaching all the servers of the system [144]; this is not possible in our system given the correctness constraints that lead to discarded updates in some corner cases.

In the above cases, the system satisfies the requirements of data consistency, in the sense that the metadata of a file point to the data of the file itself.

Client reboot

It is possible that the client disconnection is accompanied by a local reboot. In this case, the memory contents are wiped out, but we can rely on the local journal for recovering most of the previously acknowledged updated that were not transferred to the server. We follow the same rule that we described above for the network disconnection. The only difference is that we need to retrieve the updates from the journal to the memory of the client and then transfer them to the server as long as there was no occurrence of conflicting read or write from a different client in the meantime. It is reasonable that all the data and metadata updates which were not safely copied to the journal before the client crash, are lost.

In the case that we have a sequence of back-to-back reboots, the recovery procedure spans multiple reboots. By following the conflict-resolution rule based on timestamps, an update that has been already restored at the server from the recovering client, is simply discarded by the client when retried at a subsequent reboot. Instead, those updates that were not completed, will eventually be restored, as long as the recovery is not intervened from a different client accessing the same file.

According to our approach, updates from a client are propagated to the server in the order at which occurred. This is ensured by having a constant time period for each update at the expiration of which the update is transferred to the server. Similarly, in the case of recovery, the updates are propagated to the server in the order at which they are found within the journal. However, journaling occurs in the order at which a fixed time period expires after the completion of an update in memory. Updates to different objects are relatively ordered according to their attached local timestamps. Concurrent updates to the same object are forced synchronously with the possibility of configurable out-of-order tolerance and the assumption that strict ordering is enforced at a higher level through locking.

6.3 Summary

In order to enhance the end-to-end durability of shared storage, we propose the integration of the client of a distributed filesystem with a host-based journal. We explored the possi-

bility to combine the performance and durability of the local device with the availability and manageability of a distributed filesystem. We focused on the provided consistency semantics of the proposed storage protocol, with special interest on handling client failures such as network disconnection and reboot. Finally, we argued about our design decisions regarding the event ordering and the resolution of conflicting client accesses.

Chapter 7

The Arion Prototype

In this chapter we initially present the necessary background information in order to subsequently give a detailed description of the Arion prototype. In our prototype implementation, we integrated the Linux JBD into the CephFS kernel-based filesystem client of Ceph. We implemented the Arion host-side journaling based on Linux JBD2 and the kernel-level client of Ceph (v0.80.1). The Arion development required 3417 new commented lines across 15 files of Linux kernel (v3.6.6). Our current prototype implementation fully supports (i) the journaling of mutated data and metadata from the client memory to the host-side journal, (ii) the filesystem recovery to a consistent state after a client crash that leaves the host hardware operational, and (iii) the checkpointing of journaled data to the storage servers for journal space reclamation.

7.1 Background

7.1.1 Linux

The Linux kernel maintains in memory a page cache with data and metadata blocks of recently accessed disk files [28]. A page descriptor stores bookkeeping information about the address space and the inode of a page. For every disk block cached in memory, there is a block buffer that stores the actual data, and a buffer head structure that maintains bookkeeping information about the block. The dirty pages are written to disk in several cases: at timeout expiration, under space pressure in the main memory or the journal device, and by explicit flush request from the user.

The Linux kernel implements filesystem journaling with a special kernel layer, the Journaling Block Device (JBD). The journal is physically implemented as either a device partition or a hidden local file. The records of multiple low-level operations from a system call are stored as a single transaction in the journal. For the journaling I/O of each block buffer, the kernel dedicates a separate buffer head structure. Additionally, a journal head structure links each block buffer with the respective transaction.

The journal commit operation writes to the journal the dirty buffers of a transaction followed by a commit block. The kernel also allocates buffers for one or more journal descriptor blocks. The corresponding journal blocks are used to mark the beginning of the transaction and store the list of tags that identify the journal blocks of the transaction.

7.1.2 Ceph Architecture

The Ceph is an object-based parallel filesystem designed for scalability, performance and high availability [193]. It consists of four main components: the clients provide a POSIX-like filesystem interface; the metadata servers (MDS) manage the namespace hierarchy; the object storage devices (OSD) reliably store data and metadata; and the monitors (MON) manage the server cluster map. Figure 7.1 depicts the interaction between the four components.

Ceph targets scalability by separating the management of filesystem metadata from the storage management of the respective data. It also improves the scalability of the system further through dynamic distributed metadata management. Ceph decouples data

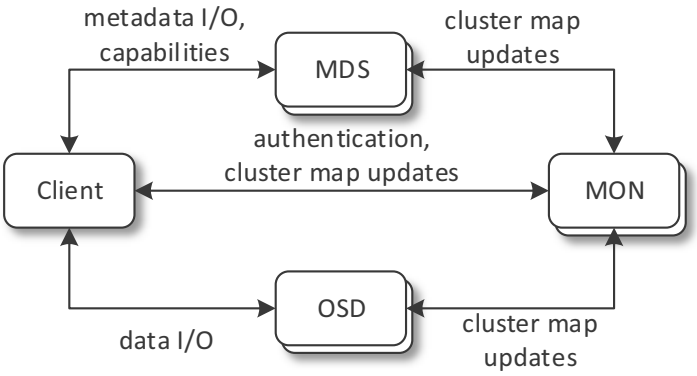


Figure 7.1: The relation between different Ceph components.

and metadata operations by eliminating file allocation tables and replacing them with *CRUSH*; a pseudo-random data distribution algorithm. Instead of relying on a central lookup table, Ceph clients and OSDs can efficiently compute information regarding the object distribution across the cluster by means of the CRUSH mapping function. In particular, a cluster of MDSs is responsible for the metadata operations, whereas each client communicates directly with the OSDs for the file I/O operations. Besides providing high scalability and improved performance, CRUSH also ensures data safety and high availability in case of server failures. An asynchronous communication model is used for the inter-node communication, whereas TCP guarantees the ordered and reliable delivery of the exchanged messages. In case of communication failure, the sender is asynchronously notified.

7.1.3 Ceph Data and Metadata Management

A set of MDSs acts as a scalable, consistent, distributed cache of the file namespace. The metadata is cached in memory, but it is also persistently stored across the OSDs as a collection of regular objects. A journal of recent metadata updates in each MDS allows their efficient transfer to storage media.

Ceph OSD servers collectively provide the abstraction of a single shared object store. For this purpose, OSDs are organized into a *Reliable Autonomic Distributed Object Store* (RADOS) cluster. In order to forward an I/O request to the proper OSD servers, Ceph utilizes a cluster map to specify which OSDs participate in the storage cluster and how data is distributed among them. The cluster map allows RADOS to perform tasks such

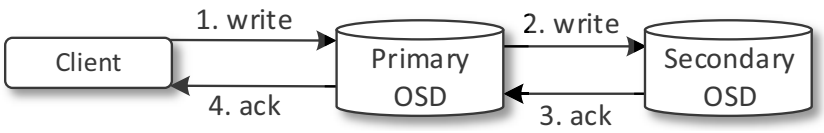


Figure 7.2: A replication example with two replicas.

as data migration, replication, failure detection and recovery. A monitor service, which consists of a small set of monitor processes coordinated through Paxos, is responsible for manipulating the cluster map.

Ceph maps each object to a placement group of multiple OSDs identified through CRUSH. In particular, each object is mapped to a specific placement group, while each placement group for replication purposes is stored in multiple OSDs. Placement groups can also be dynamically assigned to OSDs for load balancing. Each OSD manages its local storage typically using the B-tree filesystem (btrfs) for its advanced features. A local journal allows an OSD to maintain multiple versions of every updated object, and serialize the individual updates within the placement group. Generally the journal improves the I/O performance, since small writes can be safely delayed and batched before reaching the filesystem. Moreover, random writes can benefit from the sequential disk throughput of the journal. Upon a write request, an OSD writes to the journal a description of the request and flushes the update to the filesystem. Periodically, the OSD has to synchronize the journal with the filesystem to reclaim journal space. In case of an OSD recovery after a crash, the OSD replays the operations from the journal to bring the filesystem to a consistent state.

RADOS manages the replication of data across multiple OSDs for fault tolerance. A variant of the primary-copy replication policy is used to serialize all the incoming updates that refer to a particular placement group. Writes are replicated synchronously in order to provide strong consistency. Specifically, a write request is initially directed to the primary OSD of the group. Then, the primary OSD forwards the request to the remaining replicas. The client is acknowledged only when the update has been safely written to the disk of all the replicas in the placement group (Figure 7.2). Especially, with respect to acknowledging a write operation, an OSD applies the update both to the journal and the local filesystem in parallel.

7.1.4 Ceph Client

A Ceph client can take advantage of one of the following service interfaces:

- The *CephFS* service, which provides a POSIX filesystem usable with mount or as a filesystem in user space (*FUSE*).
- The *Ceph Block Device (RBD)* service, which provides thin-provisioned block devices with features such as snapshotting and cloning.
- The *Ceph Object Storage (RGW)* service, which provides RESTful APIs with interfaces that are compatible with Amazon S3 and OpenStack Swift.

Upon a write request in an address space, the kernel-based filesystem client prepares a page in the cache. A partial page update first fetches the original page from the OSDs; subsequently, the kernel copies the user modifications to the page and marks the inode object as dirty. The writeback of dirty pages occurs asynchronously as the Linux pdflush threads wake up periodically to scan the list of dirty inodes and writes their dirty pages to the OSDs. In Linux, the writeback time refers to the wake-up period, and the expiration time refers to the time length after which a dirty page is flushed. At notification by the OSD for the data writeback completion, the client transfers the dirty inode to the MDS and receives acknowledgement when the inode update is safely stored.

The client access to the data and metadata of a file is controlled by the MDS by means of *capabilities*. A capability is a set of bits that indicate which operations are permitted to the client for a particular inode. In order to cache an inode, the client must hold the respective capability. A client can hold an exclusive access capability, which allows it to modify the inode locally and propagate the updates to the MDS asynchronously. In case of a shared access capability, the client is assured that it has a consistent view of the inode. The MDS can revoke conflicting client capabilities to prevent inconsistencies. In addition, locks ensure the correct serialization of updates that span multiple objects. A Ceph client keeps an open session with every MDS in the cluster. It periodically contacts an MDS to renew its held capabilities from the respective session. A capability can be released when it is no longer needed.

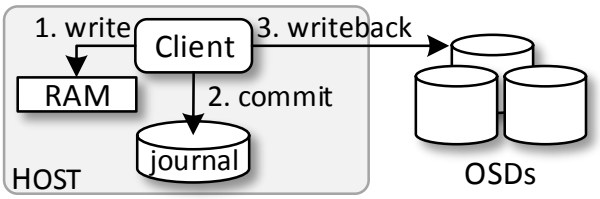


Figure 7.3: A write request is applied to the kernel memory, then added to the host journal and finally reaches the servers.

7.2 Implementation

7.2.1 Mounting Ceph

In our prototype implementation, we incorporate a local journal at the CephFS client (Figure 7.3). At mount time, Ceph initially parses the mount options and creates a Ceph filesystem client. It also allocates a special `ceph_fs_client` data structure to store client-related information. At this point the kernel initializes a messenger instance for the communication with the other hosts in the system, and a monitor client which is responsible for the interaction with the monitor instances. The monitor client always keeps an active connection with a monitor so as to receive map updates and send periodic keep-alive messages for the detection of connection failures. An OSD client is responsible to calculate the data layout according to CRUSH and submit read and write requests to the proper OSDs. In addition to that, the OSD client keeps track of pending I/O requests and in case of communication failures it retries the affected operations. Ceph also initializes an MDS client for the metadata handling operations. The MDS client keeps an open session with a set of metadata servers and forwards any metadata requests. Keep-alive messages ensure the liveness of any held cap or lease.

We expanded the `ceph_fs_client` data structure of CephFS by adding two extra fields: the `journal_bdev` refers to a specific block-device control structure in the kernel, and the `s_journal` refers to the journal control structure of the journal block device. We pass the journal block device to the kernel through the new mount option `journal_dev=<journal_path>` that we added. The function `ceph_load_journal()` is responsible to initialize the journal control information at mount time. Particularly, it reads the journal from the block device and allocates the appropriate in-memory control

structures. Eventually, the journal is released when the Ceph client is destroyed (e.g., `umount`).

7.2.2 Buffer Management

Upon an asynchronous write request, the kernel client invokes the `ceph_aio_write()` function. The client initially gets the appropriate capability for write and buffering, marks the capability as dirty, and invokes the `generic_file_aio_write()` function. For every page involved in the I/O, the `ceph_write_begin()` method of the address space object is used to prepare the page cache. It searches into the page cache and, if necessary, it creates a new page at a given page cache position. This step also ensures that only clean pages or pages that were dirtied within the same snapshot context can be modified. In case of partial page updates, the corresponding page is fetched from the proper OSD. Next, the kernel copies the modifications from the user-mode address space to the page cache and the `ceph_write_end()` method marks the corresponding buffer page as dirty. The inode object is also marked as dirty for writeback. Additionally, when the client closes the file, it relinquishes the corresponding capability to the MDS.

For the journaling support in the CephFS client, we modified the `ceph_write_begin()` function to allocate disk block buffers and buffer heads during a write. We insert each block buffer into the active transaction of the journal by creating a journal head. When a block buffer is eventually updated, it should be added to the dirty list of the active transaction.

The actual data transfer takes place during writeback. The client calculates the location of the data objects and communicate directly with the proper OSDs. The Linux kernel pdflush threads periodically scan the list of dirty inodes and asynchronously flush their dirty buffers to stable storage. For a particular inode, the `ceph_writepages_start()` function prepares and schedules an OSD message including the modified pages. When the writeback is complete, the OSD sends a reply message to the client. An asynchronous callback function at the client is assigned to flush the inode along with the dirty capability to the MDS. Finally the MDS sends back an acknowledgement to state that it has safely committed the inode updates. Figure 7.4 depicts the the above steps.

In our prototype, we adjust properly the writeback timeouts of the kernel in order to

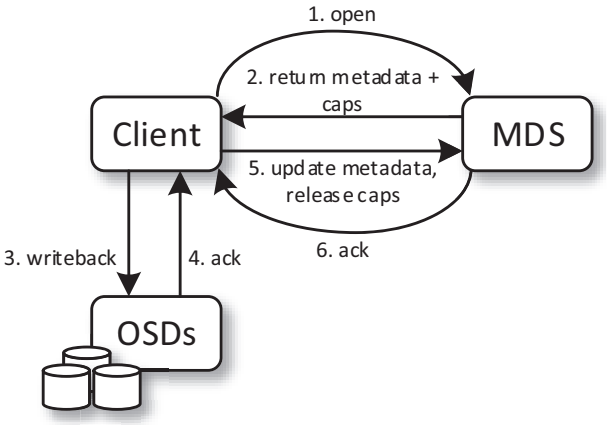


Figure 7.4: A simple example of writing to a file.

delay the actual network transfer. When a particular page is invalidated or written back, we invalidate the related journal entry through the `ceph_journal_invalidate_journal_entries()` function. Specifically, we (i) unmap the journalled buffer in order to ensure that it won't be checkpointed; (ii) free the corresponding journal block buffer; and, if possible, (iii) drop any related transaction.

A kernel page can be in various states which are indicated by specific flags. In a journalled filesystem, upon a write request the corresponding page is allocated and marked as `up-to-date`. The page is later marked `dirty` during the commit phase. Eventually, the page can be safely cleaned and freed after being persisted, either written back or checkpointed. Since Ceph is not a journaling filesystem, the updated page is by default marked as dirty during `ceph_write_end()`. Instead, we add an intermediate state, called `JBD_state`, to indicate that a page has been marked as ready for journaling but has not been committed yet to the journal. For this purpose we use the `PG_JBD` flag. Thus upon a write request, the page is also marked as `JBD` and, subsequently marked as `dirty` during commit.

The `private` field of a page descriptor is typically used by the local filesystem to link the page with the respective block buffer. Instead, Ceph uses the `private` field to record the context required to support the snapshot service. In Arion, we introduce a special data structure, called `ceph_metapage`, to associate a page with the block buffer, the snapshot context, and inode-specific information. We use the `private` field for linking the page descriptor to the respective metapage.

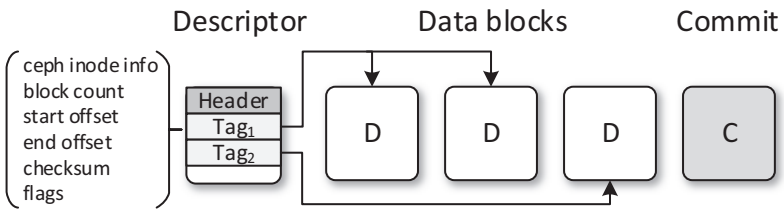


Figure 7.5: A descriptor block contains multiple tags, each corresponding to the block updates of a particular inode. Every tag includes (i) information regarding the inode; (ii) the number of the following data blocks in the journal; (iii) starting and ending offsets; (iv) a journal-specific flag; and, (v) a checksum.

7.2.3 Metadata Management

The journaling of metadata operations within Ceph was particularly challenging to implement because there are several cases in the I/O path at which an inode is marked as dirty. Unlike a local filesystem, an inode object of Ceph does not correspond to specific disk blocks known by the client. Instead, we initially store into the `ceph_metapage` the inode information of each block buffer including the inode version in order to reflect the corresponding write modification. We also introduce new Ceph-specific tags in the journal descriptor block to store the inode-related information.

In particular, during commit the descriptor block fills with fixed-length tags, with each tag corresponding to the block updates of a particular inode (Figure 7.5). We replace the descriptor tag of the traditional JBD with a new journal block tag that we introduce. Originally in JBD each tag corresponded to one block buffer and contained the filesystem location of the modified block, one flag for the journal-specific properties of the block and a checksum. In our design, we remove the first attribute and we introduce Ceph-specific fields for the modified data blocks and related offsets, along with the respective inode number, version, size, access permissions and times of different types. These fields are necessary for the replay of the journaled updates during crash recovery.

7.2.4 Journal Commit

Normally, JBD associates each journal block buffer with (i) an extra buffer head which specifies the respective block number in the journal, and (ii) a new journal head which

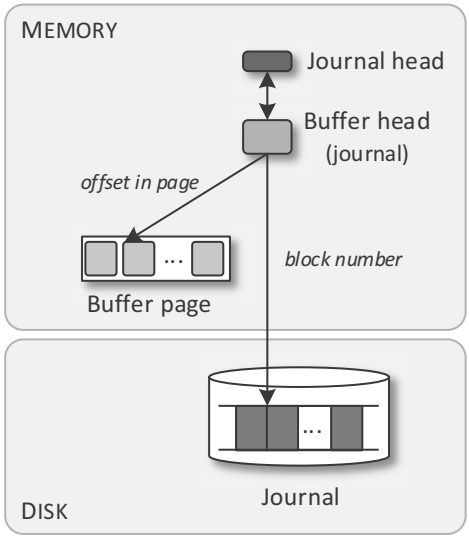


Figure 7.6: Ceph uses a single buffer head for a given journal block buffer.

keeps the corresponding bookkeeping information. In our design, this supplementary buffer head is unnecessary since our initial buffer head has no corresponding disk block. Therefore, we simply use the initial buffer head for the I/O needs of journaling (Figure 7.6). Similarly, there is no need to allocate any additional journal head. This decision results in several modifications during the commit process.

In our design, the commit procedure is initiated when either the commit interval expires, a maximum number of journal buffers threshold is reached, or some updates need to be synchronously written to disk. When the transaction moves to commit state, the kernel acquires a journal descriptor block. The descriptor block contains tags that map the journal block buffers to the corresponding metadata information. We associate every block buffer that should be journalled with the next available journal block. Then, we copy the inode-related information from the corresponding metapage to the current descriptor tag. We also allocate a new tag when a block buffer belongs to a different inode. Thus, we accumulate multiple block buffer updates of the same inode into a single tag. Every time a block buffer of the same inode is met, we increase the number of the involved data blocks in the tag, and re-calculate the offset fields properly, as well as the inode properties inside the tag (i.e., access and modification times, inode size etc.). When the descriptor block is filled with tags, we move it to the journal along with the involved block buffers, followed by a commit block. The transaction is eventually inserted in the

journal list of transactions that need to be checkpointed.

