Fast and Reliable Stream Storage Through Differential Data Journaling

Andromachi Hatzieleftheriou

MASTER THESIS

Ioannina, January 2009

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ
ΠΑΝΕΠΙΣΤΗΜΙΟ ΙΩΑΝΝΙΝΩΝ
DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF IOANNINA

Γρηγορη και Αξιοπιστη Αποθηκευση Ροων με Διαφορικη Καταγραφη Δοσοληψιων Δεδομενων

Η ΜΕΤΑΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ ΕΞΕΙΔΙΚΕΥΣΗΣ

υποβάλλεται στην

ορισθείσα από την Γενική Συνέλευση Ειδικής Σύνθεσης

του Τμήματος Πληροφορικής Εξεταστική Επιτροπή

από την

Ανδρομάχη Χατζηελευθερίου

ως μέρος των Υποχρεώσεων για τη λήψη του

ΜΕΤΑΠΤΥΧΙΑΚΟΥ ΔΙΠΛΩΜΑΤΟΣ ΣΤΗΝ ΠΛΗΡΟΦΟΡΙΚΗ

ΜΕ ΕΞΕΙΔΙΚΕΥΣΗ

ΣΤΑ ΥΠΟΛΟΓΙΣΤΙΚΑ ΣΥΣΤΗΜΑΤΑ

Ιανουάριος 2009

Dedication

To my family and NP...

Acknowledgements

At this point, I would like to thank all those people, who each in his way have helped for the successful completion of this thesis.

I am mostly grateful to Prof. Stergios Anastasiadis for his systematic supervision and guidance throughout this research, from the early stages of the design, to the last ones of the composition of this thesis. During all these months, through his core knowledge on the field of computer systems, he gave me the opportunity to indulge in this particular area of knowledge.

The deepest gratitude to my parents for everything they have done, from the early stages of my education till this point, and especially for the moral and financial support they provided me, and the tolerance they have shown during all these years. I am really grateful for their continuous encouragement.

I would like to thank all the people of the Systems Research Group *(SRG)* at the University of Ioannina, who turned the endless hours of study into a joyful experience. Especially Lamprini Konsta and George Margaritis, through numerous hours of discussions, provided me useful feedback at several checkpoints of my thesis. Special thanks to Nikolaos Papanikos for all the help and encouragement, and primarily for all his valuable tolerance during the months that passed.

Finally, it should be noted that all the work presented in this thesis was in part supported by the project INTERSAFE with approval number 303090/YD7631 of the INTERREG IIIA Greece-Albania 2000-20006 neighboring program.

Table of Contents

List of Figures

List of Tables

Abstract

Andromachi T. Hatzieleftheriou, MSc, Computer Science Department, University of Ioannina, Greece. January, 2008. Fast and Reliable Stream Storage Through Differential Data Journaling.

Thesis Supervisor: Stergios V. Anastasiadis.

Real-time storage of massive stream data is emerging as a critical component in modern computing infrastructures used for continuous monitoring purposes. Traditional file and database systems are not designed for such operation environments and incur excessive resource requirements when handling high-volume streaming traffic.

In this thesis, we examine the possibility of employing data journaling techniques in order to combine sequential throughput with low latency during synchronous writes. Experimentally we demonstrate that low-rate streams incur remarkably high data journaling traffic in a commonly used production file system. Therefore, to alleviate the problem we introduce differential data journaling in a prototype subsystem that we have designed and implemented for a widely available operating system. Through extensive experimentation, we show that our implementation achieves substantial reduction in the required disk throughput combined with very low write latency.

Εκτεταμενη Περιληψη

Ανδρομάχη Χατζηελευθερίου του Θωμά και της Φωτεινής. MSc, Τμήμα Πληροφορικής, Πανεπιστήμιο Ιωαννίνων, Ιανουάριος, 2008. Γρήγορη και Αξιόπιστη Αποθήκευση Ροών με Διαφορική Καταγραφή Δοσοληψιών Δεδομένων.

Επιβλέπων: Στέργιος Αναστασιάδης.

Η αποθήκευση μεγάλου όγκου ροών δεδομένων σε πραγματικό χρόνο αποτελεί βασική υπηρεσία των σύγχρονων συστημάτων υπολογιστών, κυρίως σε περιπτώσεις εφαρμογών παρακολούθησης. Τέτοιες εφαρμογές χρησιμοποιούνται ευρέως στις μέρες μας για τη διαχείριση υπολογιστικών υποδομών και την προστασία φυσικών χώρων.

Σύμφωνα με προηγούμενες εργασίες, τα παραδοσιακά συστήματα διαχείρισης δεδομένων, όπως είναι τα συστήματα αρχείων γενικού σκοπού και οι σχεσιακές βάσεις δεδομένων, δεν επαρκούν για την αποθήκευση ροών που παράγονται με συνεχή ρυθμό από αισθητήρες σε πραγματικό χρόνο. Στη γενική περίπτωση, ένα σύστημα παρακολούθησης λαμβάνει συνεχώς νέα δεδομένα από ένα μεγάλο πλήθος συνδέσεων-αισθητήρων και τα αποθηκεύει για κάποιο χρονικό διάστημα, το οποίο εξαρτάται από το είδος της επεξεργασίας στην οποία πρόκειται να υποβληθούν. Οι αισθητήρες μπορούν, για παράδειγμα, να παράγουν βίντεο και ήχο υψηλής ποιότητας με υψηλό ρυθμό μετάδοσης, ή να στέλνουν περιοδικά πληροφορίες για τη διακύμανση κλιματολογικών συνθηκών με πολύ χαμηλότερο ρυθμό. Κάτω από αυτές τις ετερογενείς συνθήκες, προκύπτει η ανάγκη για ένα σύστημα ικανό να αποθηκεύει αξιόπιστα την εισερχόμενη ροή, χωρίς παράλληλα να επηρεάζει την ακολουθιακή αναπαραγωγή των δεδομένων που λαμβάνει.

Τα σύγχρονα συστήματα αρχείων εφαρμόζουν τεχνικές καταγραφής δοσοληψιών (*journaling*) προκειμένου να βελτιώσουν το βαθμό αξιοπιστίας που προσφέρουν. Βασικό γνώρισμα αυτής της μεθόδου είναι ότι επιτρέπει τη μεταφορά των δεδομένων ή των μεταδεδομένων από τη μνήμη στο δίσκο σύγχρονα με ακολουθιακό τρόπο. Έτσι, αναβάλλεται προσωρινά

η χρονοβόρα μετακίνηση των δεδομένων ή των μεταδεδομένων στην τελική τους θέση στο δίσκο, ενώ ταυτόχρονα μειώνεται η καθυστέρηση εγγραφής που γίνεται αντιληπτή από την εκάστοτε εφαρμογή. Κατά κύριο λόγο, οι τεχνικές αυτές εφαρμόζονται στα μεταδεδομένα του συστήματος, ενώ κάποια συστήματα αρχείων επιπρόσθετα υποστηρίζουν καταγραφή δοσοληψιών στα δεδομένα που τροποποιούνται (*data journaling*). Σχετική έρευνα έχει δείξει ότι μέσω της καταγραφής δοσοληψιών δεδομένων, μπορούν να εξυπηρετηθούν αιτήσεις εγγραφής τυχαίας προσπέλασης με ακολουθιακή απόδοση δίσκου. Αντίθετα, σε περιπτώσεις μεγάλων αιτήσεων εγγραφής ακολουθιακής προσπέλασης έχει παρατηρηθεί ότι η τεχνική αυτή μειώνει την απόδοση του δίσκου, καθώς αυξάνεται σημαντικά η κίνηση στο αποθηκευτικό μέσο. Στην περίπτωση που μελετάμε, βασικό μας μέλημα είναι η αξιόπιστη και αποδοτική αποθήκευση πολλαπλών εισερχόμενων ροών, των οποίων η συνολική συμπεριφορά είναι τυχαίας προσπέλασης, παρόλο που καθεμία γράφει ακολουθιακά σε κάποιο ξεχωριστό αρχείο. Σε τέτοια περιβάλλοντα παραμένει αδιευκρίνιστο ποιά είναι η καταλληλότερη μέθοδος για τη διαχείριση της εισερχόμενης ροής.