7.2.5 Journal Recovery

Client-side journaling allows the client to recover its recent state after a crash. During the recovery process of the client journal, the filesystem scans the journal in chronological order for complete transactions, and applies them by contacting the proper OSDs. At mount time, the Ceph client first verifies that there are no log records in the journal after a crash, otherwise it initiates a recovery procedure by invoking the function `ceph_journal_recover()`. In order to apply the updates from the journal, the client (i) checks whether the inode has been accessed after the crash; (ii) obtains the proper capabilities from the MDS; (iii) sends the updates to the OSDs; and (iv) modifies the actual inode.

In particular, for a given inode we initially read the corresponding metadata information from the journal descriptor block during the replay phase. According to the respective field of the tag, we also read from the journal the appropriate number of the following data blocks. After reading all the involved pages, we call the `ceph_recover_inode_pages()` function. In this way, we locate the respective inode and contact the MDS in order to get the latest inode attributes. We skip write requests that refer to an older inode version than the one that the server holds after a crash, assuming that during the client failure the write capability expired and was granted to another client. Specifically, we use the file modification time (`mtime`), the file access time (`atime`), and the inode change time (`ctime`) as an indication that the file has been accessed during the client downtime of the client. Hence we avoid to recover obsoleted writes.

If the file has not been modified after the crash, we contact the MDS to obtain the capability for writing and buffering in order to recover the inode along with its corresponding data pages. Afterwards, based on the starting and the ending offset fields from the descriptor tag, we send the data pages to the proper OSDs with a single synchronous request invoking the `ceph_recover_osd_pages()` function (Algorithm 7.2). When the update is acknowledged we (i) update the inode accordingly, (ii) mark the corresponding capabilities as dirty, (iii) inform the MDS, and (iv) release the capabilities and the inode. The above procedure is repeated for every inode that resides in the journal descriptor

Algorithm 7.1 Recover data and metadata corresponding to a specific journal tag

1: **function** CEPH_RECOVER_INODE_PAGES($journal\_tag, fs\_client, pages$)
2: Extract inode attributes from the $journal\_tag$
3: Locate the proper inode object
4: Contact the MDS for the latest inode attributes
5: **if** $mtime_{jrn} < mtime_{MDS}$ OR $ctime_{jrn} < ctime_{MDS}$ OR $atime_{jrn} < atime_{MDS}$ **then**
6: **return** 0 ▷ The inode has been modified since the crash occured
7: **end if**
8: Get the required capabilities for writing and buffering
9: **if** we received the required capabilities **then**
10: CEPH_RECOVER_OSD_PAGES($cephfs\_client, inode, endoff, startoff, pages, cnt$)
11: Update the inode attributes according to the $journal\_tag$
12: **else** ▷ Missing the capabilities required to recover
13: **return** 0
14: **end if**
15: Mark the capabilities dirty
16: Inform the MDS about the updated metadata
17: **return** 1
18: **end function**

Algorithm 7.2 Recover the OSD pages for a specific inode

1: **function** CEPH_RECOVER_OSD_PAGES($fs\_client, inode, endoff, startoff, pages, cnt$)
2: Prepare a new OSD request for the *inode*'s *pages* according to the input offsets
3: Register and send the OSD request
4: Wait for *pages* to reach the OSD disk
5: Release *pages*
6: **return** *cnt*
7: **end function**

tags. The above procedure is described in Algorithm 7.1.

If the system crashes again before the recovery finishes, the same journal records can be reused in order to complete the recovery. The journalled data blocks are replayed

normally, unless the corresponding metadata information was updated during the previous replay phase. In this case, the filesystem already holds a consistent view of the journalled data since the previous recovery procedure. Finally, when the replay phase is completed, the system can safely invalidate the recovered journal entries.

7.2.6 Journal Checkpoint

The limited amount of space in the journal leads to the need for efficient space reclamation. Besides, committed transactions that have all their blocks written to the final on-disk location, no longer need to be kept in the journal. *Checkpointing* is the process of ensuring that a section of the log is fully committed to disk, so that this area can be reclaimed.

The checkpointing process flushes the metadata and data buffers of a journal transaction not yet written to their final location on disk, allowing the transaction to be safely removed from the journal. Checkpointing is initiated when either the journal is being flushed to the disk (e.g., `umount`), or a new handle is started and the required number of buffers is not guaranteed. Especially, a checkpoint process is triggered when the amount of free journal space is between 1/4 and 1/2 of the journal size.

In our prototype, checkpointing begins with the first chronologically transaction on the list of transactions that need to be checkpointed, and synchronously sends its modified buffers to the remote OSDs. Finally, it removes the corresponding journal entries and updates the journal tail properly.

7.2.7 Flushing Dirty Data to Disk

In Table 7.1 we summarize the alternative types of disk I/O and the respective data destination in case of the original Ceph and the Arion systems. In particular, in both systems the updates are initially buffered in main memory until they are finally written back to the servers due to a timeout expiration, memory pressure, or a write-buffering capability revocation. Additionally, in the original Ceph design, a synchronous flush request forces data to the servers. Instead, upon a synchronous flush request, the Arion client writes dirty data to the local journal by forcing a journal commit. Similarly, a journal commit is initiated to persist dirty data to the local journal due to a timeout expiration, or when a maximum number of journal buffers threshold is reached. During

Type of I/O	Destination	
	Arion	**Ceph**
Writeback Timeout expiration `dirty_expire_centisecs` (30s) `dirty_writeback_centisecs` (5s) Memory pressure `dirty_background_ratio` (10%) `dirty_ratio` (20%) Revocation of write buffering capability	servers	servers
Explicit flush request `fsync`	client journal	servers
Journal commit Timeout expiration (1s) Max buffers threshold (journal_size/4) `unmount`	client journal	–
Journal recovery Replay journaled data	servers	–
Journal checkpoint Unavailable journal space `unmount`	servers	–

Table 7.1: Types of disk I/O and the respective data destination in case of Arion and Ceph. For different types of disk I/O, Arion achieves data durability by directing the I/O traffic either to the client-side journal, or to the filesystem servers. Instead Ceph transfers the corresponding data over the network to the servers. The parentheses include the default parameter values.

normal operation, checkpointing also flushes journaled data to the servers in order to reclaim the journal space. In case of a client failure, Arion replays journaled data to the proper servers during recovery. Finally, when the Arion client unmounts the filesystem, a commit and a checkpoint process are typically initiated to flush dirty data to the servers.

7.3 Summary

We increase the statefulness of the client in a large-scale object-based filesystem, by incorporating the Linux JBD layer into the CephFS kernel-based filesystem client of Ceph. The Arion prototype provides the following functionality: (i) the commit of updated data and metadata from the memory of the client to the local journal, (ii) the filesystem recovery in case of a client failure using a properly designed conflict resolution approach, and (iii) the checkpointing of journaled data for journal space reclamation.

Chapter 8

Performance Evaluation of the Arion System

In this chapter, we describe our experimentation environment, the measured performance and resource consumption of Arion and original Ceph over a local cluster and a large-scale setup. Moreover, we evaluate the scalability of our system on top of a large-scale public cloud environment consisting of up to 114 filesystem server and client nodes.

We study the performance and resource efficiency of microbenchmarks and application-level workloads (e.g., mail server, OLTP, desktop workloads). We also investigate the performance of the database logging activity with real traces, or by directly running a memory-based NoSQL store on top of the shared storage system. Additionally, over a multi-tier configuration we examine the storage layer of a commonly-used key-value store. Finally, we measure the time needed to recover the system to a consistent state after a crash. Given the endurance and performance characteristics of novel devices such as solid-state drives based on flash memory, we also evaluate our prototype over an alternative storage setup based on SSDs.

8.1 Experimentation Environment

Local Cluster. In the local cluster setup, the host machine is a rack server with 2 quad-core x86 2.66GHz processors, 7GB RAM, 2 bonded 1GbE links, and two 300GB 15KRPM SAS HDs in RAID0 configuration. The host uses Linux kernel v3.14.14 with Xen v4.4.0 for virtualization, and the guest runs Linux v3.6.6 over 2GB RAM and 2 pinned VCPUs. Arion uses a 2GB disk partition at the host for local journal. The guest client mounts directly the distributed filesystem, and the hypervisor provides local access to the network and journal devices.

Each of Ceph and Arion uses 5 nodes consisting of 3 OSDs, 1 MON and 1 MDS (Ceph v0.80.1). The nodes are rack-based servers, each with 2 quad-core x86 2.66GHz processors, 3GB RAM, 1 GbE link, and two 300GB 15KRPM SAS HDs used separately. The servers run Linux kernel v3.10.41. We keep the replication level of the OSDs at the default value of 3. Every OSD dedicates one disk for journaling (1 GB partition).

We clear the caches before each experiment. At the host the write buffers of the hard disks are *disabled*, but on the servers of Ceph and Arion all the disks have their write buffers activated [152]. The use of RAID0 with two disks does not give unfair advantage to host journaling because the storage backend already consists of multiple servers with two disks each. In the shown graphs we include 95% confidence intervals from 5 repetitions, unless specified otherwise.

Public Cloud. Additionally, we examine the scalability of the proposed system in a public cloud environment. Our testbed consists of EC2 instances from the US East region of the Amazon Web Services (AWS). We use up to 114 instances of types `m1.large` (HDD-based) or `c3.large` (SSD-based) as fileservers and filesystem clients, as described in Table 8.1. All instances run Debian8.

In particular, each of Ceph and Arion uses up to 114 nodes consisting of a varying number of OSDs and clients, 1 MON and 1 MDS (Ceph v0.94.2). The servers run Linux kernel v3.10.41. We keep the replication level of the OSDs at the default value of 3. Every OSD dedicates a 5GB disk partition for journaling. Each client mounts directly the distributed filesystem. At the Arion clients we set the two storage devices in RAID0 configuration, and use a 5GB disk partition for local journal.

AMI	vCPU	RAM(GiB)	Storage	Network
HDD-based Setup				
m1.large	2	7.5	2x420GB	Moderate
SSD-based Setup				
c3.large	2	3.75	2x16GB	Moderate

Table 8.1: Amazon Web Services experimentation environment.

In case of the original Ceph we keep the default timeout intervals, whereas we extend the writeback and expiration intervals of Arion to 120 seconds. Similarly to the previous experimental setup, we set the commit interval of Arion equal to 1 second. As previously, we clear the caches before each experiment.

8.2 Performance Evaluation

In this section, we present the extensive experimental evaluation of the Arion prototype through microbenchmarks, application-level workloads, and real-world applications across alternative local and large-scale storage setups. We also measure the time needed to recover the system to a consistent state after a crash.

8.2.1 Filebench

Our first set of experiments is based on the Filebench v1.4.9.1 macrobenchmark (fileserver, varmail, createfiles). In Figure 8.1 we use the default settings of two Filebench modes to compare the performance and efficiency of Ceph and Arion in the local cluster setup. We examine Ceph with the writeback and expiration time respectively set to the default 5s and 30s (Ceph), or both set equal to 1s (Ceph-1), or the filesystem mounted in synchronous mode (Ceph-sync). We also examine Arion with dirty blocks periodically copied to the host-side journal every 1s, and the writeback and expiration times both set equal to 60s (Arion-60) or infinity (Arion-inf) to minimize writeback.

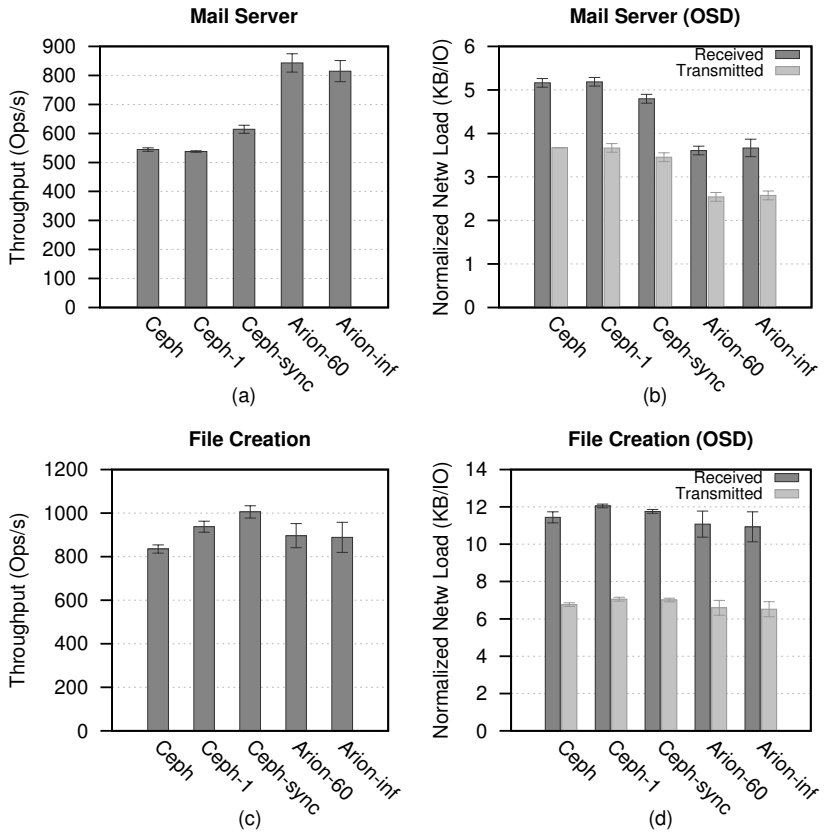


Figure 8.1: Operation throughput and normalized network load with the varmail (a,b) and createfiles (c,d) modes of Filebench across different settings of Ceph and Arion.

Mail Server. Varmail emulates multi-threaded I/O activity of a server *synchronously* storing email messages across 50000 files. In Figure 8.1a, Arion-60 achieves operation throughput of 842.8 operations/second, which is 55% higher than the 544.6 operations/second of the default Ceph. Also, Arion-60 increases the data throughput of Ceph (2MB/s) by 50% and reduces the average latency of Ceph (95ms) by 38%. Ceph-1 has performance similar to that of Ceph. In Figure 8.1b, we show the received and transmitted OSD network traffic normalized by the number of completed operations during the experiment. Arion-60 reduces the received network load of Ceph —normalized in KB/IO— by 30% and the transmitted by 31%. In the above experiments, the server disk I/O is the bottleneck due to the synchronous writes.

Metadata-intensive Workload. We examine a metadata-intensive workload with file creations in Figures 8.1c,d. It is interesting that Ceph-sync manages to improve the per-

formance of Arion-60 by 12% but also increases the network load by 6.2% in the received and transmitted cases. With respect to the default Ceph, Arion-60 has comparable performance and load. Ceph-sync has slightly better performance probably because it handles the metadata updates directly at the MDS. Thus, Ceph-sync avoids the extra load of transferring metadata to the client required by the asynchronous settings.

8.2.2 Microbenchmarks

In the local setup we further explore the relative behavior of the two systems using the FIO v2.1.7 microbenchmark assuming Zipfian write pattern with α=1.0001 (e.g., [93]). Accordingly, there is a high percentage of overwrites: ∼66.7% of the write requests refer to overlapping file offsets. The benchmark asynchronously writes a total of 2GB data in a preallocated file of 2GB size with block size in the range 2-16KB. In the following experiments we examine the performance and the resource efficiency of Ceph and Arion in the local cluster setup, and also evaluate the scalability of the two systems in a large-scale public cloud environment.

Local Cluster Setup. From Figure 8.2a, in comparison to Ceph (1ms) and Ceph-1 (1.5ms), Arion-60 achieves lower latency (0.6ms) by 40% and 53%, respectively. In Figure 8.2b, we examine the total network traffic received over time at one OSD of each system. We notice that Ceph terminates at instance 249 with 2.2GB total received traffic. In contrast, Arion-60 ends the experiment at 159s (36% shorter) with received volume 1.3GB (41% lower).

In Figures 8.2c,d we examine the bandwidth utilization of the journal and filesystem storage device at one of the OSDs. In particular, we depict the percentage of CPU time during which I/O requests were issued to the device, according to the `iostat` monitoring tool. We show Ceph-1 that keeps the durability characteristics similar to those of Arion. The depicted Arion-60 OSD utilizes the journal and filesystem device at 22.3% and 20.7% on average; the respective utilizations of Ceph-1 are 21.4% and 88.2%. We conclude that Arion-60 reduces the filesystem device utilization by 76.5% with respect to Ceph-1 in the examined case.

In the same experiment we also evaluate the performance of Arion with alternative writeback timeouts. In Figures 8.3a-c we examine Arion with dirty blocks periodically

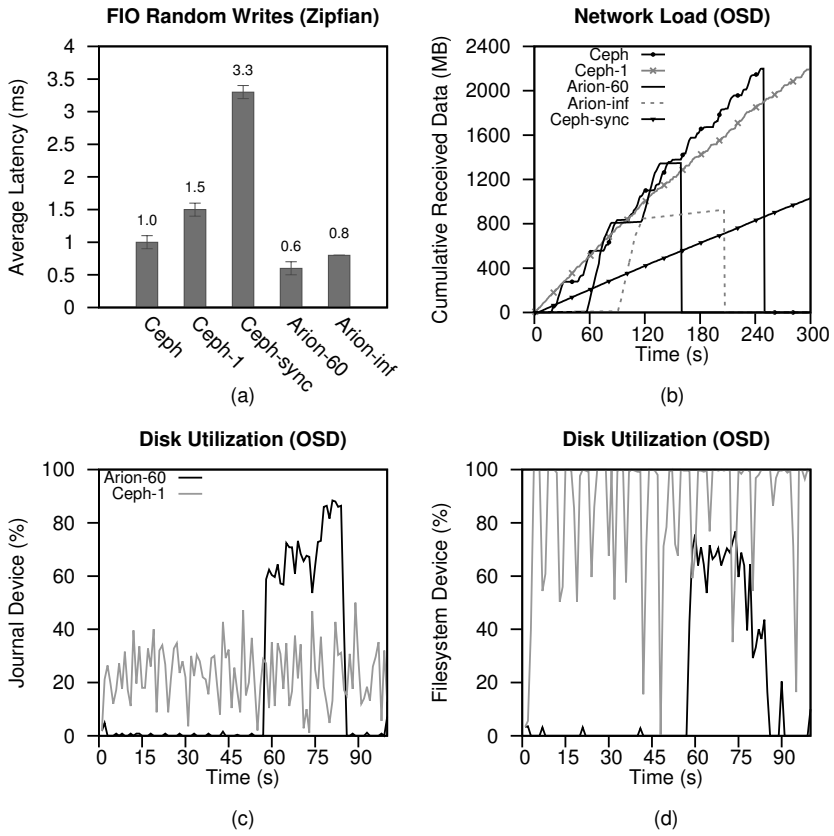
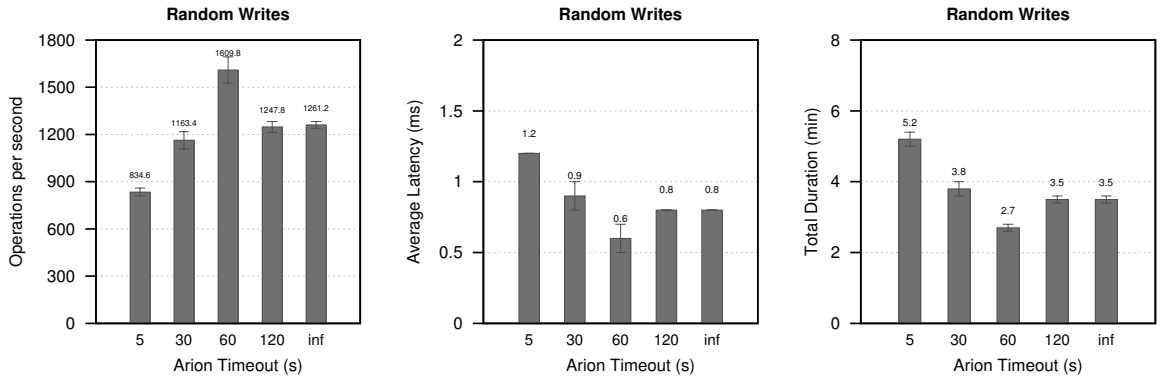


Figure 8.2: Average latency (a), cumulative network load at OSD (b), and disk utilization at the journal (c) and filesystem (d) OSD disks across different settings of Ceph and Arion.