Στην παρούσα εργασία, μελετάμε τη συμπεριφορά της καταγραφής δοσοληψιών δεδομένων στα πλαίσια των σύγχρονων αιτήσεων εγγραφής σε συστήματα αρχείων. Ένα βασικό μειονέκτημα αυτής της μεθόδου είναι ότι επιφέρει σημαντικό κόστος σε εύρος ζώνης δίσκου, λόγω του υψηλού όγκου των δεδομένων που στέλνονται για αποθήκευση. Προκειμένου να ελαττώσουμε τις απαιτήσεις σε εύρος ζώνης, υλοποιήσαμε μια νέα μέθοδο καταγραφής δοσοληψιών δεδομένων που αποθηκεύει μόνο την πραγματική μεταβολή στα δεδομένα ως αποτέλεσμα των αιτήσεων εγγραφής του χρήστη. Υλοποιήσαμε την προτεινόμενη μέθοδο στο προκαθορισμένο σύστημα αρχείων *ext3* του πυρήνα του λειτουργικού συστήματος *Linux*. Με λεπτομερείς πειραματικές μετρήσεις δείχνουμε ότι ανάλογα με το ρυθμό μετάδοσης των ροών, μπορούμε να μειώσουμε σημαντικά τις απαιτήσεις σε εύρος ζώνης της καταγραφής δοσοληψιών δεδομένων. Ταυτόχρονα, πετυχαίνουμε μια σημαντική μείωση στο χρόνο απόκρισης των σύγχρονων αιτήσεων εγγραφής του συστήματος αρχείων. Συνολικά, η μέθοδος που προτείνουμε είναι ικανή να προσφέρει γρήγορη και αξιόπιστη αποθήκευση, τόσο σε περιπτώσεις ροών δεδομένων, όσο και σε παραδοσιακές εφαρμογές που απαιτούν σύγχρονες εγγραφές για την αξιόπιστη αποθήκευση των δεδομένων τους.

Chapter 1

Introduction

1.1 Thesis Scope

Continuous monitoring processes are prevalent today for a wide range of purposes such as network administration, autonomic systems management and physical site safety. Such important applications make stream-oriented functionality highly relevant in modern computing infrastructures. For instance, recently proposed stream management engines demonstrate the feasibility of flexibly applying time-series operators on high-rate streams [3, 19]. Existing stream processing environments store stream data either temporarily before applying real-time operators within time windows [7], or permanently in order to support retrospective query processing [10].

Prior research has made the case that traditional data management approaches, such as relational databases and general-purpose file systems, are not engineered to efficiently store continuous stream data that are automatically generated from sensors in real time [7, 10]. Sensors may generate high-resolution video and audio streams at high rates [11], or send intermittent variations of environmental conditions at much lower rates [22]. A monitoring system receives messages from high-volume links or large numbers of sensors and stores the received data for a time period that depends on whether the applied processing occurs in real time or retroactively.

Across all types of heterogeneous streams with different rate and content characteristics, it would be desirable to store the received data reliably on the same facility without compromising the sequential playback performance required for statistical processing or effective visualization. Thus, a stream storage facility could serve as a building block for a variety of applications in the entire range from network packet processing to urban traffic control or environmental monitoring with the appropriate indexing functionality built separately at a higher level, when support for query processing is required.

In general, file system operations are either data operations that update user data, or metadata operations that modify the structure of the file system itself. Existing general-purpose file systems use journaling in order to synchronously move data or metadata from memory to disk in a sequential manner. Thus they postpone the more costly transfer of data or metadata to the disk location without penalizing the write latency perceived by the application user. Indeed, previous research has used trace-based emulation to experimentally demonstrate that data journaling can serve random writes with high sequential throughput, but actually makes throughput lower at high data volumes due to the extra disk traffic generated [25]. The study made the reasonable conclusion that data journaling should only be enabled with random writes, but disabled with large sequential writes. Instead, we focus on the efficient and reliable storage of multiple concurrent streams whose aggregate workload demonstrates random-access behavior even though appends corresponding to individual streams may be perfectly sequential. To a large extent, in such environments it remains unclear what is the most appropriate way to handle the incoming data.

In the present thesis, we investigate the performance characteristics of data journaling in the context of synchronous writes that would be required among several situations including the reliable storage of incoming streaming data. In order to lower the cost of data journaling, we introduce *differential data journaling*, that constitutes a differential version of the default data journaling mode of a widely used operating system. In particular, the primary idea of our approach is to journal only the bytes that are actually written rather than the entire corresponding blocks that contain them. Therefore, depending on the rate characteristics of the streams, we can reduce the required journaling throughput up to several factors. As a side-effect of the sequential writes to the journaling device, we also manage to substantially reduce the response time of synchronous writes. Thus, we can use data journaling to reduce the latency of writes at a reduced cost of required disk throughput.

1.2 Thesis Outline

The remainder of this thesis is organized as follows:

In Chapter 2, an overview of the related literature is presented. We review previous research related to techniques that have been proposed to provide file system reliability across system crashes and achieve high performance during data and metadata updates. Furthermore, we define the storage needs of applications that manage stream data, and present some of the most important implementations in this field. Finally, we present recent research related to redundancy elimination that intends to reduce the consumption of expensive resources, such as hard disk and memory space.

In Chapter 3, we describe an existing journaling method that is commonly used. In particular, we examine the journaling technique that the Ext3 file system applies in order to preserve metadata consistency across system failures, while minimizing the required recovery time.

In Chapter 4, the design goals of our study are defined and the general architectural decisions taken during our prototype implementation are justified.

In Chapter 5, we introduce the differential data journaling technique that we have designed and implemented for a widely available operating system. Our prototype is based on the idea of accumulating the modifications of multiple updates into a single journal block, and intends to minimize the write latency at a reduced disk throughput cost.

In Chapter 6, we explain the experimentation environment that we used in our study and present our measurements across different workloads. The experimental results are displayed graphically and our conclusions are justified.

In Chapter 7, the conclusions and the future directions of this thesis are outlined.

Chapter 2

Related Research

In this chapter, we describe approaches that have been previously proposed in order to achieve high performance in file systems during data and metadata updates. Furthermore, we review previous research that focuses on techniques which intend to provide file system reliability across system crashes. Next, we define the storage needs of streaming applications, and present some of the most important proposals in this direction. Finally, we present recent research related to redundancy elimination that intends to reduce the consumption of expensive resources, such as hard disk and memory space.

2.1 Fast and Reliable Storage Systems

File systems are central parts of modern operating systems and are expected to serve two opposing principles; performance and durability. Nevertheless, operating systems are still susceptible to hardware, software and power failures that damage both their efficiency and their reliability.

Early file systems introduced the use of a main memory buffer cache to hold writes until they are asynchronously written to disk. Those file systems suffered from potential corruption during a power failure or an operating system's crash, since recovery often required a time consuming examination of the entire state of the file system. Even today, during reboot, verifying a file system's consistency requires a special utility that recovers the file system's components to a consistent state. As disk sizes grow, this time can become a serious bottleneck, leaving the system offline for a considerable amount of time while the disk is scanned, checked and repaired. Although disk drives are becoming faster through time, this speed increase is modest compared with their enormous increase in capacity. Unfortunately, every doubling of disk capacity leads to a doubling of recovery time needed from traditional file systems checking techniques.

It is, however, possible to make file system recovery fast without sacrificing reliability and predictability. This is typically done by file systems which guarantee *atomic* completion of file system updates. The principal idea behind atomic updates is that an entire batch of updates can be written to the file system, but those updates do not take effect until a final commit update is made on the disk. In order to achieve this, the file system must keep both the old and the new contents of the updated data somewhere on disk until the final commit.

In order to predictably recover after a crash, the recovery phase must be able to work out what the file system was trying to do when the crash that led to incomplete operations to disk occurred. Consistent recovery of the metadata after a crash, due to operating system or power failure, requires the system updates to be written on disk in a specific order. There are many ways of achieving the required ordering between updates and we describe some of the most important in the rest of the present section.

2.1.1 Synchronous Writes

The system can achieve consistency simply by updating the system metadata synchronously. The synchronous metadata update mechanism first waits for the pending writes to complete, before submitting the next ones. Nonetheless, synchronous writes can significantly impair the ability of a file system to achieve high performance as it is not feasible to batch up multiple updates into a single disk operation. Similarly one can recover recently writ-

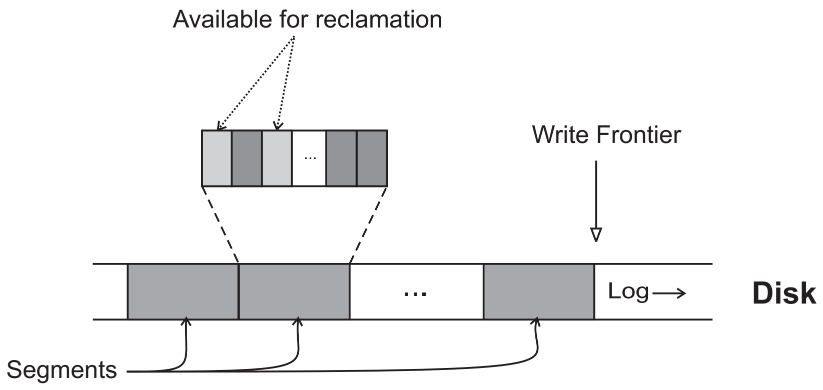


Figure 2.1: A log-structured file system treats its storage as a circular log and writes all data and metadata modifications sequentially to the head of a segmented append-only log. Log space must be constantly reclaimed and thus, a garbage collecting process is responsible for coalescing unused space into empty segments.

ten data after a crash by writing them synchronously to disk. Synchronous data writes are typically applied in database systems that store critical data [31, 8].

Xsyncfs introduces the idea of externally synchronous I/O that guarantees durability not to the application, but to the external entity that observes application output [23]. In particular, an externally synchronous system call returns control to the application before committing data. Subsequently, all output that causally depends on the uncommitted transaction is buffered, and is eventually externalized only after the commitment is successfully completed. However, in the case of applications that do not produce any output, xsyncfs commits data periodically similarly to an asynchronously mounted journaling file system, an approach that is described later in this section.

2.1.2 Log-Structured File Systems

The main idea behind the design of a log-structured file system (LFS) is to improve write performance by buffering a sequence of file system updates in the file cache and then writing all the changes to disk sequentially in a single disk write operation [27]. For this reason, a log-structured file system treats the disk as a segmented append-only log and writes all data and metadata modifications into it. The log is the only structure on disk and consists of segments that facilitate the removal of deleted areas (Figure 2.1).

Periodically, the system writes the complete and consistent file structures safely at a fixed location of the log called checkpoint region. After a crash, the file system uses the checkpoint for its initialization, and the recent portion of the log to quickly recover recently written data. In particular, upon its next mount, the file system does not need to walk all its data structures to fix any inconsistencies, but can reconstruct its state from the last consistent point in the log.

Free space must be constantly reclaimed from the tail of the log to prevent the file system from becoming full when the head of the log wraps around to meet it. When updated data is written to the end of the log, the previous copy of the data is still on disk in its old location and can be considered as dead space or a hole in the log. A garbage collecting process is responsible for coalescing these holes into empty segments which are then available for new log writes. The tail itself can skip forward over data for which newer versions exist farther ahead in the log; the remainder is simply moved out of the way by appending it back to the head.

Log-structured file systems maximize the write throughput on magnetic media by avoiding costly seeks. In addition, interleaved writes to multiple streams can be allocated closely together on disk. However, log-structured file systems induce cleaning overhead, since the size of the file system is of finite size and the log must eventually wrap around. Although write allocation in log-structured file systems is straightforward, the garbage collection of storage space after files are deleted, has remained problematic. Cleaning in a general purpose LFS must handle files of vastly different sizes and lifetimes, and all existing solutions involve copying data to avoid fragmentation. Previous study verified this high cleaning overhead, particularly under OLTP-like workloads, where small random writes make up a large portion of the disk I/O requests [28]. Over the last years, many algorithms have been proposed to reduce the cleaning cost of LFS, but the cleaning cost is still high in systems with high disk space utilization and little idle time.

A number of file systems have been implemented based on this design, including the Sprite LFS [27] and some prototype LFS implementations on Linux. HyLog uses a log-structured layout for hot pages to achieve high write performance, and overwrite strategy for cold pages to reduce the cleaning cost [32]. DualFS is a recent implementation based on a variation of log-structured file systems [24]. It uses two separates devices for the data and metadata, respectively; it employs a log-structured file system for the metadata and treats data as in typical Unix systems. We present another variation of LFS called StreamFS in Section 2.2.2, where all writes take place at a write frontier which advances as data is written [10]. StreamFS does not require a segment cleaner, and applies a prototype expiration policy in order to selectively overwrite the stored data.

2.1.3 Soft Updates

Soft updates is a mechanism that delays writes of metadata and explicitly maintains dependency information to specify the order in which data must be written to disk [13]. Thus, it eliminates the need for a log or most synchronous writes related to metadata. The system maintains for each disk block a list of all the metadata dependencies associated with the block. When a block needs to be written, which block requires other blocks to be written first, the system rolls back the affected parts of the selected block to their earlier state. After the write has completed, the system deletes all the completed dependencies and restores the block to its current value. Thus, applications see the most recent version of the metadata blocks and the system keeps disk contents consistent. After system crashes the system can be mounted and used immediately, since the only remaining inconsistencies are non-fatal errors that can be corrected in the background during normal operation.

Soft updates track and enforce metadata update dependencies, so that the file system can safely delay writes for most file operations. This method improves system performance because it aggregates multiple metadata updates into a reduced number of disk writes and postpones time-consuming operations, such as deletes, to a background process.

2.1.4 Journaling File Systems

Journaling file systems use an auxiliary log to record all *metadata* operations and ensure that the log and data buffers are synchronized in a way that guarantees recoverability. Additionally, some implementations also support logging of *data* modifications. The goal of a journaling file system is to avoid running time-consuming consistency checks on the whole file system, by looking instead in the log that contains the most recent disk write operations. Consequently, remounting a journaling file system after a system failure is a matter of a few seconds.

A journaling file system maintains a journal of the updates it intends to make, ahead of time. The log is maintained as a preallocated file within the same file system or as

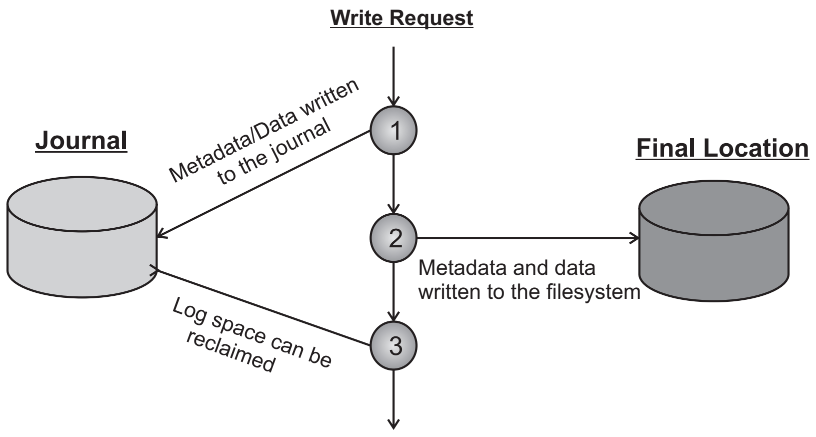

Write Request
Journal
Final Location
Metadata/Data written to the journal
1
2
Metadata and data written to the filesystem
Log space can be reclaimed
3

Figure 2.2: A journaling file system logs updates to a circular journal file before committing them to the main file system. Once the corresponding updates has been stored to their final location, copies of the blocks in the journal can be discarded allowing the journal space to be reclaimed.

a standalone separate file system. After a crash, recovery simply involves replaying the updates from the journal until the file system is consistent again. A file system transaction, which consists of a sequence of correlative updates, is marked as complete when it is journaled and followed by a commit record. Only then the corresponding updates can be written to their final location (Figure 2.2). Journaling file systems guarantee atomicity during recovery, as all the updates of a transaction can either be rejected or replayed, according to whether or not the transaction is followed by a commit record in the journal.

Through write-ahead logging the journaling file systems ensure that the log is written to disk before any pages containing data modified by the corresponding operations. Even though the system performs additional disk operations, they are efficient since they are sequential. Batching of log writes that originate from different concurrent applications, provides additional throughput improvements. In addition, file system journaling allows synchronous writes to complete faster, because they return as soon as the sequential log update completes. Therefore, costly disk operations at the final locations of the modified blocks can be deferred and completed periodically and asynchronously.

Journaling of file data helps further in that direction, but incurs significant extra throughput on the journaling device. The cost of data journaling can be high for large writes due to the significant volume of data sent to the log. Unfortunately, current implementations incur considerable logging activity even with small writes. In order to simplify the implementation, they log the entire blocks being modified rather than just their modified part. However, journaling reduces write latency in both small and large writes, since it allows the synchronous log updates to be completed sequentially.

The data and metadata journaling of the Ext3 file system has been documented[29, 12]. yFS is a recently proposed file system for general purposes that only uses journal transactions for metadata modifications [33], while it reduces disk seeking and handles large files efficiently. Earlier, Hagmann described metadata update logging in the Cedar File System to improve performance and achieve consistency [16]. In order to gain performance, it used group commit, a concept derived from high performance database systems. Also, the Echo distributed file system used a journal to record disk storage updates thus improving performance and availability [5].

Prabhakaran et al. introduced the semantic block-level analysis technique to trace and analyze file systems, and the semantic trace playback technique to evaluate file system modifications [25]. Evaluation of Ext3 over Linux showed that data journaling incurs substantial traffic to the journal but with sequential throughput, unlike the ordered mode that mainly writes data to the final location. The authors conclude that sequential workloads should better be served in ordered mode, while random workloads can benefit from data journaling. Using trace-based emulation, the authors show that differential data journaling can reduce substantially the amount of traffic to the journal in database applications.

2.1.5 Persistent Memory

There exist approaches that implement some type of stable storage through specialized hardware. The memory vulnerability to power outages can be encountered using uninterruptible power supply or a distinct Flash RAM device. Thus, writes to the final on-disk location can be deferred to a later more convenient time, when the memory space needs to be reclaimed for example. However, the main drawback of such implementations is the extra hardware expenses.