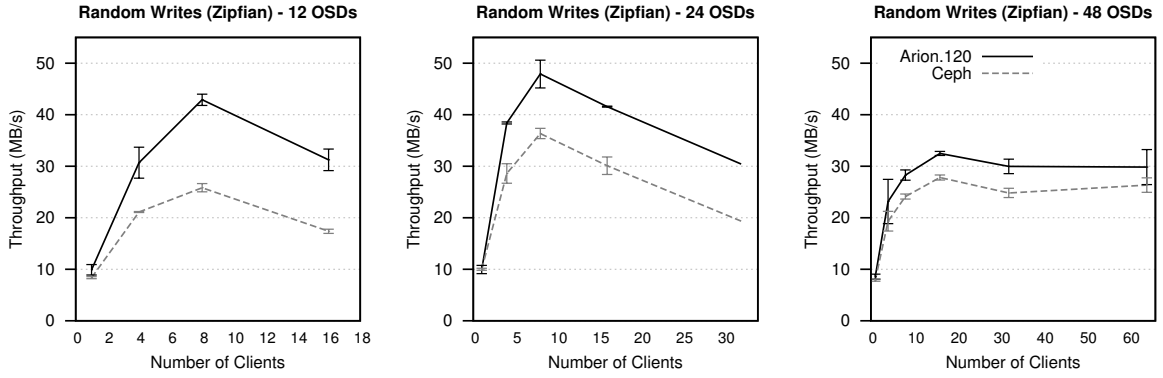
copied to the host-side journal every 1s, and the writeback and expiration times set equal to 5s, 30s, 60s, 120s or infinity to minimize writeback. We observe that Arion's performance depends highly on the frequency of the writeback process. In particular, the high network and disk load at the server-side under short-time intervals (e.g., 5s and 30s) results in lower write throughput and higher latency. Instead, an extended interval (e.g, 120s and infinity) can lead to the client memory or journal space pressure, which eventually degrades the overall system performance by forcing dirty data to the servers. Hence, in the following experiments we set the writeback timeout to 60 seconds as the appropriate time interval.

Public Cloud Setup. We repeat the same experiment in the public cloud environment. We evaluate three alternative configurations. Initially, we use 30 nodes consisting of 12



(a) (b) (c)

Figure 8.3: Arion performance under different writeback timeouts.



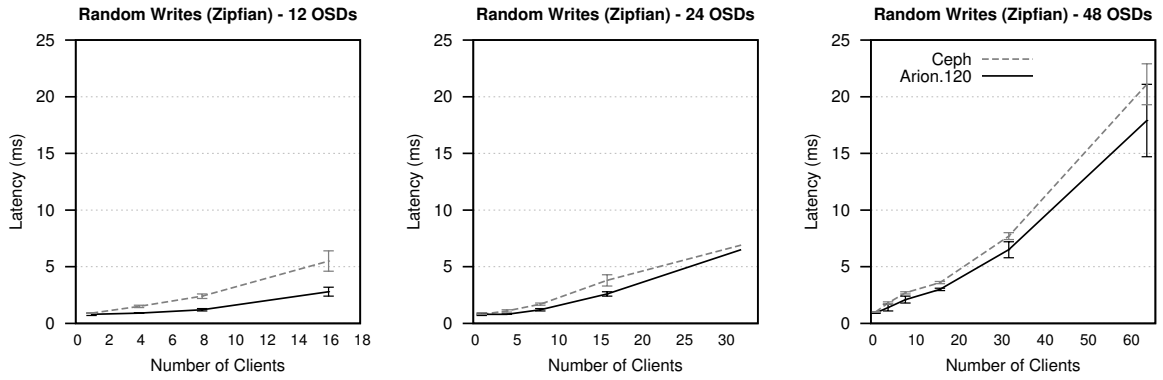
(a) (b) (c)

Figure 8.4: FIO random writes throughput.

OSDs, up to 16 clients, 1 MDS and 1 MON. In the second setup we have 24 OSDs, up to 32 clients, 1 MDS and 1 MON, and lastly we have a total of 114 machines consisting of 48 OSDs, up to 64 clients, 1 MDS and 1 MON. In the following graphs we include 95% confidence intervals from 3 repetitions[1].

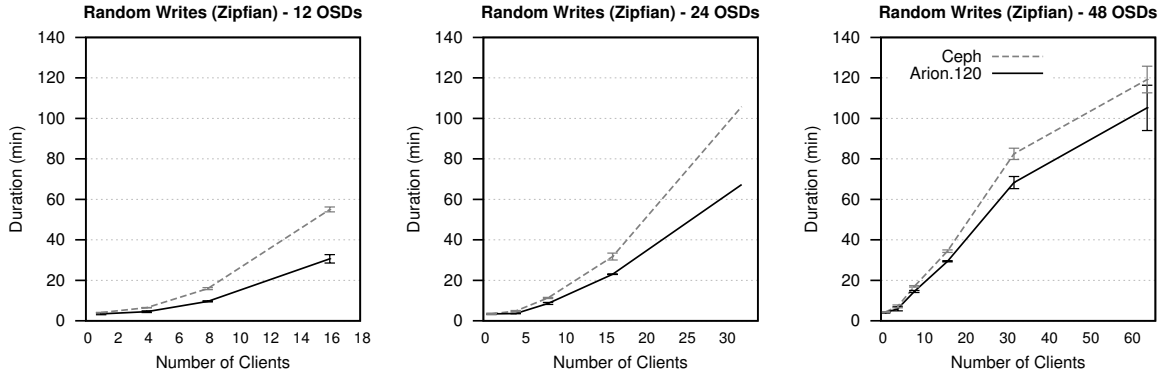
In Figures 8.4a-c we measure the rate of the total amount of written data over the average experiment duration across the clients. Arion improves the throughput of the original Ceph by up to 80% with 12 OSDs and 16 clients, 57% with 24 OSDs and 32 clients,

[1]In the experiment of the 24 OSDs with 32 clients we only show the results of a single run due to a budget limitation.



(a) (b) (c)

Figure 8.5: FIO average request latency.



(a) (b) (c)

Figure 8.6: FIO average experiment duration.

and 21% with 48 OSDs and 32 clients. Similarly, Arion reduces the average latency by up to 50%, 32% and 22% with 12, 24 and 48 OSDs respectively (Figures 8.5a-c). Figures 8.6a-c present the average duration of the experiment under different configurations. In all cases, Arion takes shorter time to complete the I/O requests of the experiment, improving the average duration up to 44.4% (12 OSDs and 16 clients).

Next, in Figures 8.7 we depict the device utilization of the filesystem and the journal disks at one OSD over time, according to the `iostat` monitoring tool. We present a specific time window starting at the beginning of the experiment for different number of OSDs and 16 filesystem clients. It is interesting that the device utilization across the

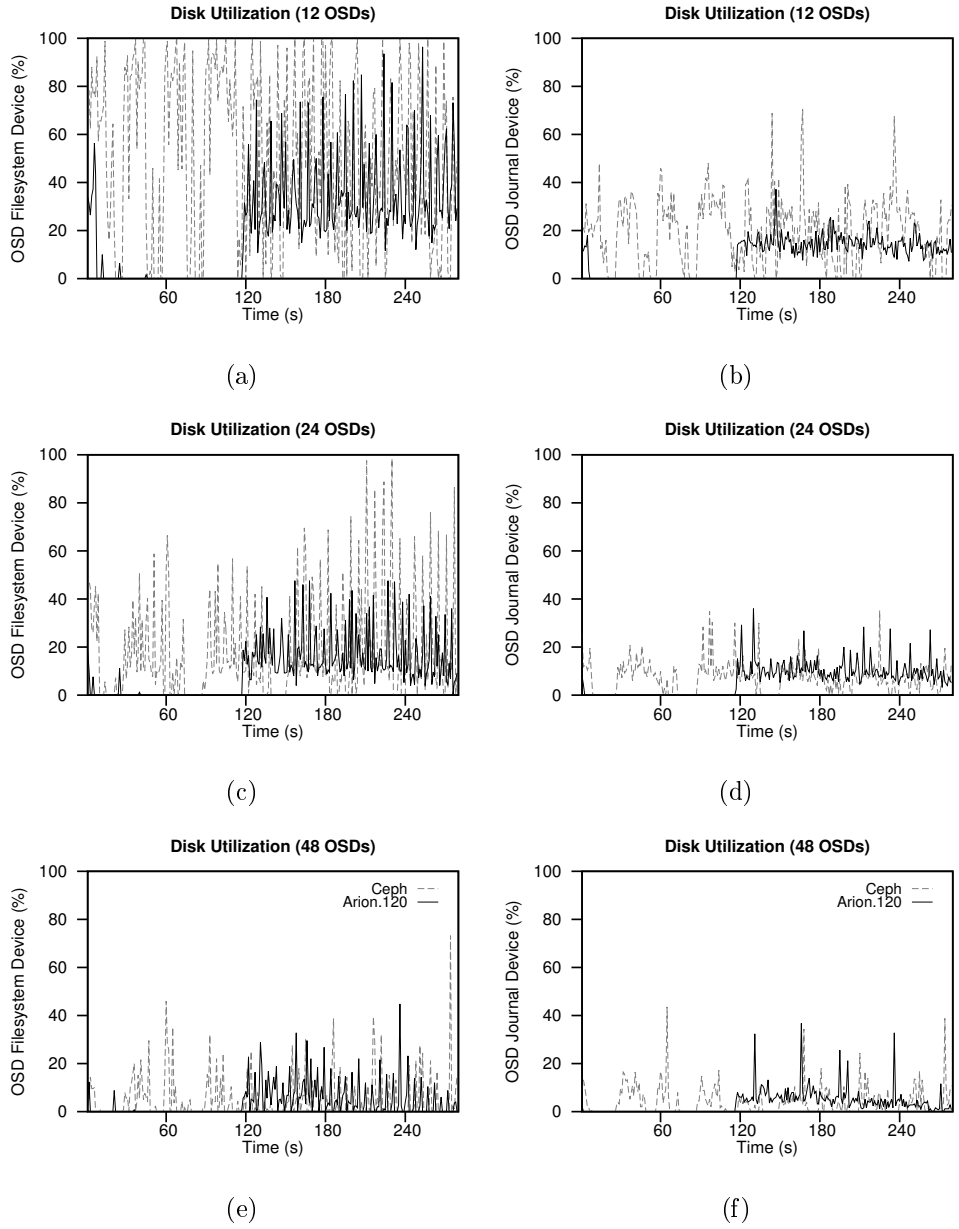


Figure 8.7: FIO device utilization over time with 16 clients for (a,c,e) the filesystem and (b,d,f) the journal disks at one OSD.

two systems remains comparable, but the device utilization per OSD drops as the system scales out.

Overall, we observe that under alternative configurations, Arion consistently improves the performance of the original Ceph by taking advantage of the local journal device throughput at the clients.

8.2.3 Databases

In order to investigate the performance, resource efficiency and scalability of Arion with synchronous database workloads, we run the MySQL OLTP benchmark from Sysbench v0.4.12 [175]. We use MySQL with the default InnoDB storage engine configured for high durability (i.e., the log is flushed to disk at each transaction commit). We run a varying number of virtualized MySQL servers over shared storage. More specifically, we store both the database and log files over Ceph and Arion, in the local cluster and public cloud setups. In the following experiments, we also evaluate Arion over an alternative storage setup based on solid-state drives which are attractive due to their intrinsic characteristics.

Local Cluster Setup. In the local cluster setup, we run up to two virtualized MySQL servers (v5.1) with shared storage, and generate requests on another node with 3GB RAM and CPU 8x2.66GHz. We examine the non-transaction mode of Sysbench, which consists of insert or update-key requests. Each table contains 10000 rows and we use 1 and 10 threads to issue a total of 20000 requests per server.

In Figures 8.8a-d we measure the average number of insert and update-key operations per second with one and two MySQL instances running on top of different host machines. In the first set of experiments (Figures 8.8a,b) we examine the update-key operations. In case of a single MySQL instance, Arion improves the throughput of Ceph by 4.9 times for a single thread, and by 2.3 times for 10 threads (Figure 8.8a). In Figure 8.8b we evaluate the performance scalability of Arion and Ceph using two concurrent MySQL instances. We show that with two instances Arion achieves aggregate operation throughput improved by 83.1% and 39.8% for 1 and 10 threads respectively. On the contrary, the performance of Ceph doesn't scale at all, since the journal device of the servers becomes the resource bottleneck, with an average disk utilization of ∼85%. Indeed, Arion increases the operation throughput of Ceph by 8.2 times for 1 thread and by 3.2 times for 10 threads. The same observations also apply to the operation throughput of the insert requests (Figures 8.8c,d).

Overall, Arion manages to improve up to 8 times the operation throughput of OLTP workloads in a local setup with respect to Ceph. In particular, Arion services the synchronous write requests locally through the host-side journal, reducing the I/O traffic directed to the remote servers, thus allowing higher performance scalability.

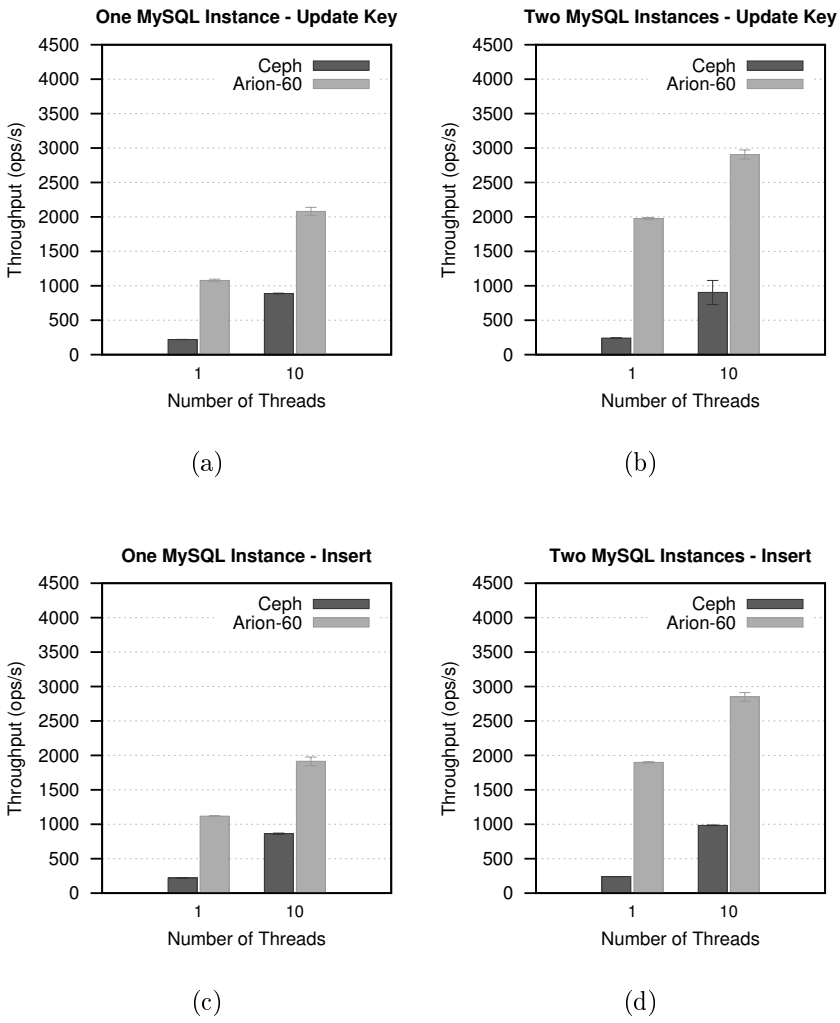
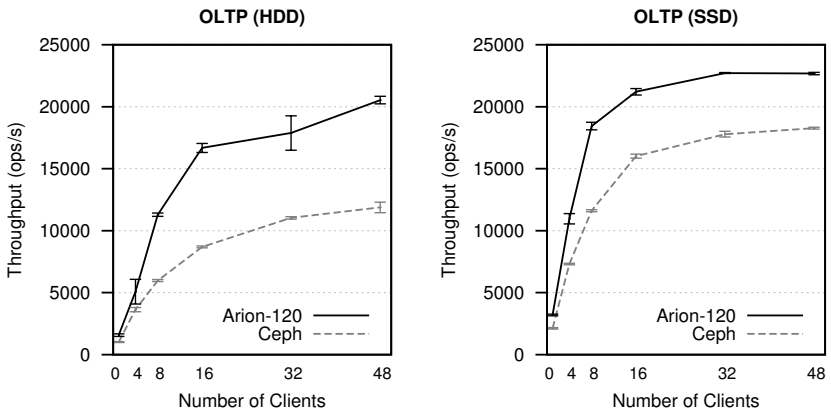


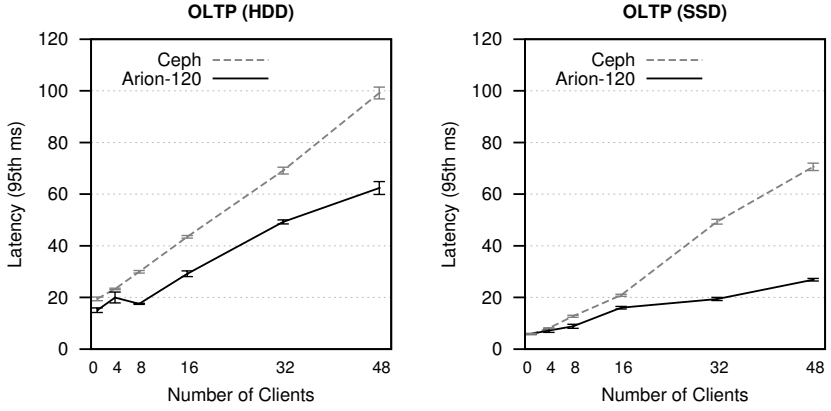
Figure 8.8: (a,b) Update-key and (c,d) insert operations per second in a local cluster setup.

Public Cloud Setup. We further investigate the performance scalability of Arion in case of a synchronous database workload over a large-scale cloud setup. We run a separate MySQL server (v5.5) on each filesystem client, and generate requests from a different node with 1GB RAM and 1 vCPU. Each table contains 100000 rows, and we have 10 threads issuing a total of 100000 update-key requests per server. In the following experiments, we examine two deployments with 12 OSD servers and 1-48 Ceph or Arion clients. The first setup is based on HDD and consists of m1.large instances; the second one is based on SSD and consists of c3.large instances (Table 8.1). In the shown graphs we include 95% confidence intervals from 3 repetitions.



(a) (b)

Figure 8.9: Operations throughput of update key requests in a public cloud setup based on (a) HDD and (b) SSD.



(a) (b)

Figure 8.10: 99th percentile latency of update key requests in a public cloud setup based on (a) HDD and (b) SSD.