The Rio file cache makes ordinary memory safe for persistent storage, through the use of an uninterruptible power supply, that allows the file system to avoid synchronous writes and guarantee the file system consistency at the same time [8]. However, durability is guaranteed only as long as the power in on or the batteries remain charged.

Another approach, the Network Appliance's WAFL (Write Anywhere File Layout) file system checkpoints the disk to a consistent state periodically and uses Non-Volatile RAM (NVRAM) for fast writes between checkpoints [18]. NVRAM is used to keep a log of NFS requests that WAFL has processed since the last consistency point. WAFL keeps the new copies of the updated data in different locations from the old copies, and eventually reuses the old space once the updates are committed to disk. After an unclean shutdown, it replays any requests in the log to prevent them from being lost. The Write Anywhere File Layout 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 NVRAM. Nevertheless, NVRAM is characterized by capacity, reliability and cost limitations.

2.1.6 Other Implementations

Hildebrand et al. highlight the prevalence of small and sequential data requests in scientific applications [17]. They show that it is possible to improve the overall write performance of parallel file systems by using parallel I/O for large write requests and a distributed file system for small write requests. The Virtual Log is another effort to minimize the latency of small synchronous writes by building the log-structured file system over a log with entries that are not necessarily physically contiguous [31]. Virtual Log is an approach to improve small disk write performance even in systems with no idle periods, but it requires detailed knowledge of the disk layout and the location of the disk head at any moment, which might be difficult to obtain from modern disks. Finally, the Google File System handles large files typically mutated by appending new data sequentially rather than overwriting existing data, at random file locations [14].

2.2 Stream Archival Servers

Recently a new class of data-intensive applications has become widely recognized; streaming data management applications. This class includes financial applications, network monitoring, security, telecommunications data management, web applications, manufacturing and sensor networks. In the data stream model, individual data items may be relational tuples, e.g., network measurements, call records, web page visits, sensor readings, and so on. However, their continuous arrival in multiple, rapid and time-varying streams yields some fundamentally new research problems.

In particular, data arrival rates which can vary from hundreds of thousands of packets per second per link to much lower rates, complicate the storage management for such applications. Currently, the design of a streaming-oriented storage system can be based on two possible architectures; either a relational database can be used to store the incoming stream data, or a custom index can be built on top of a conventional file system. Nonetheless, at the above mentioned heterogeneous data rates, both common database index structures and general-purpose file systems have been documented to perform poorly [7, 10, 2]. This motivates the need for a new storage system, that runs on commodity hardware and is specifically designed to satisfy the storage needs of streaming data.

2.2.1 Traditional Databases

Nowadays, network monitoring systems are useful for a multitude of purposes, such as physical site safety, network and security forensics. Monitoring applications differ substantially from conventional business data processing. Traditional Database Management Systems (DBMS) have been oriented toward business data processing, and consequently are designed to address the needs of these applications [7]. Particularly, a DBMS is considered to be a passive repository storing a large collection of data elements and typically only humans initiate queries and transactions on this repository. Furthermore, traditional DBMSs are not designed for rapid and continuous loading of individual data items, and they do not directly support the continuous queries that are typical of data stream applications. Finally, a DBMS assumes that applications require no real-time services.

Applications that continuously monitor and store massive numbers of streams in real-time could benefit from DBMSs, due to the high volume of monitored data and the query requirements that arise. However, traditional DBMSs seem to have remarkable inefficiencies under such circumstances. First, monitoring applications continuously receive high volumes of data from external sources, such as sensors, rather than from humans issuing transactions. Moreover, while for a DBMS data do not have a notion of time and any update operation overwrites the previous value, data stream represent a sequence of values for the same entity. Thus, the static model of databases, with dynamically changing queries being executed over static data, is not designed for handling stream data, which has static queries being executed over dynamically changing data. Last but not least, handling data streams would require the DBMS to serve real-time applications, making it imperative that the DBMS employ intelligent resource management (e.g., scheduling) and graceful degradation strategies (e.g., load shedding) during periods of high load. These are not features of a traditional DBMS which is designed as a store-and-query model instead.

Digital streaming infrastructures replace traditional closed-circuit television systems in urban traffic-control applications to store large numbers of video feeds [11]. Previously, environmental, oceanographic and meteorological conditions have been measured and stored over distributed relational databases [22]. Aurora is a stream processing engine that has been developed to support primitives for streaming applications, handle query processing on incoming messages in real time and gracefully deal with spikes in message load [7, 3]. The CoMo is a passive monitoring system that can be used as a building block for a network monitoring infrastructure that processes and shares network traffic statistics over multiple sites [19]. Como includes a storage process that is data agnostic and treats all data blocks equally. Also, load shedding techniques were developed to maintain the accuracy of traffic queries within acceptable levels at extreme traffic conditions [4].

2.2.2 General-Purpose File Systems

The storage needs of monitoring applications result in continuous sequential writes to the underlying storage system. In order to reduce disk seek overheads and improve system throughput, the system should employ data placement techniques that exploit the particular I/O characteristics of streams. General-purpose file systems are not engineered to efficiently store continuous stream data that are automatically generated from sensors in real time. Unix-like file systems, for instance, are typically optimized for writing small files and reading large ones sequentially, while monitoring and querying applications either write very large files at high data rates, or apply small writes at much lower rates, while issuing small reads.

File systems periodically write data to disk and transaction processing applications view transactions as committed only after the data has been written to disk. A modified version of the log-structured file system has been recently used for the storage of high-volume streams [10]. StreamFS has incoming stream data written to a frontier that moves in a circular fashion along the disk space and selectively overwrites the expired data. However, StreamFS has been specifically designed for high-rate streams typically generated in network monitoring systems; it is unclear how it would behave in heterogeneous environments where high-rate and low-rate streams co-exist. Additionally, an aggregate high-rate stream typically contains a large volume of information that makes necessary to build an index structure online during data storage and scan entire segments of the stored data during retrospective query processing. Instead, demultiplexing of the incoming data into separate files would possibly facilitate and reduce the load of the subsequent selective retrieval and processing.

In order to improve their operation reliability, recent general-purpose file systems apply journaling techniques to preserve metadata consistency across system crashes at minimal recovery time. Such techniques are therefore in high demand, especially, in environments where high availability is important, not only to improve recovery times on single machines, but also to allow a crashed machine's file system to be recovered on another machine when we have a cluster of nodes with a shared disk. Comparisons across different journaling methods with general-purpose file server traffic has shown that, depending on the sequentiality workload characteristics, either ordered data writing or data journaling may lead to better performance [25]. Nevertheless, the problem is that the block access sequence on a content server is effectively random when many slow streams access large files concurrently, even though individual stream appends are perfectly sequential [1]. Therefore, it might be useful to build system facilities for the storage of heterogeneous streams with different rate and content characteristics.

2.2.3 Playback Servers

Several research projects and commercial products of media streaming servers have already established the feasibility of streaming stored files. Recent years have witnessed an ever-increasing demand for media-on-demand applications on the Internet. Typically, users access online media clips by clicking on a hyperlink using their Web browser, which results in the browser opening a media player to play the selected media file. The playback servers are responsible to deliver the selected media file to the player through *streaming*. In the streaming mode of data delivery, the initial portion of the media is loaded into the player buffer, which takes a brief time period. The remainder of the content is obtained across the network, while the media file is being played back. A stream file is received, processed, and played simultaneously and immediately, leaving behind no residual copy of the content on the receiving device.

Therefore, the main purpose of a playback server is to read from disk the required stored stream file, and then deliver it to the proper client. Reading a stream file from the disk refers to finding and retrieving the blocks that contain the requested data. Additionally, read-ahead techniques are applied in order to enhance disk performance. Read-ahead consists of reading several adjacent pages of data of a file from disk, before they are actually requested. On the other hand, streaming storage deals with the stream files' write operations. Thus, the basic challenge of a streaming storage server is to quickly, reliably and efficiently, in terms of disk throughput, store the incoming data. Write operations on disk-based stream files are slightly more complicated, since special care must be taken in order to avoid compromising their sequential playback performance.

Streaming workloads differ from traditional web workloads in many respects, presenting a number of challenges to system designers and media service providers. For instance, transmitting media files requires more computing power, bandwidth and storage and is more sensitive to network jitter than web objects. Furthermore, media access lasts for a much longer period of time and allows for user interaction.

In particular, although proxy caching has been successful in delivering static text-based content, it is more difficult to deliver streaming media content. First, the size of a media object is generally much larger than a text-based object, rendering the caching of entire media objects as static objects inefficient. Furthermore, a client requesting some media object demands continuous streaming delivery. While, the occasional delays that occur when transferring data over the Internet are acceptable for text-based Web browsing, for streaming media data this transfer delay results in undesirable playback jitter at the client side.

Instead, whole-file transfers, or file downloading can provide continuous playback, but it introduces a significant startup delay, in addition to large buffer space requirements on the client. In comparison to traditional file downloading, media data streaming allows significantly faster playback initiation, provides guarantees for uninterrupted data decoding, and requires minimal buffering requirements from the client devices.

2.3 Redundancy Elimination

Several approaches have been proposed that intend to reduce the consumption of expensive resources, such as hard disk and memory space or transmission bandwidth. Reducing the number of required bytes is equivalent to the elimination of data redundancy within memory or the storage device. A number of techniques that have been proposed towards this effort include *data compression*, *duplicate suppression* and *delta encoding* methods. Particularly, data compression eliminates the redundancy inside an object, duplicate suppression refers to the elimination of identical objects and, finally delta encoding eliminates the redundancy between similar objects.

Significant improvements have occurred over the past decades in the field of virtualization. The main research interest lies in the multiplexing of hardware resources among virtual machines that run commodity operating systems, in order to reduce the host's management overhead. Nevertheless, main memory is not amenable to inexpensive multiplexing and thus a variety of redundancy elimination techniques, such as page sharing of identical pages, memory compression inside individual pages and delta encoding between similar pages, are performed to achieve high memory consolidation. Related study shows that substantial memory savings are available from the sharing of identical pages between virtual machines when running homogeneous workloads [30]. The Difference Engine, an extension to the Xen virtual machine monitor, demonstrates the potential memory savings available from leveraging a combination of whole page and sub-page sharing and memory compression [15].

Kulkarni et al. exploited similarity at the block level in order to reduce the number of bytes needed to represent an object when it is stored [21]. In particular, they proposed the use of compression, duplicate block suppression and delta encoding to eliminate redundancy of stored data in a scalable and efficient way. Finally, Venti is a network-based storage system intended primarily for archival purposes [26]. This approach enforces a write-once policy, preventing accidental or malicious destruction of data, while duplicate copies of a block can be coalesced in order to reduce the consumption of storage.

2.4 Summary

The prevalence of continuous monitoring processes for system management purposes and general physical site safety make stream processing applications highly relevant in modern computing infrastructures. Prior research has made the case that neither traditional databases, nor general-purpose file systems are sufficiently engineered to efficiently store continuous stream data that is automatically generated from sensors in real time.

Furthermore, current file systems mostly care to maintain their integrity across crashes without compromising their performance. They achieve this goal by flushing *metadata* updates at sequential disk throughput or by avoiding the violation of the dependencies across the block updates. Existing techniques that complete the *data* updates synchronously, require significant extra disk throughput in order to achieve that at relatively low latency. This overhead comes from the large amounts of data that needs to be written to disk, even in cases of small updates. However, a number of effective techniques have been proposed over the last decades, in order to reduce the consumption of expensive resources, such as memory and disk space.

In this thesis, we reconsider the ability of conventional file systems to serve the needs of streaming workloads, and towards this direction we modify a widely available file system in order to alleviate its relevant design inefficiencies. At the same time, we demonstrate that it is possible to reduce substantially the throughput overhead of synchronous data writes while maintaining low latencies, as well.

Chapter 3

Journaling in the Ext3 File System

Journaling results in noticeable reduction of the time period spent during the recovery of a file system to a consistent state after a crash. In this chapter, we analyze the popular Linux journaling file system, Ext3 [29, 12]. In particular, we examine the journaling techniques that are applied, in order to achieve high consistency guarantees across system crashes at minimal recovery time, and detect design inefficiencies that incur significant performance overhead to the journal device.

3.1 Background

As disk capacities grow faster than disk access speeds over time, modern file systems use journaling to support fast recovery after a crash [29, 12, 6, 25]. Journaling reduces possible downtime of several hours to a few seconds by avoiding running time-consuming consistency checks over the entire capacity of the file system. Instead, it simply replays the most recent disk writes stored in the log. Ext3 implements journaling by performing each high-level change to the file system in two steps:

1. First, it copies the modified blocks into the journal.

2. Then, it transfers the modified blocks into their final disk location.

The journal is treated as a circular buffer; once the necessary information has been stored to its final location, copies of the blocks in the journal can be discarded allowing the journal space to be reclaimed.

3.1.1 Basic File System Concepts

A *file system* refers to a collection of files and file management structures on a physical or logical mass storage device. It describes a method of organizing blocks on a storage device into files and directories. The common file model used by the widely known Linux operating system is *object-oriented*. Object is a software construct that defines both a data structure and the methods that operate on it. It consists of the following object types:

- The *superblock object* that stores information relating to a mounted file system.

- The *i-node object* that stores information about a single file. Each i-node object is associated with an inode number that uniquely identifies the file within the file system.

- The *file object* that stores information concerning the relation between an open file and a process.

- The *dentry object* that stores information about the linking of a directory entry with the corresponding file.

The architecture depicted in Figure 3.1 illustrates the relationships between the major file system-related components in both user space and the Linux kernel. In particular, a *system call interface* layer provides the means to perform function calls from user space into the kernel. The Linux kernel contains a *Virtual File System* layer which provides a

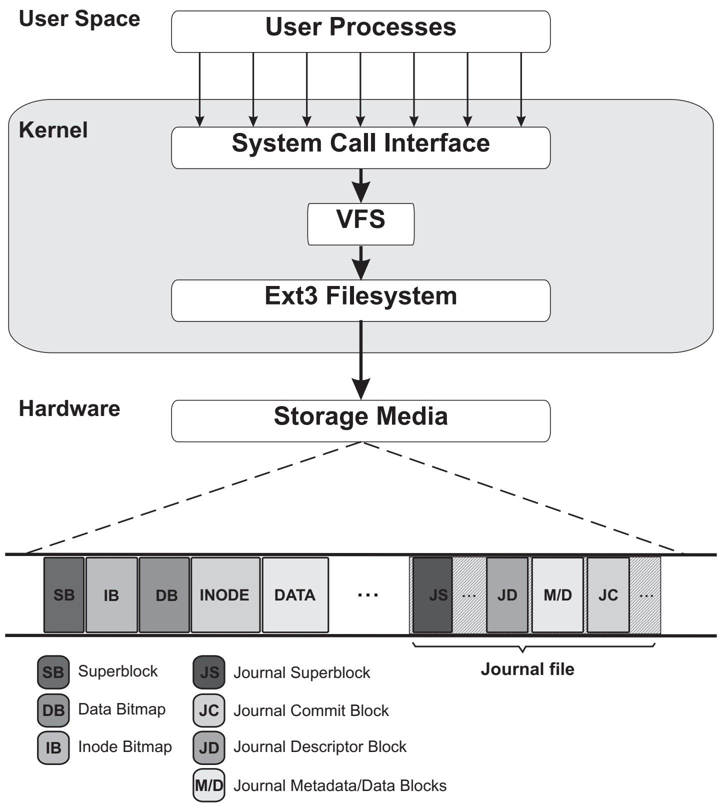


Figure 3.1: We illustrate the architectural view of the Linux operating system and distinguish the Ext3 file system inside the kernel. Furthermore, we figure the on-disk layout of the Ext3, which is based on the generic Unix file system structure.

common interface abstraction for file systems supported by the kernel. VFS constitutes an indirection layer which handles the file oriented system calls and calls the necessary functions in the physical file system code to do the appropriate I/O. Finally, the file system is responsible for applying the corresponding I/O requests on the proper devices.

3.1.2 Introduction to Ext3

The *Third Extended File System*, known as *Ext3*, is a journaling file system that is commonly used by the Linux operating system, and constitutes the default file system for the most recent Linux distributions. Ext3 is largely based on the Ext2 file system. Particularly, its on-disk layout is entirely compatible with the existing of an Ext2 file system with an additional disk structure, the journal file (Figure 3.1). Thus, all data and metadata updates are placed into the standard Ext2 structures that constitute the final location structures.

Information about pending file system updates is written to the journal. By forcing journal updates to disk before updating complex file system structures, this write-ahead logging technique enables efficient crash recovery. A simple scan of the journal and a redo of any incomplete committed operations are needed to recover the file system to a consistent state. The journal file is, by default, located within the file system, although it can be also stored on a separate device or partition. The journal is treated as a circular buffer and thus, once the necessary information has been written to its fixed on-disk location, the corresponding journal space can be reclaimed.

3.1.3 Journaling Modes

Ext3 uses three kinds of journaling; writeback, ordered and data journaling mode.

- In *writeback mode* Ext3 logs only the file system metadata, while data blocks are written directly to their fixed location. Although this mode is considered to be the fastest, it provides the weakest consistency guarantees of the three modes, since it does not enforce any ordering between the journal and the fixed-location data writes. Particularly, the contents of a file might be written before or after the journal is updated. As a result, files modified right before a crash can become corrupted. Thus, while metadata blocks are considered to be consistent, no guarantee is provided to the corresponding data blocks.

- In *ordered journaling mode*, only metadata writes are journaled. However, data writes to their fixed location are ordered right before the journal writes of the metadata, thus reducing the risk of corrupting data during recovery. In contrast to writeback mode, this mode provides more sensible consistency semantics, since data and metadata are guaranteed to be consistent after recovery. This is the default journaling mode on many Linux distributions.