In Figures 8.9a,b we measure the rate of the total number of completed operations at the servers over the average duration of the experiment across the clients. At the first setup based on HDD, Arion improves the operations throughput of Ceph from 40% with 8 filesystem clients up to 92% with 16 clients (Figure 8.9a). Similarly, in the setup based on SSD, Arion achieves up to 59% improved throughput in comparison to Ceph in case of 8 clients (Figure 8.9). Figures 8.10a,b depict the 99th percentile latency of the Arion and

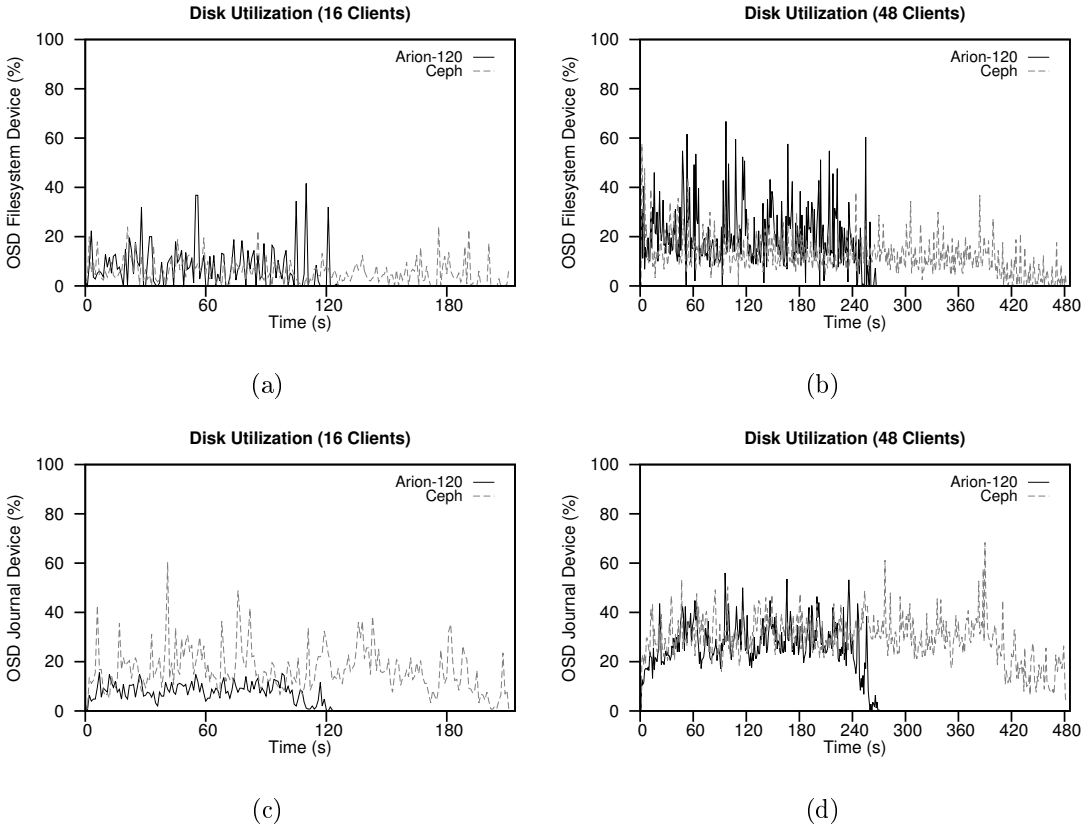


Figure 8.11: HDD device utilization over time with 16 and 48 concurrent clients for (a,b) the filesystem and (c,d) the journal device at one OSD.

Ceph systems. Arion reduces the request latency of Ceph by up to 42% with 8 clients in the setup based on HDD (Figure 8.10a), and up to 62% with 48 clients in the setup based on SSD (Figure 8.10b).

In Figure 8.11 we examine the HDD device utilization of the filesystem and the journal disks at one OSD during the duration of the experiment according to the `iostat` monitoring tool. In both systems, the OSD devices' utilization is comparable. Notably, we observe that the same experiment takes shorter time to complete in case of the Arion system. In Figure 8.12 we show the SSD device utilization of the filesystem and the journal disks at one OSD over time by extending the time window by a few seconds after the completion of the experiment. Likewise the setup based on HDD, both Arion and Ceph have similar device utilization, while Arion additionally reduces the duration of the experiment.

In general, the above experimental results validate our previous observations about

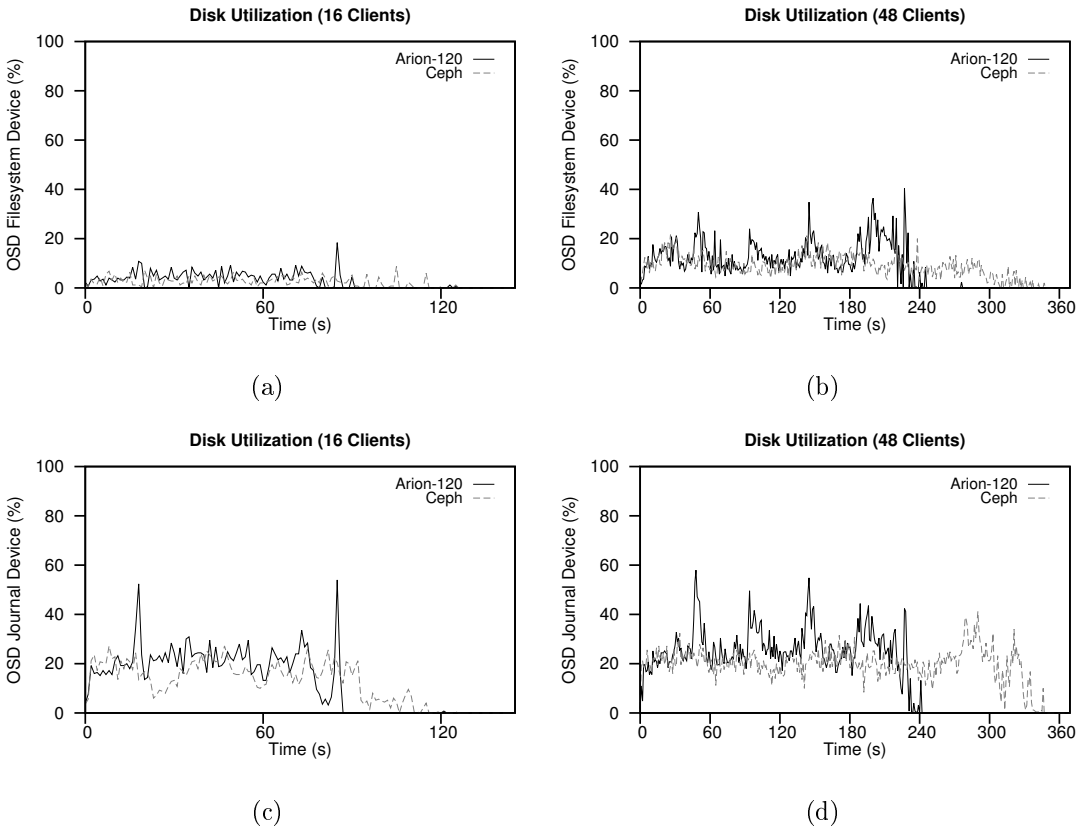


Figure 8.12: SSD device utilization over time with 16 and 48 concurrent clients for (a,b) the filesystem and (c,d) the journal device at one OSD.

the performance improvement, resource efficiency, and scalability of Arion in comparison to Ceph over a large-scale public cloud environment. Additionally, we observe that both systems benefit from the performance characteristics of solid state disk drives, further resulting in higher scalability. Interestingly, the relative behavior of Arion with respect to the original Ceph is comparable over the alternative storage device setups.

8.2.4 Groupware and Database Logging

In the following set of experiments we evaluate the ability of Arion to efficiently serve the synchronous I/O traffic of groupware and database logging activity. In particular, we examine the logging activity of an email server and a well-known memory-based key-value store.

	Latency (99th ms)	
Mode	**Single VM**	**Two VMs**
Ceph/osync	4.11	4.37
Ceph/fsync	4.07	4.59
Arion-60/fsync	1.98	2.11
Arion-60/nosync	0.03	0.04

Table 8.2: The 99th percentile latencies across different log flushing configurations.

Jetstress. We consider the Jetstress Tool that emulates the disk I/O load of the Microsoft Exchange messaging and collaboration server [92]. The experimental setup is the same with the one described in Section 5.2.3. We use the original interarrival times to replay a 15min extract from the middle of the log trace on top of a one and two virtualized filesystem clients. Each VM runs over a separate host machine. In Table 8.2 we present the 99th percentile latencies for the log writes to be flushed to stable medium across 3 repetitions. We examine four alternative log flushing configurations. Initially, both Ceph (Ceph/fsync) and Arion (Arion-60/fsync) have each log write followed by a disk flush request. We also consider Ceph with the O_SYNC option at file open to immediately force log writes to the remote storage (Ceph/osync). Finally, we also present the baseline case of asynchronous write requests where Arion periodically persists dirty blocks to the host-side journal every 1s (Arion-60/nosync). We assume that the durability of synchronous writes is similar to that of bypassing the page cache.

In case of a single VM, Arion-60/fsync reduces the 99th percentile write latency by 51.8% and 51.4% with respect to Ceph/fsync and Ceph/osync. Similarly, the latency is reduced by up to 54% when we have two concurrently-running instances. Therefore, Arion manages to reduce the write latency of the logging activity by persisting the synchronous writes to the client-side journal without penalizing the disk and network bandwidth of the servers.

Redis NoSQL Store. The role of DRAM in storage systems has been strengthened over the last years in order to meet the needs of latency-sensitive large-scale web applications. As a result, several memory-based key-value stores have been developed recently, such as

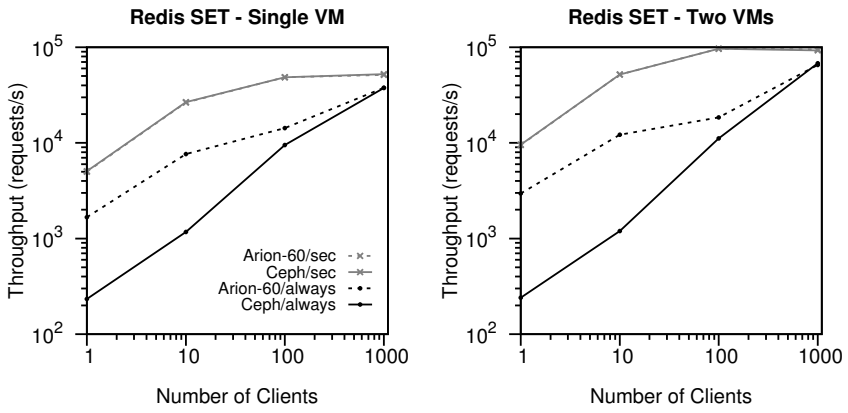


Figure 8.13: Average operation throughput of Redis NoSQL store under different synchronization policies.

RAMCloud [134] and Redis [153]. These systems typically use a persistence log at the server for durability. In fact, the log can be replicated across multiple storage servers for high availability in case of hardware failures. For instance, RAMCloud keeps a single copy of data in DRAM, and stores the redundant copies on disk or flash using striping. In case of Redis, availability can be provided by storing the log over a replicated network filesystem.

In this experiment, we examine the logging activity of Redis; a memory-based key-value store which persists each write to an append-only operational log [153]. Traditionally the log is stored at the local filesystem of the server. Instead, a replicated network filesystem can be used for reasons of reliability and manageability. In our experimental setup, we evaluate Redis on top of a shared storage system by storing the persistence log over Ceph or Arion. There are three different synchronization policies: (i) fsync/never allows the operating system to flush dirty data to the append-only log periodically according to the pdflush kernel daemons, (ii) fsync/always synchronizes every write to the log with fsync(), and (iii) fsync/everysec synchronizes dirty data to the log once per second. The second option provides the highest level of durability since it ensures that dirty data will be stable before an acknowledge is returned to the client. However, for performance reasons, fsync/everysec is the default option.

We use the benchmarking tool distributed with Redis v2.8.17 and configure it to execute SET requests to a range of 1000000 random keys with the default value size of 3

bytes and concurrency of 1,10, 100 and 1000 connections from a single benchmark client with 3GB RAM and CPU 8x2.66GHz. We run a Redis server on a separate node at the side of the filesystem client. In Figures 8.13a,b we present the average transaction throughput of SET operations for Ceph and Arion under the second and the third synchronization policies.

In the case of a single virtualized Redis server, Arion/always increases the average operation throughput of Ceph/always up to 7 times, from 233 operations/second to 1669 operations/second for a single connection. Similarly, Arion/always improves the throughput of Ceph/always by an order of magnitude in the case of two separate virtualized Redis servers. We observe that the performance gain decreases as we increase the number of connections. Especially, the write request sizes range from several tens of bytes for a single connection, to hundreds of kilobytes for 1000 connections. Thus, fewer connections result in many small synchronous write requests which can benefit from the sequential disk throughput of the local journal device. In the same figures we also measure the performance of the fsync/everysec synchronization policy, which results in significant operation throughput improvement in both systems, with respect to fsync/always due to batching multiple write requests into memory before flushing them to stable media. As a result, the operation throughput of Arion is comparable to that of Ceph.

The above results strengthen our previous observations regarding the synchronous I/O workloads. Overall, Arion improves the operation throughput and reduces the application-perceived latency with regard to Ceph by flushing synchronous write requests to the client-side journal.

8.2.5 LevelDB

In large-scale cloud environments, it is typical to have a distributed database running on top of a cloud-scale filesystem. Prominent examples include Google's Bigtable over GFS [37] and Apache HBase over HDFS [74]. In a scalable datastore, the data is dynamically partitioned across the available servers for resource efficiency and fault-tolerance. At each server, a storage layer is responsible for the memory and disk management [173]. Data is arranged on disk over a tree-based data structure which is usually stored on top of a large-scale filesystem. In this experiment, we focus on the storage layer of such a multi-

tier architecture. We evaluate the performance of an open-source key-value database library over shared storage using Ceph and Arion as the storage backend.

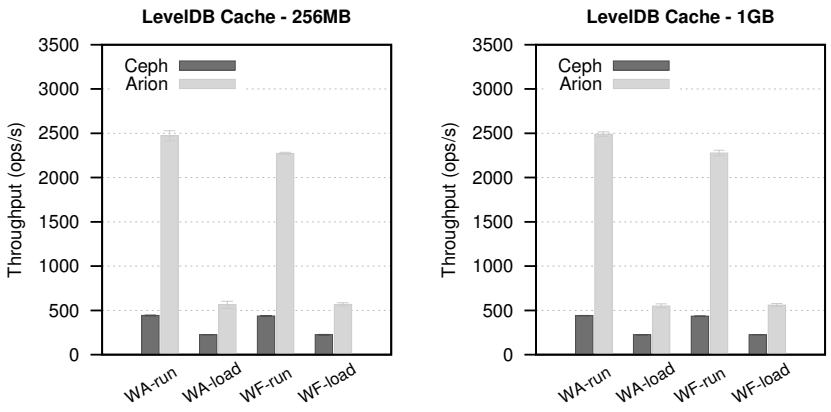
We examine the LevelDB key-value store from Google, which is layered on a variation of the Log-Structured Merge Tree (LSM) and provides a simple API with GET, PUT, SCAN and DELETE operations [53]. LevelDB initially appends an incoming update to a write-ahead log for durability, and then it inserts the update into a memory buffer, called memtable. When a memtable reaches a predefined threshold (4MB by default), the memory contents are sorted and written to disk as an SSTable. Furthermore, SSTables are organized into a series of ordered levels. By default, each write to LevelDB is asynchronous; it returns as soon as it is buffered in the page cache of the operating system.

We examine the performance of LevelDB over shared storage with the Yahoo! Cloud Serving Benchmark (YCSB)[47]. We evaluate two of the built-in YCSB workload configurations. Workload A is considered write-heavy with 50 percent reads and 50 percent updates, whereas workload F consists of 50 percent reads and 50 percent read-modify-write operations. Each YCSB experiment has a load and a run phase. We set the number of keys in the workloads to 100000 with Zipfian key access, keeping the default value size of 1KB. We configure LevelDB to synchronously write the update requests to disk before responding to the benchmarking client. In these experiments we examine LevelDB with cache sizes 256MB and 1GB.

In Figures 8.14a,b we show the operation throughput of YCSB load and run phases for Arion and Ceph under the workloads A and F. Arion improves the operation throughput of the load phase of workloads A and F with respect to Ceph by 2.5 times (from ∼225 operations/second to ∼563 operations/second), regardless of the actual cache size. Similarly, the operation throughput of the run phase of both A and F is increased by up to 5.6 times, from 441 operations/second to 2490 operations/second for workload A.

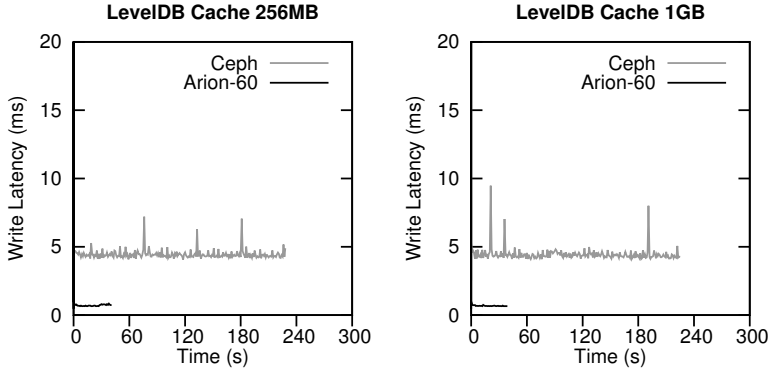
Figures 8.15a,b depict the average write latency over time for the run phase of workload A. In particular, Arion reduces the update latency of workloads A and F from 4.4ms to 0.7ms. The read latency of Arion and Ceph remains the same, around 0.1ms. Overall, in Figures 8.15a,b, Arion reduces the total duration of the run phase of workload A from 225s to 39s and from 228s to 41s, for caches 1GB and 256MB respectively.

Next, we demonstrate the resource utilization over time of workload A during the load (Figure 8.16) and the run phases (Figure 8.17). We observe that the bottleneck resource



(a) (b)

Figure 8.14: YCSB throughput for workloads A and F over LevelDB with cache sizes (a) 256MB and (b) 1GB.



(a) (b)

Figure 8.15: YCSB update latency over time with cache sizes (a) 256MB and (b) 1GB.

of Ceph is the server's disk due to the small synchronous update requests. In accordance to previous measurements, we also notice that Arion also decreases the incoming network traffic at the server-side.

In the evaluated setup, we conclude that under synchronous workloads Arion achieves higher operation throughput and lower update latency in comparison to Ceph. More importantly, the client-side journal of Arion reduces the resource consumption (disk and network) at the filesystem servers, and thus results in improved overall system performance

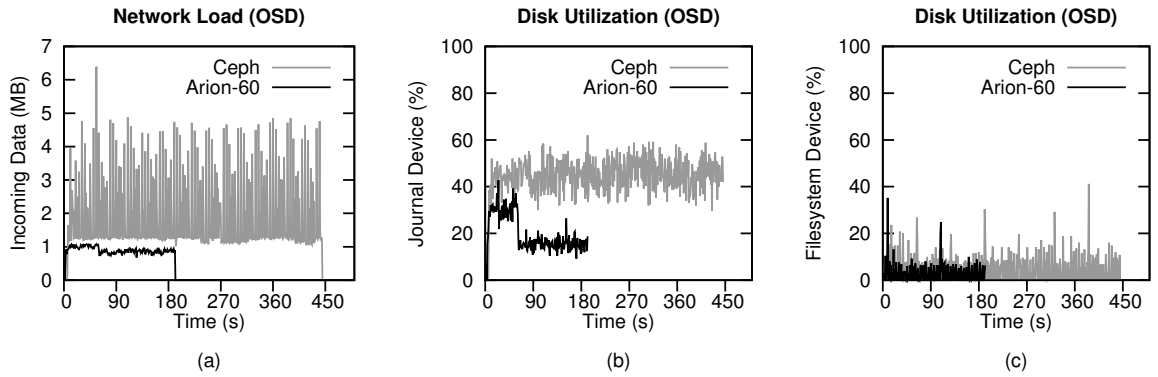


Figure 8.16: Resource utilization for workload-A load phase with 256MB cache.

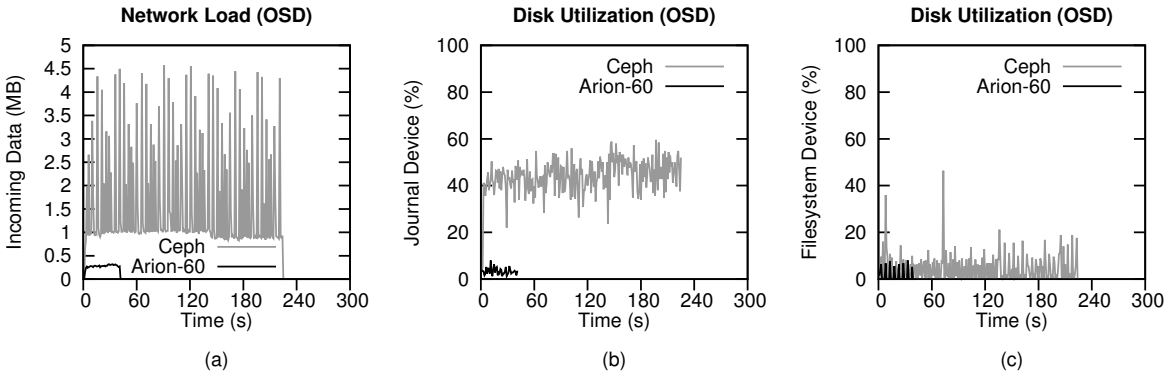


Figure 8.17: Resource utilization for workload-A run phase with cache size 256MB.

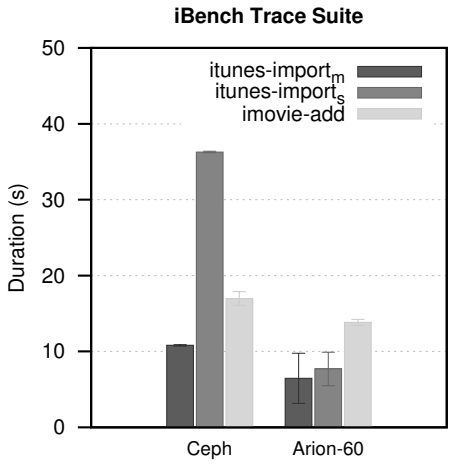
and efficient resource utilization. We also experimented (not shown) with asynchronous workloads, and noticed that the performance of Arion and Ceph is comparable, because in both systems asynchronous write operations are acknowledged to the application when they reach the memory of the client.

8.2.6 Desktop Applications

In a shared workspace environment, the home directories of collaborating users can be maintained in a shared filesystem. Typical file exchanges of unstructured data (e.g., documents, multimedia files) are enabled through shared folders in a Dropbox-like manner [56]. In this experiment, we evaluate the behavior of Arion when used as shared storage backend in desktop environments.

Workload	RD/WR Accesses	RD%	WR%	fsync-cnt
itunes-import$_m$	57	48.0	52.0	32
itunes-import$_s$	139	66.3	33.7	95
imovie-add	547	47.8	52.2	185

Table 8.3: iBench workload characteristics.

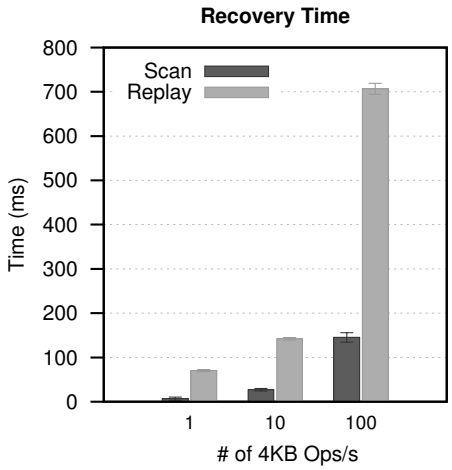


(a)

Figure 8.18: Average duration of iBench desktop workloads.

In particular, we examine several multimedia traces collected from a desktop environment and provided with the iBench trace suite [75]. We select three different iLife workloads with sufficient write-to-read ratio. We use the iTunes media player traces of importing (i) an album of ten MP3 songs (itunes-import$_s$), and (ii) a 3-minute MPEG-4 movie (itunes-import$_m$). We also evaluate the iMovie video editor's trace of adding a clip from a 3-minute MPEG-4 movie into a project (imovie-add). We replayed the traces with Magritte [194] over a single virtualized filesystem client. Table 8.3 describes the detailed characteristics of the three workloads.

Figure 8.18 depicts the total duration of the workloads for the Ceph and Arion systems. Arion reduces the runtime of itunes-import$_m$, itunes-import$_s$ and imovie-add by 41%, 79% and 19% respectively with regard to Ceph. This performance increase is due to the high frequency of `fsync()` calls observed in the above workloads, according to a



(a)

Figure 8.19: Average time to recover the completed transactions from the journal device after a client crash.

related study [75]. Furthermore, each explicit flush request typically synchronizes small amounts of data leading to small synchronous I/O write traffic to the filesystem. Overall, we demonstrate the performance advantage of Arion with respect to Ceph, in case of general-purpose desktop environments.

8.2.7 Recovery

Finally, we evaluate the ability of the system to recover quickly after a system crash, which leaves the journal with log records before the respective updates are checkpointed to the filesystem. It is known that when the free journal space lies between $\frac{1}{4}$ and $\frac{1}{2}$ of the journal size, JBD2 automatically checkpoints the updates to the final location. Therefore, we use writes that are small enough to prevent checkpointing before the crash, but also useful for some application classes, e.g., event stream processing [29].

In Figure 8.19, we have 1 thread doing 1, 10 or 100 writes per second with request size 4KB for a total duration of 30 seconds. Then we cut the power of the system. At the subsequent reboot, we verify that Arion fully and correctly recovers the unique written data, while in the kernel we measure the duration of filesystem recovery. We breakdown the total recovery across the passes that scan and replay the committed transactions. We

observe that the recovery time of the Arion client lies in the range of 77.4ms-852.2ms, depending on the load of client write activity before the crash.

8.3 Summary

We conducted our experimental evaluation over a local cluster setup and a large-scale public cloud environment consisting of up to 114 filesystem server and client nodes. We experimentally demonstrated improved performance for specific durability guarantees, and reduced network and disk bandwidth at the storage servers over a wide range of microbenchmarks, application-level workloads and multi-tier storage setups. We examined both synchronous and asynchronous I/O traffic. Especially in case of synchronous workloads, Arion reduces significantly the I/O load directed to servers by flushing write requests to the client-side journal. We observed that Arion improves the OLTP operation throughput up to a factor of 8 in a local cluster setup, and up to 92% with 12 filesystem clients and 12 OSDs in the public cloud environment, with respect to Ceph. We validated the above results with several application-level workloads, real traces of database logging activity, or by directly running a memory-based NoSQL store on top of the shared storage system in a local cluster setup. On a multi-tier configuration, Arion increases the operation throughput of a commonly-used key-value store by 5.7 times in comparison to Ceph, at improved disk and network utilization. We also measured the recovery time of the Arion client in the range 77.4ms-852.2ms, depending on the load of client write activity before the crash. Finally we also demonstrated the performance advantage and resource efficiency of Arion over alternative storage device setups in the cloud environment.

Chapter 9

Related Research

In this chapter we review the related research that was published over the past decades regarding the reliable storage management in local and large-scale environments. Initially, we survey the previous work on the storage management in virtualization and cloud environments. Then we present a study of the research on distributed filesystems and transaction processing systems. We also outline the most important studies on flash-based caching and filesystem logging. Finally, we discuss several device and application-specific reliability issues based on other related research.

9.1 Virtualization Environments

Existing solutions of cloud storage typically provide centralized management of virtual disks over a common backend, either block-based (e.g., Amazon EBS [2], OpenStack Cinder [135], Microsoft's Blizzard [121]) or object-based for improved scalability (e.g., Ceph RBD [35], Amazon S3 [4], OpenStack Swift [174]). VMFS stores disk volumes over shared cluster-based block storage [188]. VMFS employs a log-based checkpoint facility, known as distributed journaling, which enables the fast recovery from individual servers' failures.

Parallax maintains block-level virtual disk images on centralized block storage without write sharing [119]. Within each host, a dedicated storage VM translates requests for virtual blocks into requests for physical blocks on the shared blockstore. Furthermore, each host contains a local disk cache in order to hold persistent data, without the need to contact the primary shared storage immediately. Similarly, Capo uses the local disks of the hosts for multicast-based preload and block-based write-through or writeback caching [166]. Lithium implements the block-level volumes of virtual machines as a log distributed across the local storage of compute nodes [73]. The above approaches have been criticized for the semantic gap, the limited sharing opportunities and the increased performance overheads due to unnecessary multiple translations between the file and block interface [80, 106, 176].

Existing systems already apply file-based protocols in virtualization environments [58, 9]. Guests can use a standard file protocol to access a fileserver commonly installed at the host, at the cost of limited scalability and sharing [94, 145]. The host uses either a file-based protocol to connect to a fileserver, or an object-based protocol to access multiple object servers. Then, a local guest uses a file interface to access the fileserver exported by the intermediate node. However, the host may become a performance bottleneck since it eventually acts as a caching proxy and all the I/O traffic passes through it. For instance, Ventana combines file-based sharing with the versioning, migration and access control of virtual disks [145]. A client-side manager offers disk-based caching but relies on NFSv3 at the host to connect the virtual machines with object-based storage servers. Similarly, VirtFS uses a network protocol to connect a host-based fileserver to multiple local guests [94]. The scalability of Ventana and VirtFS is limited by the centralized

NFS-like server running at the host. Instead in Arion we advocate the networked access of a scalable distributed filesystem directly by the guests.

OpenStack Manila enables the integration of filesystem shares with guest machines [114]. 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. For flexibility reasons, alternative scalable backend filesystems are supported (e.g., NFSv4, GPFS). Similarly, the Amazon Elastic Filesystem (EFS) service supports the NFSv4 protocol to provide shared filesystem access to Elastic Compute Cloud (EC2) instances [58]. The design of Arion can further improve the durability of the memory-based cache of the filesystem client.

Recent studies also examined the isolation of the filesystem data structures and the I/O data path among different virtual machines co-located over a single host [165, 112]. In this context, for improved performance the authors propose transaction splitting over either the same physical journal, or via multiple physical logs.

9.2 Cloud Storage

Cloud-scale filesystems such as the Google File System (GFS) and the Hadoop Distributed File System (HDFS), are designed for high throughput, write-once sequential I/O [65, 171]. The above systems achieve high scalability and fault tolerance by striping and replicating the data in large chunks across the locally attached storage of the cluster servers. However, they provide weaker consistency guarantees by relaxing the POSIX semantics. Also, a centralized master node is responsible for the metadata management, resulting in scalability and performance bottleneck.

Cloud-backed filesystems use unmodified cloud storage services as backend storage [191, 23]. BlueSky provides on-site NFS-based proxy service of remote cloud storage through local disk caching of journal and log segments [191]. However, it lacks support for controlled file sharing and the proxy results in limited scalability. Instead, SCFS provides FUSE-based caching of entire files at the client memory and disk without the proxy bottleneck [23]. However, it lacks the journaling integration with a scalable distributed filesystem of Arion for flexible file sharing. CacheFS supports local disk-based caching

but is practically limited to read-only filesystems [86].

A recent study proposed the preliminary design of a high-performance scalable distributed filesystem with special focus on large-scale parallel and distributed applications [83]. DiDAFS aims to provide direct user-level access to remote shared storage. It also allows for client-side memory-based caching by providing alternative epoch-based consistency semantics. However, the above system depends on specialized hardware support (i.e., RDMA-aware disk controllers).

RAMCloud is a memory-based object-based storage system designed for high availability and quick recovery in case of failures [134]. RAMCloud distributes data, in the form of log segments, to secondary storage across multiple machines for fault tolerance and reconstructs lost data in parallel. A write request is acknowledged to the application when the update reaches the memory of several backup servers, while dirty data remains unflushed until the memory buffer reaches a predefined size. Nevertheless, despite the significant performance improvement of the memory-based approach, there is at least two orders of magnitude discrepancy between the memory and disk capacities [6].

9.3 Distributed Filesystems

Andrew pioneered client disk-based caching but lacked the explicit separation in data and metadata management of object-based storage [158, 193]. Coda exploited data caching strictly for availability during disconnected operation [100]. During a communication failure, a Coda client logged locally the mutating system calls. At network reconnection, each server received and replayed all the logged operations together as one transaction. On the contrary, Arion continuously logs mutations during normal operation and writes them back to efficiently maintain consistency.

The DEcorum filesystem introduced tokens to track different types of access accross the clients [99]. The Sprite distributed filesystem disabled client caching of files concurrently updated by different clients [131]. Echo introduced ordered write-behind to delay the automatic writing of cached blocks to server disks [115]. NFSv4 delegates request handling to the client for reduced latency and network traffic [142]. The client caches modified data for a predefined time period, and flushes it to the server at file close (close-to-open-

Filesystem	Cache Location	Update Policy	Validation	Client Durability
Arion [193]	disk	write-behind	capabilities	✓
AFS [158]	disk	on close	callbacks	✓
Coda [100]	disk	on close	callbacks	✓[1]
DEcorum [99]	disk	write-back	tokens	-
Sprite [131]	memory	write-behind	callbacks	-
NFSv4 [142]	memory	on close	delegations	-
Echo [115]	memory	write-behind	leases	-

Table 9.1: Comparison of distributed filesystems.

consistency). The safe asynchronous write of NFSv3 lets the server reply to a write before the data is stable on disk.

Unlike Arion, traditional filesystems limit client caching to volatile memory, or do not apply durable host-side caching for improved performance and reduced resource utilization at high scalability. Table 9.1 summarizes the main features of the above systems.

CalvinFS is a replicated, scalable filesystem that leverages a high-throughput distributed database system for metadata management [183]. The database relies on a log to store a global totally-ordered sequence of transaction requests over a replicated, distributed storage layer. Deterministic locking is a scheduling protocol that resembles two-phase locking, but requires from transactions to request all the locks that they need in their lifetime atomically and in the relative order in which they appear in the log. Unlike Arion, CalvinFS lacks the client support for local storage integration that we advocate.

9.4 Transaction Processing

Database consistency can be preserved through transaction correctness [22]. SiloR is a multicore database system that uses logging and checkpointing for fast recovery to a transactionally-consistent state without replication [204].

[1] during disconnected operation only

Early work focused on the update synchronization of multiple database copies [182]. Mutual consistency refers to the state convergence of the copies, and internal consistency refers to the preservation of invariant relations in the stored items of a copy. A majority consensus algorithm requires that a request be accepted and applied to all database copies only if it is approved by a majority of the copies. A named element has a value and a timestamp of the time at which the current value was received. Base variables are the data elements used by a query, and update variables are the data elements modified by a request. Two requests are conflicting if the base variables of the one and the update variables of the other have non-empty intersection. The voting rule mandates that the timestamps of the base variables in an update request be compared to the timestamps stored in the database copy. If after the initiation of a new request, in the meantime its base variables have been modified by an appoved conflicting request, then the new request is rejected as invalid by the respective voting copy.

An atomic action A_p is defined as a computation specified by program P and composed of primitive computational steps executed at different times and places [154]. Concurrency atomicity suggests that each step not in A_p either precede or follow all steps in A_p, and failure atomicity requires that either all steps in A_p or none of them complete. A decentralized system consists of nodes with storage blocks that do not lose their content due to failure, and support atomic read and write actions of an entire block. Pseudotime allows the relative ordering of read and write operations and enables decentralized reservation of pseudotime ranges without communication among the participants. If a write arrives at a node and has pseudotime less than that of an already-executed read it is rejected because otherwise it would cause the value returned by the previous read to be incorrect.

Viewstamped replication guarantees that the concurrent execution of transactions on replicated data is equivalent to a serial execution on non-replicated data (one-copy serializability) [133]. A system consists of networked nodes, and a distributed program consists of modules, each running at a single node. A module group consists of multiple module copies, called cohorts, which behave as a single entity. One cohort is designated as the primary, which executes procedure calls, and the remaining cohorts are backups, which are essentially passive recipients of state information from the primary. View is a set of cohorts that are capable of communicating with each other and have a designated

primary; only active views process transactions. Clients create transactions, make remote calls, and coordinate two-phase commit, while servers process remote calls and participate in two-phase commit.

Weak consistency allows database copies at different servers to vary and eventual consistency enables the servers to converge towards identical database copies in the absence of updates [179]. Eventual consistence relies on two properties: total propagation requires that each write be eventually received by each server, and consistent ordering requires that all servers apply non-commutative writes to their databases in the same order.

Snapshot isolation is a type of multiversion concurrency control in which a transaction reads the committed data from a snapshot as of the time the transaction started [20]. The transaction can only commit, if in the time period from the start to the commit, no other transaction wrote the same data written by the committing transaction. Snapshot isolation is non-serializable and provides concurrency advantage for read-only transactions, but it is not considered advantageous for long-running update transactions.

Spanner is a replicated database that assigns globally-meaningful commit transactions to distributed transactions for reflecting serialization order [48]. It supports external synchrony according to which the commit timestamp of a transaction is lower than that of another transaction if the former transaction commits before the latter starts. It also serves globally-consistent reads at a timestamp. The TrueTime API exposes clock uncertainty to assign write timestamps in monotonically increasing order and serve reads from sufficiently up-to-date replicas.

Inconsistent Replication (IR) is a protocol that offers fault-tolerance without ordering consistency [201]. The Transaction Application Protocol for Inconsistent Replication (TAPIR) provides optimistic transaction ordering on top of IR. Based on loosely synchronized clocks, the clients order their transactions according to proposed timestamps generated from their local clock and identifier. A read to a specific version of a key conflicts with a write to the same key completed before the proposed timestamp. A write conflicts with a read or write to the same key occurring after the proposed timestamp. Although the above rules provide linearizability, they can be slightly weakened to support serializability by allowing reads of past versions and writes in the past under specific timestamp conditions.

Highly Available Transactions (HAT) provide to groups of multiple operations over

multiple data items transactional guarantees that do not suffer unavailability during system partitions and do not incur high network latency [12]. HAT systems are shown to achieve a wide range of isolation levels, but fail to support prominent semantics that include snapshot isolation and one-copy serializability.

In a different semantical formulation, clients are system participants that reside on physically distinct devices and all the operations of a client are parts of a transaction [30]. The activities from each device are represented as a stream of operations interrupted by special yield operations that mark the transaction boundary. Eventually consistent transactions uphold atomicity and isolation guarantees without serialization. They provide strong guarantees that all code runs in transactions, and transactions never fail or roll back. They order transactions by both visibility and arbitration relations, unlike traditional transactions that only use a single order relation.

9.5 Flash Memory

Non-volatile memory can be used at the client and server of a distributed filesystem for I/O efficiency [14]. Writeback caching can improve performance, reduce server load, and eliminate cache warmup on restart [11]. In-place commit over non-volatile memory unifies the buffer cache with journaling [108]. Offering disk-based caching through journaling is an extension of Arion that we plan for future work.

Mercury pointed out the zero recovery point objective (RPO), i.e., no recently-written data lost from a crash. It uses flash memory in the block I/O virtualization stack of the hypervisor to provide write-through caching [31]. Non-zero RPO can be applied for improved performance via block-level writeback caching at the host. Update order is preserved by explicit tracking of the dependency between I/O requests or transaction grouping of modified blocks [101]. Due to concerns about the consistency and durability of these ordering schemes, a recent block-level solution satisfies asynchronously but explicitly the ordering constraints of application-specified write barriers [151]. Nevertheless, host-side block-based caching lacks native support for writeable file sharing within or across hosts [31, 101, 151, 11, 89].

FVP is a fault-tolerant layer that pools together all the host-side flash devices in a

cluster [24]. By intercepting the VM I/O and redirecting it to host-side flash devices, the layer leads to predictable write throughput. By replicating the VM writes to peer host-side flash devices, it preserves VM mobility and tolerates cascading host and flash failures. Unlike FVP, Arion achieves improved I/O performance and durability by relying on a separate local journal per VM rather a shared caching pool across the cluster. We leave for our future work the study of the related caching and replication issues.

9.6 Filesystem Logging

The log-structured filesystem addresses the problems of synchronous metadata updates and small writes by coalescing data writes sequentially to a segmented log [156]. Previous research reported cleaning overheads and performance limitations under particular workloads [164]. We also experimentally notice reduced read performance of the log-structured approach in some cases (Section 5.2.1). Group commit is a known database logging optimization that is used to amortize the I/O cost of inserting transaction commits to the log. It accumulates the log records from multiple transactions, and periodically flushes them to the log [55]. Instead, we emphasize fitting multiple subpage modifications from concurrent synchronous writes into a single block, and investigate the related benefits in a general-purpose journaled filesystem.

The virtual log uses a tree to logically link non-contiguous disk blocks and uses free sectors close to the head to minimize the latency of small synchronous writes [192]. StreamFS is a modified version of the log-structured filesystem for storing high-volume streams [54]. Instead, we also handle the storage traffic of low-rate streams. The hFS filesystem stores metadata and small files in a separate partition from large files. It differentiates updates by file size rather than write size that Okeanos does [203].

In the Ceph distributed filesystem, the storage servers support journaling of both data and metadata similarly to the data journaling mode of ext3 [193]. Ceph provides two new journaling modes: (i) In the writeahead mode, a write transaction returns as soon as it reaches the journal. (ii) In the parallel mode, a write transaction is written to both the journal and the filesystem, and returns when either of the two commits. The Ceph designers admit that they write all data twice for safety, and mention the

related performance tradeoff between write latency and write throughput. For the efficient storage of the journal, they support several hardware options. In our study, we extensively examine the resource requirements of data journaling, and propose two new modes to retain high performance at moderate journal traffic.