- The *full data journaling* mode journals both metadata and data blocks. This mode minimizes the risk of losing file updates, but incurs additional disk accesses. It is considered to provide the strongest consistency guarantees of the three modes, while it seems to have different performance characteristics, in some cases worse, and surprisingly, in some cases better. In particular, the sequential nature of the journal can improve performance, while in other cases performance gets worse because each block is typically transferred to disk twice; once to the journal and then later to its final location. In the rest of this thesis, we prefer to use the term *data journaling* when we refer to the full data journaling mode in order to stress out the fact that it journals data in addition to metadata.

In our research, we focus on the efficient and reliable storage of multiple concurrent streams. Hence, we concentrate on the consistency guarantees provided through ordered and data journaling, since writeback mode offers the weakest consistency semantics of the three modes. However, for reasons of completeness, in our experimental measurements we examine the behavior of all the three modes.

Figure 3.2 depicts the behavior of three different journaling modes during the commit and the checkpoint intervals; the processes of updating the on-disk journal structure and the final on-disk location respectively. According to the mount options, the write updates are either written directly to their final on-disk location, or to the journal. Depending on the consistency semantics that each mode provides, the updates can take place synchronously or not. In particular, time flows downwards following the arrows, while boxes represent file system updates. Additionally, the two timelines represent commit and checkpoint time. As shown in Figure 3.2(a), during the commit time, the writeback mode writes synchronously metadata to the journal, while data blocks can be flushed asynchronously to their final location at any time. Thus, the required disk overhead is low since only metadata is logged. In Figure 3.2(a), the dotted boxes are used to imply that no ordering is required between data and metadata updates as they can occur in any order. Ordered journaling mode flushes data synchronously to the fixed location before the corresponding journal record is updated (Figure 3.2(b)). Next, when the proper time interval expires, metadata is finally written asynchronously to the appropriate fixed location. Consequently, a small amount of information (only metadata) is written to the journal sequentially and efficiently. However, synchronous data writes to the file system incur heavy disk traffic, which limits the system's performance for small writes. In data journaling the log is updated synchronously with both metadata and data records at each commit interval (Figure 3.2(c)). When the proper time interval expires, both metadata and data are finally written asynchronously to their fixed on-disk locations. Once again,

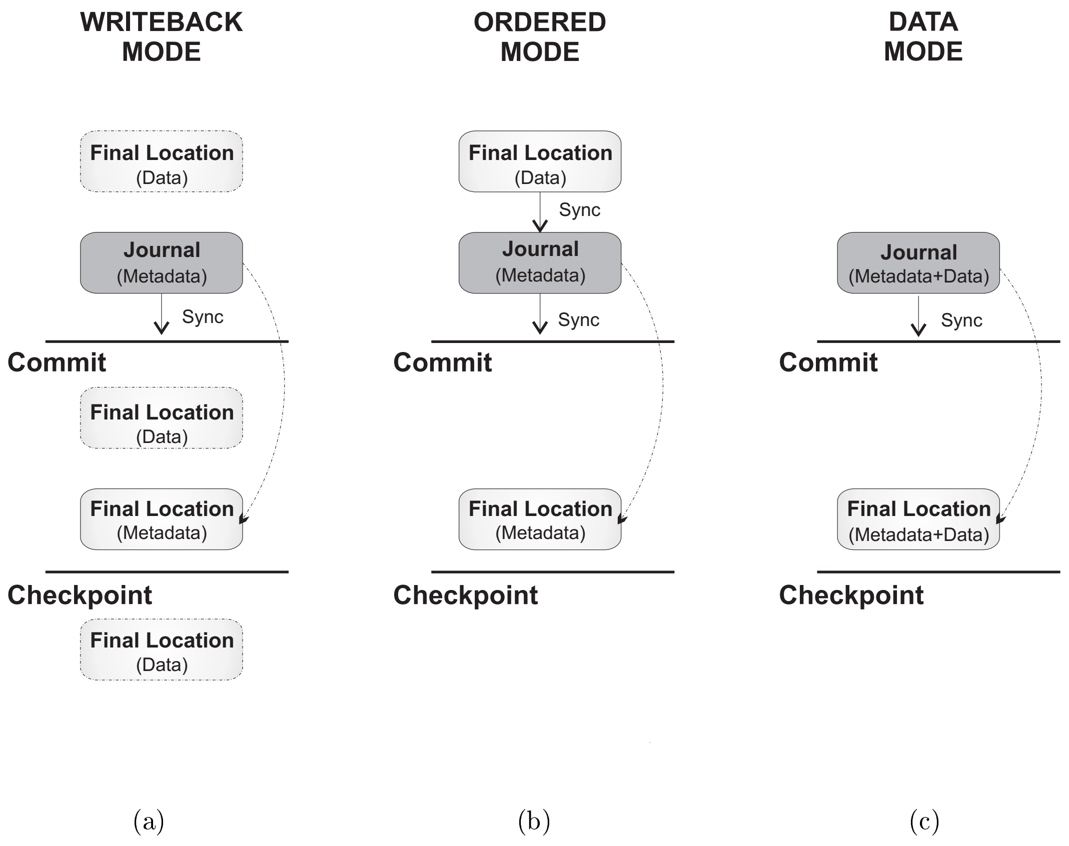


Figure 3.2: The behavior of the three different journaling modes through time. Time flows downwards following the arrows, while the boxes represent file system updates. The two timelines represent commit and checkpoint; the processes of updating the on-disk journal structure and the final on-disk location, accordingly. Depending on the consistency semantics that each mode provides, the updates can take place synchronously or not.

journal writes are efficient due to the append-only nature of the log. Nevertheless, when large volumes of data need to be written, the duplicates due to the journal writes impair the overall system's performance. Although journal writes negatively affect the performance of large data writes, small writes can benefit from the sequential journal. There, data modifications can be batched together while deferring their movement to the final location, thus reducing disk head seeking overhead.

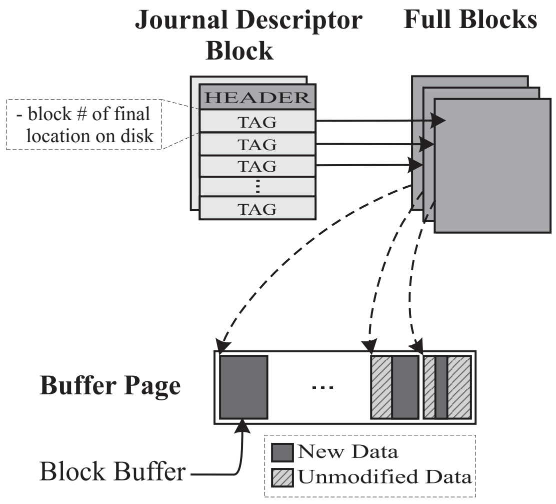


Figure 3.3: In the original design of the Ext3 data journaling, there is a full block in the journal for each write operation, despite the size of the new data modification. In addition, in the journal descriptor block a new auxiliary tag is allocated each time a write update is logged, and it is used to describe the correspondence between the journal and the fixed location disk block.

3.1.4 Journal

Ext3 handles the journal through a special kernel layer called *journaling block device* (JBD). The journal is implemented as either a hidden file within the root directory of the file system or a separate disk partition. Each *log record* in the journal corresponds to one low-level operation in the file system that updates one disk block. The journal represents with a log record the entire modified block of the file system rather than the range of block bytes actually modified (Figure 3.3). Thus, the journal is wasteful in terms of disk throughput and space, but simple in terms of processing complexity because it uses the buffers of the modified blocks directly. Additionally, each log record is associated with auxiliary information that contains the number of the corresponding block in the file system and several status flags.

As shown in Figure 3.4, Ext3 uses additional metadata structures to track the list of journaled blocks. The journal superblock tracks summary information for the journal, such as the block size and head and tail pointers. A journal descriptor block, as we explain later in this chapter, marks the beginning of a transaction and describes the subsequent journaled blocks, including their final fixed on-disk location. In data journaling mode, the descriptor block is followed by the data and metadata blocks; in ordered and

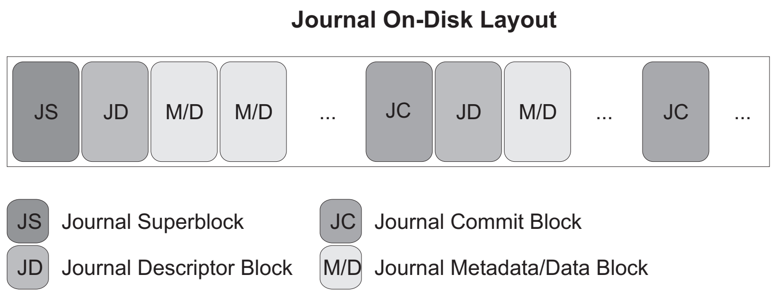


Figure 3.4: We illustrate the on-disk layout of the journal. The journal consists of a journal superblock, journal descriptor blocks, full data and metadata blocks, and journal commit blocks.

writeback mode, the descriptor block is followed by the metadata blocks. Finally, a journal commit block is written to the journal at the end of the transaction to mark its successful completion and verify that the corresponding data and metadata updates are safe on disk.

3.1.5 Transactions

Each high-level operation of the file system (e.g. a system call) is usually split into a series of low-level operations that manipulate disk data structures. The *atomic operation handle* refers to a set of low-level operations. When the system recovers from a failure, it ensures that either the whole high-level operation is applied, or none of its low-level operations is. For reasons of efficiency, instead of flushing each atomic handle to the journal, the system groups into a *single transaction* the records of multiple atomic operation handles. All the log records of a handle belong to one transaction. After its creation, the transaction accepts log records of new handles for a fixed period of time. The system stores all the log records of a transaction consecutively on the journal. After the log records have been committed to the file system, the system reclaims all the blocks of the transaction.

The JBD layer handles each transaction as a whole. A transaction is considered *complete* (equivalently in state **T_FINISHED**), if all its log records are fully residing in the journal including the commit block. It is *incomplete*, if at least one log record of the transaction is not in the journal. An incomplete transaction can be in one of the following states

T_RUNNING It still accepts new atomic operation handles.

T_LOCKED It does not accept new handles, but waits for the accepted handles to finish.

T_FLUSH All the handles in a transaction are complete and the transaction is being written to the journal.

T_COMMIT All the log records have been written to the journal except for the commit block of the transaction.

When recovering from a failure, the system skips all incomplete transactions and transfers the blocks of the complete transactions to the file system.

3.1.6 Kernel Buffers

The Linux kernel uses the *page cache* to temporarily keep page copies from recently accessed disk files in memory. In most cases, the kernel refers to the page cache when reading or writing from disk. In particular, before a file write occurs, the kernel verifies whether the corresponding page exists in the page cache. In case that it is found, the write is applied to that page in memory. Otherwise, when the write perfectly falls on page size boundaries, the page is not read from disk, but allocated and immediately marked as dirty. Otherwise, the corresponding page is fetched from disk and requested modifications are done. Pages that have been modified in memory for writing to disk, are marked dirty and have to be flushed to disk before they can be freed.

A *block buffer* is the buffer of an individual disk block in memory. As depicted in Figure 3.5, each block buffer has a *buffer head* descriptor that specifies all the necessary handling information required by the kernel in order to locate the corresponding block on disk. Generally, the page cache does not allocate the block buffers individually, but in units of pages called *buffer pages.* The kernel addresses individual blocks using the buffer heads pointed to by the corresponding buffer page.

3.1.7 Flushing Dirty Buffers to Disk

Write operations are deferred in the page cache. When data in the page cache is newer than the data on the backing store, that data is called dirty. Dirty pages that accumulate in memory eventually need to be written back to disk. Dirty page writeback occurs in two situations:

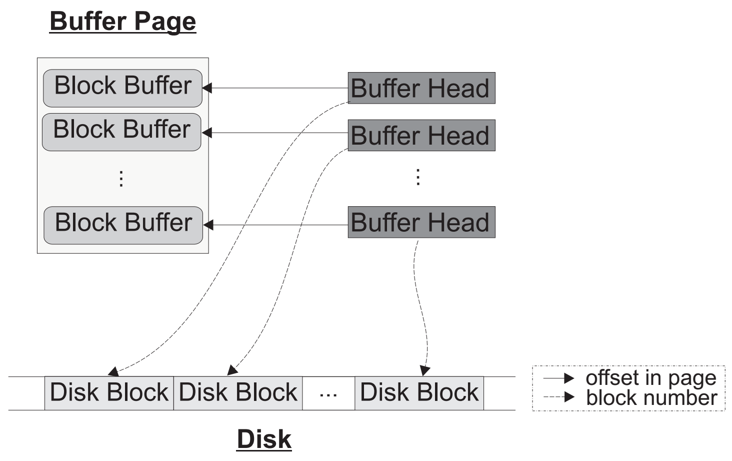


Figure 3.5: A buffer page is a page of data associated with special descriptors, called buffer heads. Their main purpose is to quickly locate the disk address of each individual block in the page.

- When free memory shrinks below a specified threshold, the kernel must write dirty data back to disk in order to free memory.
- When dirty data grows older than a specific threshold, sufficiently old data is written back to disk, in order to ensure that dirty data does not remain dirty indefinitely.

The Linux kernel uses a group of general purpose kernel threads called *pdflush* to systematically scan the page cache looking for dirty pages to flush, and additionally, ensure that no page remains dirty for too long.

Therefore, a number of pdflush kernel threads flush dirty pages to their final location on disk through two separate mechanisms:

- Systematically scan the page cache every *writeback period.*
- Implement a timeout mechanism on each page according to a configurable *expiration period.*

Furthermore, the JBD layer uses an additional kernel thread, known as *kjournald* thread. This kernel thread is responsible for two things:

- Every so often the current state of the file system needs to be committed to the journal on disk. This happens periodically and the corresponding time interval is known as *commit interval.*

- The dirty buffers of the committed transactions need to be flushed periodically to the final on-disk location, in order to reclaim space in the log.

A user can also use the *fsync* system call to synchronously flush all the data and metadata dirty buffers of the specified file descriptor to disk. Actually, fsync moves the blocks to the journal or the final disk location depending on the mount mode.

3.2 Commit Policy

The *commit* of a transaction involves writing to journal the dirty buffers that were modified by this tranaction, and then writting a commit record to mark the process as complete. The commit policy is initiated, either when the commit interval expires, or when the write updates need to be synchronously written to disk (i.e., through fsync).

Each invocation of the *write* system call creates a new atomic operation handle that is added to the current active transaction. When the transaction moves to commit state, the kernel acquires a *journal descriptor block*. This block contains tags that map block buffers to their final location on disk of the file system (Figure 3.3). When a journal descriptor block fills up with tags, the kernel moves it to the journal together with the corresponding block buffers. The kernel allocates additional journal descriptor blocks as needed for each transaction.

For each block buffer that will be journaled, the kernel allocates a separate buffer head specifically for the I/O needs of journaling. Additionally, the kernel creates an auxiliary structure called *journal head* that associates the block buffer with the respective transaction. So, as depicted in Figure 3.6, for each journal block buffer there is (i) a buffer head that specifies the respective block number in the journal and, (ii) a journal head that points to the corresponding transaction.

In general, the buffer head of a journaled block buffer points to the original copy of the block buffer. However, if this block buffer is going to be used concurrently by another transaction, then the kernel creates in memory a new copy of the block buffer for the journal I/O transfer needs. When all the log records of a transaction have been safely written to the journal, the system allocates and synchronously writes to the journal a final commit block that states the transaction has committed successfully.

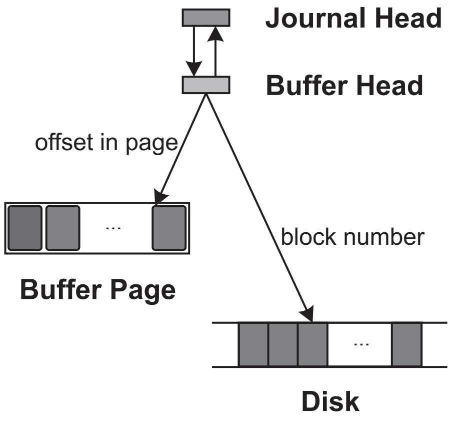


Figure 3.6: Two special structures, a buffer head and a journal head, need to be allocated for each block buffer that is going to be journaled. The buffer head specifies the respective block number in the journal, while the journal head points to the corresponding transaction.

3.3 Checkpoint Policy

Obviously, there is a limited amount of space in the journal, and this space needs to be reused. Besides, committed transactions that have all their blocks written to the final on-disk location, no longer need to be kept in the journal. The process of ensuring that a section of the log is committed fully to disk, so that this area can be reclaimed, is known as *checkpointing*.

The checkpointing process flushes the metadata and data buffers of a transaction not yet written to their actual location on the disk, allowing the transaction to be safely removed from the journal. The journal can have multiple checkpointing transactions, and each checkpointing transaction can have multiple buffers. The process considers each committing transaction, and for each transaction, it finds the metadata buffers that need to be written to the final location on disk. Subsequently, all these buffers are flushed in one batch. Once all the transactions are checkpointed, their log is removed from the journal.

In particular, checkpointing is initiated when the journal is being flushed to the disk (e.g., unmount) or when a new handle is started. A new handle can fall short of guaranteed number of buffers, so it may be necessary to carry out a checkpointing process in order to release some space in the journal. 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 general, the size of the journal is a configurable parameter in Ext3.

3.4 Recovery Policy

The *transaction committing* completes when a transaction has flushed all its records to the journal and has been marked as finished. This is done for each running transaction within a specified time period by the *kjournald* kernel thread. Subsequently, the *transaction checkpointing* completes when all the blocks of a committed transaction have been moved to their final location on disk and the corresponding transaction records are removed from the journal.