The general idea of subpage logging is not new. Previously, researchers at DEC prototyped and used the Echo distributed filesystem [26]. For improved performance and availability, Echo logged subpage updates, and bypassed the logging of page-sized or larger writes [81]. The development of Echo was discontinued in early 1992, partly because it run on hardware that lacked fast enough computation relative to communication.

Recent research introduced semantic trace playback (STP) to rapidly evaluate alternative filesystem designs without the cost of real system implementation or detailed filesystem simulation [148]. STP was used to emulate journaling of block modifications instead of entire modified blocks in a filesystem. Although the authors showed reduced amount of data written to the journal, they did not examine the general performance and recovery implications. Due to the obsolete hardware characteristics or the high emulation level of the above studies, they leave questionable the general architectural fit and actual performance benefit of journal bandwidth reduction in current filesystems. A recent work examine the performance overhead in case of journaling of journal, where the logging activity of applications runs on top of journaling filesystems [168]. This work can be complementary to Okeanos.

Optimistic crash consistency decouples ordering from durability to recover the filesystem in consistent state without expensive cache flushes [42]. Incorrect ordering is detected through transactional checksums or completely avoided through delayed block reuse; data journaling is optionally activated to preserve the storage layout at block overwrites. Backpointer-based consistency eliminates the need for ordering by adding a backpointer to every block using the out-of-band bytes provided by some devices [43]. Corrupted files are detected upon access, rather than at mount-time.

Similarly, several studies try to reduce the overhead of blocking when writing in-memory pages to disk. Externally synchronous I/O guarantees durability to an external observer of application output rather than the application itself [132]. If an application does not produce output, xsyncfs commits data periodically and asynchronously. Non-blocking writes decouple the writing of data to a page from its presence in mem-

ory [33]. However, synchronous fetching is still required under synchronous writes and writes directed to the journal device.

In earlier work, Hagmann described metadata update logging in the Cedar File System to improve performance and achieve consistency [72]. Soft updates track and enforce metadata update dependencies so that the filesystem can safely delay writes for most file operations [163]. Also, subpage journaling of metadata updates is made widely available today through popular commercial filesystems, such as the IBM JFS and MSNTFS [148] Unlike Okeanos, the above systems only focus on metadata rather than data updates. Subpage updates have been previously handled efficiently in the context of distributed shared memory by the Millipage system [90]. Instead, we introduce wasteless and selective journaling as a general filesystem service.

9.7 Other Reliability Issues

Device Issues High-performance synchronous writes can be handled through specialized hardware, such as battery-backed main memory (NVRAM) [40]. WAFL improves write performance by writing file system blocks to any location on disk and in any order, while deferring disk space allocation with the help of non-volatile RAM [82]. Reportedly, NVRAM creates a single point of failure over disk arrays, while dual-copy NVRAM cache can be costly [87]. Disk-specific knowledge can be exploited to align the data accesses on track boundaries, and avoid rotational latency and track-crossing overhead [161, 5]. This approach operates at the disk level and could complement our methods, when we update the filesystem.

High Performance Computing The I/O characteristics of parallel applications have led to middleware techniques (e.g. data sieving or collective I/O) that handle as contiguous the non-contiguous requests from parallel processes [180]. Additionally, checkpointing has a prominent role in the robust execution of high-performance parallel applications [60]. The Parallel Log-Structured Filesystem (PLFS) introduces an interposition layer that transparently writes to different files the checkpoint data from different processes instead of having all data written to a single shared file [19]. The Checkpoint-Restart File System

(CRFS) is a user-level filesystem that aggregates per-file writes in memory [139]. When the writes fill up a preconfigured chunk size (e.g. 4MB), they are asynchronously transferred to disk. The above approaches are complementary to Okeanos because they are specialized for parallel applications or checkpoints, and operate at the middleware or the user level rather than within a general-purpose filesystem.

Real-time Stream Processing In real-time data processing, application operators can recover from failures through synchronous logging at high latency [104]. Recent research combines software transactional memory with asynchronous logging to optimistically parallelize stream operators [29]. However, this approach is limited to operators that do not perform external actions such as I/O [36].

9.8 Summary

In this chapter we presented an extensive study of the related research across a wide range of storage systems. In virtualization environments a block-based interface is typically used at the cost of limited sharing opportunities and increased performance overheads. Furthermore, the proposed file-based approaches in virtualization and cloud environments either face limited scalability, or limit client caching to volatile memory, similarly to traditional distributed filesystems. We also reviewed the most important studies on transaction processing. Several studies over the last years have outlined the importance of host-side caching, but operate at the block-level and hence lack native support for controlled file sharing. Finally, we summarized previous work on filesystem logging, and discussed several device and application-specific reliability issues.

Chapter 10

Discussion and Future Work

In this chapter we discuss some challenging topics that have arised during the design and development of the proposed systems. We also present our plans for future work, and describe our ongoing work on related open research issues.

10.1 Alternative Storage Technologies

Across a range of consistency conditions, existing filesystems can be wasteful or underperforming. We propose and implement several improvements that address these weaknesses without penalizing the behavior of the filesystem beyond a reasonable increase in disk traffic. The main theme in the Okeanos design is to improve performance and consistency at low cost. Thus, adding extra spindles to improve I/O parallelism or a properly-sized NVRAM to absorb small writes, are alternative approaches likely to reduce latency and raise throughput [40, 82]. However, such solutions carry some notable drawbacks that primarily have to do with increased cost and maintenance concerns about additional faulty parts in the system.

Our effort to favor sequential writes at moderate storage traffic is compatible with the endurance and performance characteristics of novel devices such as solid-state drives based on flash memory [39]. Flash memory exhibits a number of attractive features related to low power consumption and improved access performance but also several hardware idiosyncrasies that make its behavior workload dependent. Flash memory usually consists of multiple blocks, each of which contains several pages. Data is written in units of pages, and space is erased in units of blocks. Usually a log-structured approach organizes the flash space so that writes incur low cost [196, 49]. A cleaning process periodically merges valid pages into clean blocks and reclaims the invalidated ones. The append-only nature of journaling keeps writes over flash memory relatively cheap [124]. Simple block remappings of the metadata can transform journaled updates into permanent state without relocations that lead to duplicate writes [44]. Native support of atomic writes at the flash firmware was shown to avoid duplicate writes for the safe update of database state from logged deltas of data pages [138].

In contrast, Okeanos focuses on a general-purpose local filesystem. We coalesce concurrent subpage writes to the same storage block of the journal, while we safely delay and batch small writes to the filesystem. Additionally, with selective journaling we avoid duplicate traffic to the device for sequential workloads. Our proposed modes could be directly applied as a journaled filesystem over flash memory to serve two needs: (i) reduce the amount of data sequentially written to a flash-based journal device and the wear it causes [44], (ii) decrease the number of random writes reaching the storage device of

a filesystem, because random writes are reported as harmful for the performance and lifespan of flash memory [124].

10.2 Journaling in Virtualization Environments

In virtualization environments, the block-based interface of the guest virtual machine makes small writes appear as full-block updates to the underlying filesystem. Recently, the interaction of nested filesystems has been experimentally investigated. Application workloads with reads and writes smaller than 4KB suffer the most from the full-page I/Os of the guest [80]. The data and metadata of the guest disk image are treated as data by the host filesystem. Consequently, write-intensive workloads lead to significant consistency degradation if the filesystem at the host provides metadata-only journaling. Additionally, the journaling of both data and metadata is considered non-practical due to the caused performance degradation [106]. As one solution to the performance problem of data journaling, it was recently proposed to maintain the journals of multiple virtual machines in the main memory of the host presuming that the hardware and virtual machine monitor are sufficiently reliable [88].

Instead, the wasteless and selective journaling modes could be used either as guest filesystems to reduce the downward write traffic, or as host filesystem to consistently serve the disk images of multiple virtual machines. In particular, we envision that the implementation of the proposed journaling modes at the client-side journal of a large-scale filesystem can further improve the resource efficiency of the Arion system. On the other hand, application at the host filesystem would make sense under the assumption that the guests communicate with the host through a virtualization-optimized I/O interface that flexibly supports requests of different sizes. Accordingly, we could safely serve the incoming small writes from multiple concurrent threads running across different guests and persistently store both the data and metadata of the guest filesystems. Thus, we anticipate increased consistency in the recovery of virtual images from crashes and improved guest performance during normal operation. In ongoing research, we investigate possible extensions of the proposed journaling modes in virtualization environments.

10.3 Host-side Caching

Persistent host-side caching primarily targets the improved performance and efficiency of networked storage. Typically, it uses a block-based interface that inherently lacks both the support for data sharing across different hosts and the ability for interposition in the file-based protocol of a distributed filesystem. It also makes the consistency preservation of network storage a challenging problem because the semantic gap between the file and block interfaces complicates the atomic grouping of dirty blocks by I/O request, and their ordering according to filesystem-imposed dependencies. Finally, the persistence of mapping metadata in block-based caching and the repetitive translation of I/O requests across different storage layers can introduce considerable overheads in networked storage I/O [11, 80].

The original design of Ceph cannot recover any writes that returned after they were only placed at the volatile memory of the client before a crash. Therefore, the Arion architecture is innovative because it adds durability into the client memory cache through journal-based recovery, conditionally propagates the updates to the servers after client reconnection, and also permits the clients to scalably communicate directly with the object servers of the storage backend. Overall, assuming host machines with sufficiently reliable local storage, our approach overcomes several sharing, scalability, and consistency limitations of related existing solutions. In our future work we plan to extend the host-based journaling to support caching of blocks evicted from memory [108].

10.4 Live VM Migration

A key functionality in virtualization environments is the live virtual machine migration across different hosts. Live migration involves the transfer of the memory and device state of a running VM to a different host without service interruption. Consequently, the virtual machine should only hold a limited amount of local persistent state in order to enable efficient dynamic live transfer. Although the memory requirements of virtual machines are reduced with memory deduplication and compression, disk based caching at the host provides additional benefit in that direction [71].

Client-side journaling facilitates the live migration of client virtual machines across different hosts. Essentially, each log file safely stores recent storage updates sequentially. A live migration iteratively transfers modified VM state between two hosts, until the amount of concurrently modified state drops below a threshold [117]. Eventually, the VM is suspended and resumed at the destination after the transfer of the last modified blocks. In the Arion design, the client-side journal plays the role of a write queue that naturally separates the more recent updates of each individual client and permits their efficient copy from source to destination at sequential disk throughput. Furthermore, we can trivially skip repetitive transfers of hot blocks, because they naturally appear multiple times at the front of the journal. Therefore, a client system is migrated with relatively low transfer volume across different hosts, unlike the migration of a storage server that requires relocation of the entire storage state. In our future work, we plan to extend the Arion system in order to natively support the live storage migration of virtual machines across different hosts.

10.5 Journal Replication

Loss or corruption of committed updates to critical data is recognized as a particularly damaging class of failure [84]. Hardware failures have been reported to contribute much less to service-level failures in comparison to causes related to software bugs and faults from operator or maintenance tasks [136, 84]. Similar studies also report the low frequency of disk failures at the hosts [128]. In the Arion design we assume sufficiently reliable local storage at the hosts, similar to that of the server machines. However, the availability of the client journal depends on the host recoverability in case of hardware failures.

One way to tolerate hardware failures is to replicate the client log across multiple physical hosts [73, 134, 95]. Nevertheless, replicating the log across multiple machines can be costly in terms of resource consumption, such as network and disk bandwidth. One possible approach to avoid disk bandwidth waste is to replicate the log in the memories of several servers, leaving the system susceptible to correlated failures. For instance, RAMCloud uses asynchronous replication and allows a write request to return as soon as it reaches the memory of a predefined number of nodes [134].

In our future work we aim to improve the availability of the Arion client-side journal by replicating it across multiple physical machines. A write request could be acknowledged to the application when the update reaches the memory of several secondary hosts. Alternatively, we could use an isolated pool of storage servers specifically for the needs of the log management. Further investigation is required in order to explore the tradeoffs among performance, durability and efficiency of the above approaches.

10.6 Rollback Recovery

Version consistency ensures that the filesystem correctly associates the metadata of a particular file with the data of the matching version. During the normal system operation, Arion achieves version consistency by propagating the metadata updates to the server after the respective data updates. However, a network disconnection or client reboot can happen in the middle of the writeback process of the modified data from the client memory to the servers. At this point, it is possible that the client has not transferred to the servers all the locally journaled data updates along with the corresponding metadata. In case of a conflicting access by a different client during the duration of lost connectivity, the Arion client aborts the transfer of the remaining locally cached updates for the conflicting file. Therefore, in case of a client failure, the metadata of the file does not necessarily match the version of the data.

In order to satisfy the requirements of version consistency, it is important to undo any partial updates that were not completed at the servers due to the client failure [125]. One possible solution is to keep a server-side write-ahead log of undo records during the forward operation. Thus, each server will be able to revert the effects of any incomplete updates that were active at the time of the client failure by applying its local undo log. To minimize the involved resource overhead, we can simply use memory-based logs at the servers of the filesystem. In the near future, we aim to extensively study rollback recovery in the Arion system.

10.7 Filesystem Multi-tenancy

In a multi-tenant virtualization environment the storage consolidation at the filesystem level is desirable for its data sharing, administration efficiency, and performance characteristics [145, 94, 120, 51]. Multi-tenant access control of shared files should isolate the storage access paths of different users in a secure way. Access control in a multi-tenant environment is hindered by the large number of the involved end users and the isolation of security administration required across independent organizations.

Authentication and authorization have already been extensively studied in the context of distributed systems [195]. However, a cloud environment introduces unique characteristics that necessitate the reconsideration of the assumptions and solution properties. Existing file-based solutions face scalability limitations because they either lack support for multiple tenants, rely on global-to-local identity mapping to support multi-tenancy, or have the guests and a centralized filesystem (or proxy) running at the same physical host [145, 94, 51].

In our recent work, we propose the Dike authorization architecture [96]. Dike combines native access control with tenant namespace isolation and compatibility to object-based filesystems. The filesystem natively manages the access control metadata of each tenant, and ensures that each tenant can only access its own namespace. Overall, Dike securely isolates the namespaces of the tenants, and enables configurable file-sharing among users of different tenants at limited performance overhead.

10.8 Flash-aware Filesystems

Flash storage is increasingly offering competitive advantages as either a standalone storage device in mobile systems or a distinct layer in the storage hierarchy of enterprise servers. It exhibits a number of attractive features related to low power consumption and improved access performance. However, it is also relatively expensive and bears idiosyncrasies that render its performance highly workload-dependent [39]. Compatibility with legacy systems is possible through a typical block interface offered by a flash-translation layer embedded in the device controller. A log-structured filesystem is often used to write data

sequentially to the flash medium and avoid the high cost of random writes [98]. However, a costly cleaning process is required to periodically merge valid pages into clean blocks and reclaim the invalidated blocks.

Alternatively, only a few flash-aware filesystems have been recently developed specifically for the underlying hardware characteristics [107, 93]. Flash-aware filesystems are attractive for various reasons. First, they can avoid duplication of functionality across the filesystem and the device firmware. Second, they can leverage semantical information about the application access patterns to optimize the traffic to the storage medium without compromising data persistence. Third, they can minimize the data relocation traffic occurring inside the device. Finally, they can provide end-to-end guarantees about the reliability and endurance properties that they promise.

In our ongoing work, we are developing a flash-optimized filesystem to further explore the above observations [76]. In order to achieve our goals, we propose to directly manage the flash storage with a composite filesystem that combines journaling with the log-structured filesystem (LFS) [156]. In the proposed design, flash storage will consist of two partitions, the journal partition and the LFS partition. The LFS partition organizes the permanent state of written pages into a segmented log, while the journal partition temporarily stores data and metadata writes in the form of transactions. Hence, the journal undertakes the additional responsibility to proactively clean the permanent state from frequently updated data and metadata.

Essentially, we rely on the cache timers to natively categorize pages into hot or cold and store them into the journal or LFS, respectively. With proper page expiration period, we are able to adjust the hotness boundary to current workload conditions. Additionally, the operation of journaling fully invalidates older blocks and allows the batching of multiple update requests in the operating system page cache. Overall, we expect that the proposed system will significantly improve the utilization and lifetime of the device by proactively cleaning the filesystem through the journal.

10.9 Summary

Our plans for future work include the extension of the proposed journaling modes for virtualization environments, and flash memory systems. Additionally, we plan to extend the host-based journaling to support caching of blocks evicted from memory. Another interesting future direction is the client journal replication across multiple physical machines for improved availability in case of hardware failures at the host. We also aim to investigate the extension of the Arion client-side journaling to support the live storage migration of virtual machines across different hosts. Finally, the undo of partially completed updates at the servers of the filesystem due client failures requires further investigation.

In this chapter, we also presented our ongoing work on providing namespace isolation and secure file-sharing over object-based filesystems in a multi-tenant virtualization environment. Lastly, we outlined the design of a flash-aware filesystem that we proposed recently, which combines journaling with the log-structured filesystem in order to improve the utilization and lifetime of the device.

Chapter 11

Conclusions

11.1 Contributions

In a multi-tier cloud environment, the constantly growing amount of data that needs to be stored and processed by a large number of concurrent users, has introduced new challenges in the design of large-scale storage systems. In this thesis we argue that the specific characteristics of multi-tier environments impose the fresh reconsideration of the I/O path for high performance, resource efficiency and improved consistency semantics. In this study we carefully investigate the implications of the consistency semantics on the resource efficiency and the performance across different tiers of the storage stack.

First, we focus on the local filesystem of the storage backend layer with the aim to strengthen the provided consistency semantics at improved bandwidth efficiency. We rely on journaling of data updates in order to ensure their safe transfer to disk at low latency and high operation throughput for their fast recovery in case of system failures. We design and implement the wasteless journaling mode which merges concurrent subpage writes to the journal into page-sized blocks. Additionally, we develop the selective journaling mode that only logs updates below a write threshold, and transfers the rest directly to the filesystem. Our experimental results include measurements from streaming microbenchmarks, application-level workloads, database logging traces, and multistream I/O over a

parallel filesystem in the local network. Across a wide range of realistic workloads over standalone servers and a multi-tier networked system, we demonstrate reduced write latency and recovery time, along with improved transaction throughput with low journal bandwidth requirements.

In the second part of this thesis, we concentrate on the frontend layer of a multi-tier environment. We set as our primary goal to improve the performance, resource efficiency, and durability of shared storage in the datacenter. We rely on a file-based interface for its performance, fine-grained sharing, and clear consistency properties. For enhanced end-to-end durability of shared storage, we integrate the client of a distributed filesystem with a host-based journal. At the host, we provide local durable storage to dirty data and metadata until they are written to the network servers. We carefully investigate the consistency semantics of the proposed design under normal system operation, and in case of client failures, such as network disconnection and reboot. We implement a prototype of the proposed Arion design over the Ceph production distributed filesystem. Over a local cluster and a public-cloud environment, we experimentally demonstrate promising efficiency and performance results for specific durability levels configured through the frequency of copying dirty blocks to the host-side journal, across alternative storage technologies.

To summarize, we believe that the resource efficiency at each storage tier of a multi-tier environment is critical to the overall system performance. Across different tiers, we combine improved filesystem consistency with high performance and efficient resource utilization, and explore interesting tradeoffs among performance, durability and efficiency for demanding real-world applications.

Bibliography

[1] Ramnatthan Alagappan, Vijay Chidambaram, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Beyond storage apis: Provable semantics for storage stacks. In *15th Workshop on Hot Topics in Operating Systems*, May 2015.

[2] Amazon elastic block store. https://aws.amazon.com/ebs/.

[3] Amazon elastic compute cloud. https://aws.amazon.com/ec2/.

[4] Amazon simple storage service. https://aws.amazon.com/s3/.

[5] Ashok Anand, Sayandeep Sen, Andrew Krioukov, Florentina I. Popovici, Aditya Akella, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Suman Banerjee. Avoiding file system micromanagement with range writes. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 161–176, San Diego, CA, 2008.

[6] Ganesh Ananthanarayanan, Ali Ghodsi, Scott Shenker, and Ion Stoica. Disk-locality in datacenter computing considered irrelevant. In *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems*, pages 12–12, 2011.

[7] Ganesh Ananthanarayanan, Ali Ghodsi, Andrew Wang, Dhruba Borthakur, Srikanth Kandula, Scott Shenker, and Ion Stoica. Pacman: Coordinated memory caching for parallel jobs. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, pages 20–20, 2012.

[8] Raja Appuswamy, Christos Gkantsidis, Dushyanth Narayanan, Orion Hodson, and Antony Rowstron. Scale-up vs scale-out for hadoop: Time to rethink? In *Proceedings of the 4th Annual Symposium on Cloud Computing*, pages 20:1–20:13, 2013.

[9] Raja Appuswamy, Sergey Legtchenko, and Antony Rowstron. Towards paravirtualized network file systems. In *USENIX Workshop on Hot Topics in Storage and File Systems*, Philadelphia, PA, June 2014.

[10] Raja Appuswamy, David C. van Moolenbroek, and Andrew S. Tanenbaum. Block-level RAID is dead. In *Workshop on Hot Topics in Storage in File Systems*, Boston, MA, 2010.

[11] Dulcardo Arteaga and Ming Zhao. Client-side flash caching for cloud systems. In *ACM International Systems and Storage Conference*, pages 7:1–7:11, Haifa, Israel, June 2014.

[12] Peter Bailis, Aaron Davidson, Alan Fekete ad Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Highly available transactions: Virtues and limitations (extended version). In *International Conference on Very Large Data Bases*, volume 7, September 2014.

[13] Jason Baker, Chris Bond, James C. Corbett, J. J. Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *Conference on Innovative Data Systems Research*, pages 223–234, 2011.

[14] Mary Baker, Satoshi Asami, Etienne Deprit, John Ousterhout, and Margo Seltzer. Non-volatile memory for fast, reliable file systems. In *ACM ASPLOS Conference*, pages 10–22, Boston, MA, October 1992.

[15] Mahesh Balakrishnan, Dahlia Malkhi, Ted Wobber, Ming Wu, Vijayan Prabhakaran, Michael Wei, John D. Davis, Sriram Rao, Tao Zou, and Aviad Zuck. Tango: Distributed data structures over a shared log. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 325–340, 2013.

[16] Hitesh Ballani, Keon Jang, Thomas Karagiannis, Changhoon Kim, Dinan Gunawardena, and Greg O'Shea. Chatty tenants and the cloud network sharing problem. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, pages 171–184, 2013.

[17] Alexandros Batsakis, Randal C. Burns, Arkady Kanevsky, James Lentini, and Thomas Talpey. AWOL: An adaptive write optimizations layer. In *USENIX Conference on File and Storage Technologies*, pages 67–80, February 2008.

[18] Doug Beaver, Sanjeev Kumar, Harry C. Li, Jason Sobel, and Peter Vajgel. Finding a needle in haystack: Facebook's photo storage. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, pages 1–8, 2010.

[19] John Bent, Garth A. Gibson, Gary Grider, Ben McClelland, Paul Nowoczynski, James Nunez, Milo Polte, and Meghan Wingate. PLFS: a checkpoint filesystem for parallel applications. In *Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–12, 2009.

[20] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ANSI SQL isolation levels. In *ACM SIGMOD Conference*, pages 1–10, May 1995.

[21] Philip A. Bernstein and Nathan Goodman. The failure and recovery problem for replicated databases. In *Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing*, pages 114–122, 1983.

[22] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley Publishing Co., Reading, MA, 1987.

[23] Alysson Bessani, Ricardo Mendes, Tiago Oliveira, Nuno Neves, Miguel Correia, Marcelo Pasin, and Paulo Verissimo. SCFS: a shared cloud-backed file system. In *USENIX Annual Technical Conference*, pages 169–180, Philadelphia, PA, 2014.

[24] Deepavali Bhagwat, Mahesh Patil, Michal Ostrowski, Murali Vilayannur, Woon Jung, and Chethan Kumar. A practical implementation of clustered fault tolerant write acceleration in a virtualized environment. In *USENIX Conference File and Storage Technologies*, pages 287–300, Santa Clara, CA, February 2015.

[25] Kenneth A. Birman, Daniel A. Freedman, Qi Huang, and Patrick Dowell. Overcoming CAP with consistent soft-state replication. *Computer*, 45(2):50–58, February 2012.

[26] Andrew D. Birrell, Andy Hisgen, Chuck Jerian, Timothy Mann, and Garret Swart. The Echo distributed file system. Technical Report TR-111, DEC Systems Research Center, Palo Alto, CA, September 1993.

[27] Dhruba Borthakur, Jonathan Gray, Joydeep Sen Sarma, Kannan Muthukkaruppan, Nicolas Spiegelberg, Hairong Kuang, Karthik Ranganathan, Dmytro Molkov, Aravind Menon, Samuel Rash, Rodrigo Schmidt, and Amitanand Aiyer. Apache Hadoop Goes Realtime at Facebook. In *ACM SIGMOD Conf*, pages 1071–1080, Athens, Greece, June 2011.

[28] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel*. O'Reilly Media, Sebastopol, CA, third edition, November 2005.

[29] Andrey Brito, Christof Fetzer, and Pascal Felber. Minimizing latency in fault-tolerant distributed stream processing systems. In *International Conference on Distributed Computing Systems*, pages 173–182, Montreal, QC, 2009.

[30] Sebastian Burckhardt, Daan Leijen, Manuel Fähndrich, and Mooly Sagiv. Eventually consistent transactions. In *European Symposium on Programming*, pages 67–86, April 2012. LNCS vol. 7211.

[31] Steve Byan, James Lentini, Anshul Madan, and Luis Pabon. Mercury: Host-side flash caching for the data center. In *IEEE International Conference on Mass Storage Systems and Technology*, Pacific Grove, CA, April 2012.

[32] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, and Arild Skjolsvold et al. Windows Azure Storage: a highly available cloud storage service with strong consistency. In *ACM Symposium on Operating Systems Principles*, pages 143–157, Cascais, Portugal, October 2011.

[33] Daniel Campello, Hector Lopez, Luis Useche, Ricardo Koller, and Raju Rangaswami. Non-blocking writes to files. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, pages 151–165, 2015.

[34] P. Carns, S. Lang, R. Ross, M. Vilayannur, J. Kunkel, and T. Ludwig. Small-file access in parallel file systems. In *IEEE International Parallel and Distributed Processing Symposium*, pages 1–11, May 2009.

[35] Ceph rados block devive. http://ceph.com/docs/master/rbd/rbd/.

[36] Sirish Chandrasekaran and Michael Franklin. Remembrance of streams past: Overload-sensitive management of archived streams. In *Very Large Data Bases*, pages 348–359, Toronto, Canada, August 2004.

[37] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 205–218, 2006.

[38] Seagate Cheetah 15K.5 SAS (ST3300655SS), August 2007. Product Manual, `http://www.seagate.com/staticfiles/support/disc/manuals/enterprise/cheetah/15K.5/SAS/100384784e.pdf`.

[39] F. Chen, D. A. Koufaty, and X. Zhang. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In *SIGMETRICS/Performance*, pages 181–192, Seattle, WA, June 2009.

[40] Peter M. Chen, Wee Teck Ng, Subhachandra Chandra, Christopher M. Aycock, Gurushankar Rajamani, and David E. Lowell. The Rio file cache: Surviving operating system crashes. In *ACM International Conference Architectural Support for Programming Languages and Operating Systems*, pages 74–83, Cambridge, MA, 1996.

[41] Peter M. Chen and Brian D. Noble. When virtual is better than real. In *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems*, pages 133–, 2001.

[42] Vijay Chidambaram, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Optimistic crash consistency. In *ACM Symposium on Operating Systems Principles*, pages 228–243, Farminton, PA, November 2013.

[43] Vijay Chidambaram, Tushar Sharma, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Consistency without ordering. In *USENIX Conference on File and Storage Technologies*, pages 101–116, San Jose, CA, February 2012.

[44] Hyun Jin Choi, Seung-Ho Lim, and Kyu Ho Park. JFTL: A flash translation layer based on a journal remapping for flash memory. *ACM Transactions on Storage*, 4:14:1–14:22, February 2009.

[45] Asaf Cidon, Stephen M. Rumble, Ryan Stutsman, Sachin Katti, John Ousterhout, and Mendel Rosenblum. Copysets: Reducing the frequency of data loss in cloud storage. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, pages 37–48, 2013.

[46] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. Better i/o through byte-addressable, persistent memory. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, pages 133–146, 2009.

[47] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, pages 143–154, 2010. https://github.com/brianfrankcooper/YCSB.

[48] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally-distributed database. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 251–264, Hollywood, CA, October 2012.

[49] Hui Dai, Michael Neufeld, and Richard Han. ELF: An efficient log-structured flash file system for micro sensor nodes. In *ACM International Conference on Embedded Networked Sensor Systems*, pages 176–187, Baltimore, MD, November 2004.

[50] Jeff Darcy. HekaFS. `http://hekafs.org`.

[51] Jeff Darcy. Building a cloud file system. *USENIX; login:*, 36(3):14–21, June 2011.

[52] Database test suite. http://osdldbt.sourceforge.net/.

[53] Jeff Dean and Sanjay Ghemawat. Leveldb. https://code.google.com/p/leveldb/.

[54] Peter Desnoyers and Prashant J. Shenoy. Hyperion: High volume stream archival for retrospective querying. In *USENIX Annual Technical Conference*, pages 45–58, Santa Clara, CA, June 2007.

[55] David J. DeWitt, Randy H. Katz, Frank Olken, Leonard D. Shapiro, Michael Stonebraker, and David A. Wood. Implementation techniques for main memory database systems. In *ACM SIGMOD*, pages 1–8, Boston, MA, 1984.

[56] Dropbox. https://www.dropbox.com/.

[57] John K. Edwards, Daniel Ellard, Craig Everhart, Robert Fair, Eric Hamilton, Andy Kahn, Arkady Kanevsky, James Lentini, Ashish Prakash, Keith A. Smith, and Edward Zayas. FlexVol: flexible, efficient file volume virtualization in WAFL. In *USENIX Annual Technical Conference*, pages 129–142, Boston, MA, June 2008.

[58] Amazon elastic filesystem. https://aws.amazon.com/efs/.

[59] John C. Eidson. *Measurement, Control, and Communication Using IEEE 1588*. Springer-Verlag London Limited, April 2006.

[60] E. N. Elnozahy and James S. Plank. Checkpointing for peta-scale systems: A look into the future of practical rollback-recovery. *IEEE Transactions on Dependable and Secure Computing*, 1(2):97–108, 2004.

[61] Emc, 2014. http://www.emc.com/leadership/digital-universe/2014iview/executive-summary.htm.

[62] Filebench, 2011. http://sourceforge.net/apps/mediawiki/filebench/index.php.

[63] Daniel Fryer, Kuei Sun, Rahat Mahmood, TingHao Cheng, Shaun Benjamin, Ashvin Goel, and Angela Demke Brown. Recon: Verifying file system consistency at runtime. In *USENIX Conference on File and Storage Technologies*, pages 73–86, San Jose, CA, February 2012.

[64] Roxana Geambasu, Steven D. Gribble, and Henry M. Levy. Cloudviews: Communal data sharing in public clouds. In *Proceedings of the 2009 Conference on Hot Topics in Cloud Computing*, 2009.

[65] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *ACM Symposium on Operating Systems Principles*, pages 29–43, Bolton Landing, NY, October 2003.

[66] J. Gray and A. Reuter. *Transaction Processing: concepts and techniques*, chapter 9. Log Manager. Morgan Kaufmann Publishers, 1993.

[67] Laura M. Grupp, John D. Davis, and Steven Swanson. The bleak future of nand flash memory. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, pages 2–2, 2012.

[68] Laura M. Grupp, John D. Davis, and Steven Swanson. The bleak future of NAND flash memory. In *USENIX Conference on File and Storage Technologies*, pages 17–24, San Jose, CA, February 2012.

[69] Ajay Gulati, Chethan Kumar, and Irfan Ahmad. Storage workload characterization and consolidation in virtualized environments. In *International Workshop on Virtualization Performance: Analysis, Characterization, and Tools*, Boston, MA, April 2009.

[70] Ajay Gulati, Ganesha Shanmuganathan, Irfan Ahmad, Carl Waldspurger, and Mustafa Uysal. Pesto: Online storage performance management in virtualized datacenters. In *Proceedings of the 2Nd ACM Symposium on Cloud Computing*, pages 19:1–19:14, 2011.

[71] Diwaker Gupta, Sangmin Lee, Michael Vrable, Stefan Savage, Alex C. Snoeren, George Varghese, Geoffrey M. Voelker, and Amin Vahdat. Difference engine: Harnessing memory redundancy in virtual machines. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 309–322, San Diego, CA, December 2008.

[72] Robert Hagmann. Reimplementing the Cedar file system using logging and group commit. In *ACM Symposium on Operating Systems Principles*, pages 155–162, Austin, TX, December 1987.

[73] Jacob Gorm Hansen and Eric Jul. Lithium: Virtual machine storage for the cloud. In *ACM Symposium on Cloud Computing*, pages 15–26, Indianapolis, IN, June 2010.

[74] Tyler Harter, Dhruba Borthakur, Siying Dong, Amitanand Aiyer, Liyin Tang, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Analysis of hdfs under hbase: A facebook messages case study. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies*, pages 199–212, 2014.

[75] Tyler Harter, Chris Dragga, Michael Vaughn, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. A file is not a file: Understanding the i/o behavior of apple desktop applications. *ACM Transactions on Compututer Systems*, 30(3):10:1–10:39, August 2012.

[76] Andromachi Hatzieleftheriou and Stergios V. Anastasiadis. Jlfs: Journaling the log-structured filesystem for proactive cleaning in flash storage. In *USENIX Annual Technical Conference*, Portland, OR, June 2011. (poster).

[77] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[78] Dean Hildebrand, Arifa Nisar, and Roger Haskin. pnfs, posix, and mpi-io: A tale of three semantics. In *Proceedings of the 4th Annual Workshop on Petascale Data Storage*, pages 32–36, 2009.

[79] Dean Hildebrand, Lee Ward, and Peter Honeyman. Large files, small writes, and pNFS. In *ACM International Conference on Supercomputing*, pages 116–124, Cairns, Australia, June 2006.

[80] Dean Hilderbrand, Anna Povzner, Renu Tewari, and Vasily Tarasov. Revisiting the storage stack in virtualized NAS environments. In *USENIX Workshop on I/O Virtualization*, Portland, OR, June 2011.

[81] Andy Hisgen, Andrew Birrell, Charles Jerian, Timothy Mann, and Garret Swart. New-value logging in the Echo replicated file system. Technical report, Digital Equipment Corporation, Palo Alto, CA, 1993. SRC 104.

[82] Dave Hitz, James Lau, and Michael A. Malcolm. File system design for an NFS file server appliance. In *USENIX Winter Technical Conference*, pages 235–246, San Francisco, CA, January 1994.

[83] Torsten Hoefler, Robert B. Ross, and Timothy Roscoe. Distributing the data plane for remote storage access. In *15th Workshop on Hot Topics in Operating Systems*, Kartause Ittingen, Switzerland, 2015. USENIX Association.

[84] Urs Hoelzle and Luiz Andre Barroso. *The Datacenter As a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan and Claypool Publishers, 1st edition, 2009.

[85] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.

[86] David Howells. FS-Cache: a network filesystem caching facility. In *Linux Symposium*, Ottawa, Canada, July 2006.

[87] Yiming Hu, Tycho Nightingale, and Qing Yang. RAPID-Cache–a reliable and inexpensive write cache for high performance storage systems. *IEEE Transactions on Parallel and Distributed Systems*, 13(3):290–307, March 2002.

[88] Ting-Chang Huang and Da-Wei Chang. VM aware journaling: improving journaling file system performance in virtualization environments. *Software - Practice and Experience*, 42(3):303–330, 2011.

[89] Fusion-io ioturbine. http://www.fusionio.com/products/ioturbine-virtual.

[90] Ayal Itzkovitz and Assaf Schuster. MultiView and Millipage - fine-grain sharing in page-based DSMs. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 215–228, New Orleans, LA, February 1999.

[91] William Jannen, Chia-Che Tsai, and Donald E. Porter. Virtualize storage, not disks. In *Proceedings of the 14th USENIX Conference on Hot Topics in Operating Systems*, pages 3–3, 2013.

[92] Microsoft Exchange Server Jetstress Tool, 2007. http://technet.microsoft.com/en-us/library/bb643093.aspx.

[93] William K. Josephson, Lars A. Bongo, David Flynn, and Kai Li. DFS: a file system for virtualized flash storage. In *USENIX Conference on File and Storage Technologies*, pages 85–100, San Jose, CA, February 2010.

[94] Venkateswararao Jujjuri, Eric Van Hensbergen, and Anthony Liguori. VirtFS virtualization aware file system pass-through. In *Ottawa Linux Symposium*, Ottawa, Canada, July 2010.

[95] Flavio P. Junqueira, Ivan Kelly, and Benjamin Reed. Durability with bookkeeper. *SIGOPS Oper. Syst. Rev.*, 47(1):9–15, January 2013.

[96] Giorgos Kappes, Andromachi Hatzieleftheriou, and Stergios V. Anastasiadis. Virtualization-aware access control for multitenant filesystems. In *IEEE International Conference on Massive Storage Systems and Technology*, Santa Clara, CA, June 2014.

[97] Jeffrey Katcher. PostMark: A new file system benchmark. Technical Report TR-3022, NetApp, 1997.

[98] Atsuo Kawaguchi, Shingo Nishioka, and Hiroshi Motoda. A flash-memory based file system. In *USENIX Technical Conference*, 1995.

[99] Michael L Kazar, Bruce W Leverett, Owen T Anderson, Vasilis Apostolides, Beth A Bottos, Sailesh Chutani, Craig F Everhart, W Anthony Mason, Shu-Tsui Tu, and Edward R Zayas. DEcorum file system architectural overview. In *USENIX Summer Technical Conference*, pages 151–164, Anaheim, CA, June 1990.

[100] James J. Kistler and M. Satyanarayanan. Disconnected operation in the coda file system. *ACM Transactions Computer Systems*, 10(1):3–25, February 1992.

[101] Ricardo Koller, Leonardo Marmol, Raju Rangaswami, Swaminathan Sundararaman, Nisha Talagala, and Ming Zhao. Write policies for host-side flash caches. In *USENIX Conference on File and Storage Technologies*, pages 45–58, San Jose, CA, February 2013.

[102] Aneesh Kumar K.V, Minming Cao, Jose R. Santos, and Andreas Dilger. Ext4 block and inode allocator improvements. In *Linux Symposium*, pages 263–274, Ottawa, Canada, July 2008.

[103] Anil Kurmus, Moitrayee Gupta, Roman Pletka, Christian Cachin, and Robert Haas. A Comparison of Secure Multi-tenancy Architectures for Filesystem Storage Clouds. In *ACM/IFIP/USENIX International Middleware Conference*, pages 460–479, Lisboa, Portugal, December 2011.

[104] YongChul Kwon, Magdalena Balazinska, and Albert Greensberg. Fault-tolerant stream processing using a distributed, replicated file system. In *Very Large Data Bases Conference*, pages 574–585, Auckland, New Zeland, August 2008.

[105] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[106] Duy Le, Hai Huang, and Haining Wang. Understanding performance implications of nested file systems in a virtualized environment. In *USENIX Conference File and Storage Technologies*, pages 87–100, San Jose, CA, February 2012.

[107] Changman Lee, Dongho Sim, Joo-Young Hwang, and Sangyeun Cho. F2fs: A new file system for flash storage. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, pages 273–286, 2015.

[108] Eunji Lee, Hyokyung Bahn, and Sam H. Noh. A unified buffer cache architecture that subsumes journaling functionality via nonvolatile memory. *ACM Transactions on Storage*, 10(1):1:1–1:17, January 2014.

[109] Andrew W. Leung, Shankar Pasupathy, Garth Goodson, and Ethan L. Miller. Measurement and analysis of large-scale network file system workloads. In *USENIX Annual Technical Conference*, pages 213–226, Boston, MA, 2008.

[110] Xing Lin, Yun Mao, Feifei Li, and Robert Ricci. Towards fair sharing of block storage in a multi-tenant cloud. In *Proceedings of the 4th USENIX Conference on Hot Topics in Cloud Ccomputing*, pages 15–15, 2012.

[111] Lanyue Lu, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Fault isolation and quick recovery in isolation file systems. In *USENIX HotStorage Workshop*, San Jose, CA, June 2013.

[112] Lanyue Lu, Yupu Zhang, Thanh Do, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Physical disentanglement in a container-based file system. In *USENIX Symposium on Operating Systems Design and Implementation*, Broomfield, CO, October 2014.

[113] M. Mammarella, S. Hovsepian, and E. Kohler. Modular data storage with Anvil. In *ACM Symposium on Operating Systems Principles*, pages 147–160, October 2009.

[114] Openstack manila. https://wiki.openstack.org/wiki/Manila.

[115] Timothy Mann, Andrew Birrell, Andy Hisgen, Charles Jerian, and Garret Swart. A coherent distributed file cache with directory write-behind. *ACM Transactions on Computer Systems*, 12(2):123–164, May 1994.

[116] Yandong Mao, Eddie Kohler, and Robert Morris. Cache craftiness for fast multicore key-value storage. In *ACM European Conference on Computer Systems*, Bern, Switzerland, April 2012.

[117] Ali Mashtizadeh, Emré Celebi, Tal Garfinkel, and Min Cai. The design and evolution of live storage migration in vmware esx. In *USENIX Annual Technical Conference*, pages 187–200, Portland, OR, June 2011.

[118] Michael Mesnier, Feng Chen, Tian Luo, and Jason Akers. Differentiated storage services. In *ACM Symposium on Operating Systems Pinciples*, pages 57–70, Cascai, Portugal, October 2011.

[119] Dutch T. Meyer, Gitika Aggarwal, Brendan Cully, Geoffrey Lefebvre, Michael J. Feeley, Norman C. Hutchinson, and Andrew Warfield. Parallax: virtual disks for

virtual machines. In *ACM European Conference on Computer Systems*, pages 41–54, Glasgow, Scotland, UK, April 2008.

[120] Dutch T. Meyer, Jake Wires, Norman C. Hutchinson, and Andrew Warfield. Namespace Management in Virtual Desktops. *USENIX; login:*, 36(1):6–11, February 2011.

[121] James Mickens, Edmund B. Nightingale, Jeremy Elson, Krishna Nareddy, Darren Gehring, Bin Fan, Asim Kadav, Vijay Chidambaram, and Osama Khan. Blizzard: Fast, cloud-scale block storage for cloud-oblivious applications. In *USENIX Symposium on Networked Systems Design and Implementation*, pages 257–273, Seattle, WA, April 2014.

[122] Madalin Mihailescu, Gokul Soundararajan, and Cristiana Amza. Mixapart: Decoupled analytics for shared storage systems. In *Proceedings of the 4th USENIX Conference on Hot Topics in Storage and File Systems*, pages 2–2, 2012.

[123] David L. Mills. Improved algorithms for synchronizing computer network clocks. *IEEE/ACM Transactions on Networking*, 3(3):245–254, June 1995.

[124] Changwoo Min, Kangnyeon Kim, Hyunjin Cho, Sang-Won Lee, and Young Ik Eom. SFS: random write considered harmful in solid state drives. In *USENIX Conference on File and Storage Technologies*, pages 139–154, San Jose, CA, February 2012.

[125] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems*, 17(1):94–162, March 1992.

[126] The Los Alamos National Lab MPI-IO Test. http://public.lanl.gov/jnunez/benchmarks/mpiiotest.htm.

[127] Craig S. Mullins. *Database Administration: The Complete Guide to Practices and Procedures*, chapter 11. Database Performance (Database Log Placement), page 308. Addison Wesley, 2002.

[128] Subramanian Muralidhar, Wyatt Lloyd, Sabyasachi Roy, Cory Hill, Ernest Lin, Weiwen Liu, Satadru Pan, Shiva Shankar, Viswanath Sivakumar, Linpeng Tang,

and Sanjeev Kumar. F4: Facebook's warm blob storage system. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, pages 383–398, 2014.

[129] MySQL. http://www.mysql.com/.

[130] Dushyanth Narayanan, Eno Thereska, Austin Donnelly, Sameh Elnikety, and Antony Rowstron. Migrating server storage to SSDs: Analysis of tradeoffs. In *ACM European Conference on Computer Systems*, pages 145–158, Nuremberg, Germany, April 2009.

[131] Michael N. Nelson, Brent B. Welch, and John K. Ousterhout. Caching in the Sprite network filesystem. *ACM Transactions on Computer Systems*, 6(1):134–154, February 1988.

[132] Edmund B. Nightingale, Kaushik Veeraraghavan, Peter M. Chen, and Jason Flinn. Rethink the sync. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, pages 1–14, 2006.

[133] Brian M. Oki and Barbara H. Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *ACM Symposium on Principles of Distributed Computing*, pages 8–17, Toronto, Canada, August 1988.

[134] Diego Ongaro, Stephen M. Rumble, Ryan Stutsman, John Ousterhout, and Mendel Rosenblum. Fast crash recovery in ramcloud. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 29–41, 2011.

[135] Openstack cinder. https://wiki.openstack.org/wiki/Cinder.

[136] David Oppenheimer, Archana Ganapathi, and David A. Patterson. Why do Internet services fail, and what can be done about it? In *USENIX Symposium on Internet Technologies and Systems*, pages 1–15, Seattle, WA, March 2003.

[137] Sarp Oral, Feiyi Wang, David Dillow, Galen Shipman, Ross Miller, and Oleg Drokin. Efficient object storage journaling in a distributed parallel file system. In *USENIX Conference on File and Storage Technologies*, pages 143–154, San Jose, CA, February 2010.

[138] Xiangyong Ouyang, David Nellans, Robert Wipfel, David Flynn, and Dhabaleswar K. Panda. Beyond block I/O: Rethinking traditional storage primitives. In *IEEE International Symposium on High Performance Computer Architecture*, pages 301–311, San Antonio, TX, February 2011.

[139] Xiangyong Ouyang, Raghunath Rajachandrasekar, Xavier Besseron, Hao Wang, Jian Huang, and Dhabaleswar K. Panda. CRFS: A lightweight user-level filesystem for generic checkpoint/restart. In *International Conference Parallel Processing*, pages 375–384, Taipei, Taiwan, September 2011.

[140] Stan Park, Terence Kelly, and Kai Shen. Failure-atomic msync(): A simple and efficient mechanism for preserving the integrity of durable data. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 225–238, 2013.

[141] Hugo Patterson, Stephen Manley, Mike Federwisch, Dave Hitz, Steve Kleiman, and Shane Owara. SnapMirror: File system based asynchronous mirroring for disaster recovery. In *USENIX Conference File and Storage Technologies*, Monterey, CA, 2002.

[142] Brian Pawlowski, Chet Juszczak, Peter Staubach, Carl Smith, Diane Lebel, and David Hitz. NFS version 3 design and implementation. In *USENIX Summer Technical Conference*, pages 137–152, Boston, MA, June 1994.

[143] Brian Pawlowski, David Noveck, David Robinson, and Robert Thurlow. The NFS version 4 protocol. In *SANE International System Administration and Networking Conference*, Maastricht, Netherlands, May 2000.

[144] Karin Petersen, Mike Spreitzer, Douglas Terry, and Marvin Theimer. Bayou: Replicated database services for world-wide applications. In *Proceedings of the 7th Workshop on ACM SIGOPS European Workshop: Systems Support for Worldwide Applications*, pages 275–280, 1996.

[145] Ben Pfaff, Tal Garfinkel, and Mendel Rosenblum. Virtualization aware file systems: Getting beyond the limitations of virtual disks. In *USENIX Symposium on Networked Systems Design and Implementation*, pages 353–366, San Jose, CA, May 2006.

[146] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. All file systems are not created equal: On the complexity of crafting crash-consistent applications. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 433–448, 2014.

[147] Milo Polte, Jiri Simsa, Wittawat Tantisiriroj, Garth Gibson, Shobhit Dayal, Mikhail Chainani, and Dilip Kumar Uppugandla. Fast log-based concurrent writing of checkpoints. In *Petascale Data Storage Workshop*, November 2008.

[148] Vijayan Prabhakaran, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Analysis and evolution of journaling file systems. In *USENIX Annual Technical Conference*, pages 105–120, Anaheim, CA, 2005.

[149] Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. IRON file systems. In *ACM Symposium on Operating Systems Principles*, pages 206–220, Brighton, United Kingdom, October 2005.

[150] Parallel virtual file system, version 2. http://www.pvfs.org.

[151] Dai Qin, Angela Demke Brown, and Ashvin Goel. Reliable writeback for client-side flash caches. In *USENIX Annual Technical Conference*, pages 451–462, Philadelphia, PA, June 2014.

[152] Abhishek Rajimwale, Vijay Chidambaram, Deepak Ramamurthi, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Coerced cache eviction and discreet-mode journaling: Dealing with misbehaving disks. In *International Conference on Dependable Systems and Networks*, pages 518–529, Hong Kong, China, June 2011.

[153] Redis nosql store. http://redis.io.

[154] David P. Reed. Implementing atomic actions on decentralized data. *ACM Transactions on Computer Systems*, 1(1):3–23, February 1983.

[155] Charles Reiss, Alexey Tumanov, Gregory R. Ganger, Randy H. Katz, and Michael A. Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace

analysis. In *Proceedings of the Third ACM Symposium on Cloud Computing*, pages 7:1–7:13, 2012.

[156] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):26–52, February 1992.

[157] Serial ATA: high speed serialized AT attachment, January 2003. Revision 1.0a, SerialATA Workgroup.

[158] Mahadev Satanarayanan. Scalable, secure, and highly available distributed file access. *Computer*, 23(5):9–21, May 1990.

[159] M. Satyanarayanan, Henry H. Mashburn, Puneet Kumar, David C. Steere, and James J. Kistler. Lightweight recoverable virtual memory. In *ACM SIGOPS*, pages 146–160, Asheville, NC, December 1993.

[160] Working Draft Project American National Standard, SCSI Block Commands-3, Technical Committee T10, INCITS, 2005, 2005. ftp://ftp.t10.org/t10/document.05/05-369r0.pdf.

[161] Jiri Schindler, John Linwood Griffin, Christopher R. Lumb, and Gregory R. Ganger. Track-aligned extents: Matching access patterns to disk drive characteristics. In *USENIX Conference on File and Storage Technologies*, pages 259–274, Monterey, CA, January 2002.

[162] Russell Sears and Eric Brewer. Stasis: flexible transactional storage. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 29–44, Seattle, WA, November 2006.

[163] Margo I. Seltzer, Gregory R. Ganger, Marshall K. McKusick, Keith A. Smith, Craig A. N. Soules, and Christopher A. Stein. Journaling versus soft updates: Asynchronous meta-data protection in file systems. In *USENIX Annual Technical Conference*, pages 71–84, San Diego, CA, 2000.

[164] Margo I. Seltzer, Keith A. Smith, Hari Balakrishnan, Jacqueline Chang, Sara McMains, and Venkata N. Padmanabhan. File system logging versus clustering: a

performance comparison. In *USENIX Annual Technical Conference*, pages 21–21, 1995.

[165] Yannis Sfakianakis, Stelios Mavridis, Anastasios Papagiannis, Spyridon Papageorgiou, Markos Fountoulakis, Manolis Marazakis, and Angelos Bilas. Vanguard: Increasing server efficiency via workload isolation in the storage i/o path. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 19:1–19:13, 2014.

[166] Mohammad Shamma, Dutch T. Meyer, Jake Wires, Maria Ivanova, Norman C. Hutchinson, and Andrew Warfield. Capo: Recapitulating storage for virtual desktops. In *USENIX Conference on File and Storage Technologies*, pages 31–45, San Jose, CA, February 2011.

[167] Justin Sheehy. There is no now. *Communications of the ACM*, 58(5):36–41, May 2015.

[168] Kai Shen, Stan Park, and Men Zhu. Journaling of journal is (almost) free. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies*, pages 287–293, Santa Clara, CA, 2014.

[169] Dong In Shin, Young Jin Yu, Hyeong S. Kim, Hyeonsang Eom, and Heon Young Yeom. Request bridging and interleaving: Improving the performance of small synchronous updates under seek-optimizing disk subsystems. *ACM Transactions on Storage*, 7(2):4:1–4:31, July 2011.

[170] Ji-Yong Shin, Mahesh Balakrishnan, Tudor Marian, and Hakim Weatherspoon. Gecko: Contention-oblivious disk arrays for cloud storage. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies*, pages 285–298, 2013.

[171] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies*, pages 1–10, Washington, DC, USA, 2010.

[172] Dawn Song, Elaine Shi, Ian Fischer, and Umesh Shankar. Cloud data protection for the masses. *Computer*, 45(1):39–45, January 2012.

[173] Richard P. Spillane, Pradeep J. Shetty, Erez Zadok, Sagar Dixit, and Shrikar Archak. An efficient multi-tier tablet server storage architecture. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, pages 1:1–1:14, 2011.

[174] Openstack swift. http://docs.openstack.org/developer/swift/.

[175] Sysbench. a system performance benchmark. http://github.com/akopytov/sysbench.

[176] Vasily Tarasov, Dean Hildebrand, Geoff Kuenning, and Erez Zadok. Virtual machine workloads: The case for new benchmarks for NAS. In *USENIX Conference File and Storage Technologies*, pages 307–320, San Jose, CA, February 2013.

[177] Vasily Tarasov, Deepak Jain, Dean Hildebrand, Renu Tewari, Geoff Kuenning, and Erez Zadok. Improving I/O performance using virtual disk introspection. In *USENIX Workshop on Hot Topics in Storage and File Systems*, San Jose, CA, June 2013.

[178] Doug Terry. Replicated data consistency explained through baseball. *Commun. ACM*, 56(12):82–89, December 2013.

[179] Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike J. Spreitzer, Marvin M. Theimer, and Brent B. Welch. Session guarantees for weakly consistent replicated data. In *International Conference on Parallel and Distributed Information Systems*, pages 140–149, Austin, TX, September 1994.

[180] Rajeev Thakur, William Gropp, and Ewing Lusk. Data Sieving and Collective I/O in ROMIO. In *IEEE Symposium Frontiers of Massively Parallel Computation*, pages 182–189, Annapolis, MD, February 1999.

[181] The Austin Group. *POSIX.1-2008 Volume 2: System Interfaces*. IEEE Std 1003.1 and The Open Group Base Specifications Issue 7, 2008.

[182] Robert H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Transactions on Database Systems*, 4(2):180–209, June 1979.

[183] Alexander Thomson and Daniel J. Abadi. CalvinFS: Consistent WAN replication and scalable metadata management for distributed file systems. In *USENIX Conference on File and Storage Technologies*, pages 1–14, Santa Clara, CA, February 2015.

[184] Ashish Thusoo, Zheng Shao, Suresh Anthony, Dhruba Borthakur, Namit Jain, Joydeep Sen Sarma, Raghotham Murthy, and Hao Liu. Data warehousing and analytics infrastructure at facebook. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 1013–1020, 2010.

[185] TPC benchmark C standard specification. Technical report, Transaction Processing Council, 1992.

[186] Stephen C. Tweedie. Journaling the Linux ext2fs filesystem. In *LinuxExpo*, pages 25–29, Durham, NC, 1998.

[187] Twitter, 2014. https://about.twitter.com/company.

[188] Satyam B. Vaghani. Virtual machine file system. *ACM SIGOPS Operating Systems Review*, 44(4):57–70, December 2010.

[189] David C. van Moolenbroek, Raja Appuswamy, and Andrew S. Tanenbaum. Towards a flexible, lightweight virtualization alternative. In *ACM International Systems and Storage Conference*, pages 8:1–8:7, June 2014.

[190] Paulo Verissimo and Luis Rodrigues. *Distributed Systems for System Architects.* Kluwer Academic Publishers, Norwell, MA, USA, 2001.

[191] Michael Vrable, Stefan Savage, and Geoffrey M. Voelker. Bluesky: A cloud-backed file system for the enterprise. In *USENIX Conference File and Storage Technologies*, pages 237–250, San Jose, CA, February 2012.

[192] Randolph Y. Wang, Thomas E. Anderson, and David A. Patterson. Virtual log based file systems for a programmable disk. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 29–43, New Orleans, LA, 1999.

[193] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *USENIX*

Symposium on Operating Systems Design and Implementation, pages 307–320, Seattle, WA, November 2006. `http://ceph.newdream.net/wiki/OSD_journal`.

[194] Zev Weiss, Tyler Harter, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Root: Replaying multithreaded traces with resource-oriented ordering. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 373–387, 2013.

[195] Edward Wobber, Martín Abadi, Michael Burrows, and Butler Lampson. Authentication in the Taos operating system. *ACM Transactions on Computer Systems*, 12(1):3–32, February 1994.

[196] David Woodhouse. JFFS: The journaling flash file system. In *Linux Symposium*, Ottawa, Canada, 2001.

[197] A. Yoshiji, R. Konishi, K. Sato, H. Hifumi, Y. Tamura, S. Kihara, and S. Moriai. NILFS - continuous snapshotting filesystem for Linux, 2009. NTT Corporation, http://www.nilfs.org/en/.

[198] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, pages 10–10, 2010.

[199] Matei Zaharia, Benjamin Hindman, Andy Konwinski, Ali Ghodsi, Anthony D. Joesph, Randy Katz, Scott Shenker, and Ion Stoica. The datacenter needs an operating system. In *Proceedings of the 3rd USENIX Conference on Hot Topics in Cloud Computing*, pages 17–17, 2011.

[200] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy Katz, and Ion Stoica. Improving mapreduce performance in heterogeneous environments. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, pages 29–42, 2008.

[201] Irene Zhang, Naveen Kr. Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan R. K. Ports. Building consistent transactions with inconsistent replication. Technical report, University of Washington, December 2014. UW-CSE-14-12-01.

[202] Yupu Zhang, Abhishek Rajimwale, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. End-to-end data integrity for file systems: A zfs case study. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies*, 2010.

[203] Zhihui Zhang and Kanad Ghose. hFS: a hybrid file system prototype for improving small file and metadata performance. In *ACM European Conference on Computer Systems*, pages 175–187, Lisboa, Portugal, March 2007.

[204] Wenting Zheng, Stephen Tu, Eddie Kohler, and Barbara Liskov. Fast databases with fast durability and recovery through multicore parallelism. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 465–477, Broomfield, CO, October 2014.

Author's Publications

Related publications:

1. **Andromachi Hatzieleftheriou**, Stergios V. Anastasiadis, *Host-side Filesystem Journaling for Durable Shared Storage*, USENIX Conference on File and Storage Technologies (FAST), Santa Clara, CA, USA, February 2015.

2. **Andromachi Hatzieleftheriou**, Stergios V. Anastasiadis, *Improving Bandwidth Efficiency for Consistent Multistream Storage*, ACM Transactions on Storage (TOS), vol 9, no 1, March 2013.

3. **Andromachi Hatzieleftheriou**, Stergios V. Anastasiadis, *JLFS: Journaling the Log-Structured Filesystem for Proactive Cleaning in Flash Storage*, USENIX Annual Technical Conference (ATC), Portland, OR, USA, June 2011 (poster).

4. **Andromachi Hatzieleftheriou**, Stergios V. Anastasiadis, *Okeanos: Wasteless Journaling for Fast and Reliable Multistream Storage*, USENIX Annual Technical Conference (ATC), Portland, OR, USA, June 2011.

Other publications:

1. 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.

2. Giorgos Margaritis, **Andromachi Hatzieleftheriou**, Stergios V. Anastasiadis, *Nephele: Scalable Access Control for Federated File Services*, Journal of Grid Computing, pub. Springer, Volume 11, Issue 1, pp 83-102, March 2013.

Short Vita

Andromachi Hatzieleftheriou was born in Serres, Greece in 1985. She studied Computer Science at the University of Ioannina, where she received her BSc and MSc degrees in 2006 and 2009, respectively. Since the April of 2009 he has been a Ph.D. candidate in the same Department under the supervision of Prof. Stergios Anastasiadis. Her research interests include file and storage systems, distributed systems and storage virtualization.