During *recovery*, the file system scans the log for committed complete transactions; incomplete transactions are discarded. Thus, if the system finds log records in the journal after a crash, it assumes that the unmount was unsuccessful and initiates a recovery procedure in three phases.

PASS_SCAN In the first phase, it finds the last record of the journal. From here, the recovery process knows which transactions need to be replayed. The exact state of the journal is unknown since the system does not know the point at which the failure occurred. The last transaction in the journal can be either in the checkpointing or in the committing state. A running transaction cannot be found, as it was only in memory during the crash. For committing transactions, the updates made need to be discarded. Thus, the system only considers committed transactions for replaying.

PASS_REVOKE During the second phase, the kernel builds a hash table from the *revoked* blocks. These are blocks of committed transactions that should not be written to their final disk location, because they are obsoleted by later operations. This is important to know in order to prevent older journal records from being replayed on top of newer data using the same block. This table is used every time that the system needs to find out whether a particular block should be replayed on disk.

PASS_REPLAY In the third phase, the recovery process writes to their final disk location the newest version of all the blocks that occur in committed transactions, and are not present in the hash table of revoked blocks.

If the system crashes again before the recovery finishes, the same journal can be reused in order to complete the recovery.

3.5 Summary

The Ext3 file system is a journaling extension to the standard Ext2 file system on Linux. Summarizing, the write updates are initially recorded sequentially in a separate area of the disk reserved for use as a journal. File system transactions which complete have a commit record added to the journal, and only after the commit is safely on disk may the file system write the updates back to their original location. During the recovery phase, the included blocks of a transaction can either be replayed or discarded. A checkpointing process is needed to flush the buffers of an already committed transaction, that have not yet been written to their final location through the normal dirty page flushing policy. Then, the transaction can be safely removed from the journal.

Journaling results in massively reduced time spent recovering a file system after a crash, and is therefore in high demand in environments where high availability is important. In addition, synchronous writes complete faster since they return as soon as the sequential log update completes. Data journaling can improve even more the response time of synchronous writes, but significant extra disk throughput on the journaling device is incurred due to the large volume of data written to the log.

Chapter 4

Architectural Definitions

In this chapter, we define the design goals of our study and explain the general architectural decisions taken before our prototype implementation. Initially, we detect the design inefficiencies of existing journaling techniques that lead to unnecessary disk overhead on the journal device. Then we propose a more efficient scheme for the fast and reliable storage of multiple concurrent updates.

4.1 Design Goals

Contemporary journaling file systems mostly care to maintain their metadata consistency. In order to provide high consistency guarantees, they only log metadata modifications in the journal. Nevertheless, two commonly used file systems, Ext3 and Reiser FS, additionally support data journaling as a mount option.

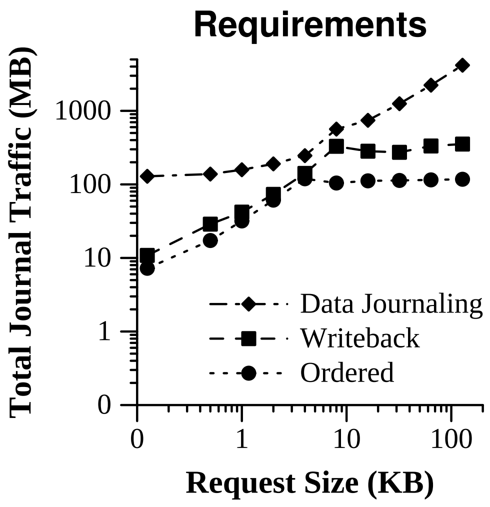


Figure 4.1: We measure the amount of traffic sent to the journal device according to the three journaling modes. The total journal traffic of data journaling is substantially higher in comparison to the other two modes. Additionally, at request sizes lower than 4KB, data journaling incurs traffic that changes sublinearly as a function of the write rate. This is reasonable since data journaling sends to the journal entire blocks rather than only the part that is modified by each write operation.

Comparisons across different journaling methods with general-purpose file server traffic, have shown that either ordered data writing or data journaling may lead to better performance depending on whether the aggregate workload is sequential or random-access [25]. Particularly, it was reported that data journaling improves the throughput of random I/O operations, but incurs much higher disk throughput than metadata journaling. This high cost of data journaling originates from the significant volume of data that is sent to the log. When the journal fills up with log records, a checkpoint process is triggered to synchronously write them to their final location, thus leading to further delay.

Furthermore, file system journaling allows synchronous writes to complete faster since they return as soon as the sequential log update completes. In the particular cases that both data and metadata blocks are logged, the benefit is higher, but this costs significant disk overhead on the journaling device. Unfortunately, the cost of data journaling can be high even with small writes, since for simplicity reasons, journaling techniques that support data journaling, log the entire blocks being modified rather than just their modified part.

In order to verify the significant overhead of data journaling, we examine the three mount options of Ext3 using periodic synchronous writes of varying request sizes. The difference in the amount of traffic sent to the journal device across the three mount options of Ext3 is depicted in Figure 4.1, where the total disk traffic is measured during a time period of 5 minutes. We observe that the total journal traffic of data journaling is substantially higher in comparison to the other two modes. Furthermore, we notice that at request sizes lower than 4KB, which is the default file system block size, data journaling incurs traffic that changes sublinearly as a function of the write rate. In particular, data journaling sends a large amount of traffic to the journal for small writes regardless of the actual size of the write requests. This is reasonable since data journaling sends to the journal entire blocks instead of the actual newly written bytes.

In the present study, we investigate the performance characteristics of data journaling in the context of synchronous writes that would be required among several situations including the reliable storage of incoming streaming data. In order to lower the cost of data journaling we introduce *differential data journaling*; a new journaling mode where a series of write modifications can be accumulated in a single journal block. Therefore, when the workload consists of many small writes we manage to reduce substantially the required journal throughput by avoiding to log a whole block for each data modification.

4.2 Partial Writes

The idea behind journaling is that an entire batch of updates can be written to the file system, but those updates do not take effect until a final commit update is made on the disk. In order to achieve this, the file system must keep both the old and the new contents of the updated data somewhere on disk until the final commit. The updated contents are stored in the journal on disk, where for each modified final block exists a corresponding journal block.

Therefore, in order to manage the partial data block modifications we need to introduce a new type of journal block. This new type is responsible for fitting as many partial modifications as possible. In case that it runs out of space, a new one can be allocated in its place.

4.3 Commit Policy

During the commit policy, dirty buffers are written to the journal followed by a commit record, that states that the process has completed successfully. As we have already explained, data journaling logs full blocks instead of the new bytes written by each update, and thus invokes unnecessary disk traffic, even in cases of small writes. Ideally, we should only journal the modified part of individual blocks, and this can be achieved through the proposed new journal block type. Through the use of this block we can substantially reduce the total number of blocks that need to be logged and, consequently we can improve considerably the journal device throughput.

4.4 Recovery Policy

During the recovery phase, the journal is initially scanned for incomplete committed transactions. If such transactions exist, they are replayed in the file system. Through this process whole blocks are read from the journal and, hence they can easily be written back to their final on-disk location.

However, our approach is more complicated than the default policy. In particular, some journal blocks include updates from more than one block modifications, and in order to be applied, the corresponding unmodified blocks need to be read from the disk. Thus, in case of partial modifications, every original block should be first read from the final on-disk location, and then written back, updated with the difference retrieved from the corresponding journal record. Nevertheless, when a block is retrieved from the journal and it is either a metadata or a fully modified block, then the default recovery process can be applied.

Furthermore, the successful completion of the recovery phase imposes the need for auxiliary information. The required information, that is known and stored for each journal block at the commit time, should include:

- the number of the corresponding block in the file system,

- the size and the starting offset of the modification inside the original disk block,

- anything else that could be useful during the replay of the partial updates from the journal blocks to their final location.

Subsequently, this information can be retrieved during the recovery process and, thus help the replay of the partial modifications.

4.5 Summary

As it is clear from the above analysis, traditional data journaling schemes can exhibit high and unnecessary disk traffic, as whole blocks are written to the journal, regardless of the modification size. In this thesis, we propose an advancement of the traditional data journaling approach, where the deltas (changes) to data blocks are journaled rather than the entire data blocks themselves. Our main idea is to accumulate a number of write modifications in a few single journal blocks, named partial journal blocks. Subsequently, during the uncommon case of recovering after a crash, we can easily recover the original blocks after applying to them the corresponding modifications from the partial blocks.

Chapter 5

Prototype Implementation

According to previous research, the journaling of both data and metadata improves the throughput of random I/O operations, while at the same time incurs much higher disk overhead than the metadata-only journaling modes. In the rest of this chapter, we outline the approach that we follow in order to keep low the overhead of data journaling and at the same time retain its significant performance gains. In particular, we describe the implementation of *differential data journaling*; a variation of the full data journaling mode of Ext3. Even though we consider our approach quite general, in our description we use the previously introduced terminology of Ext3, over which we have implemented our prototype.

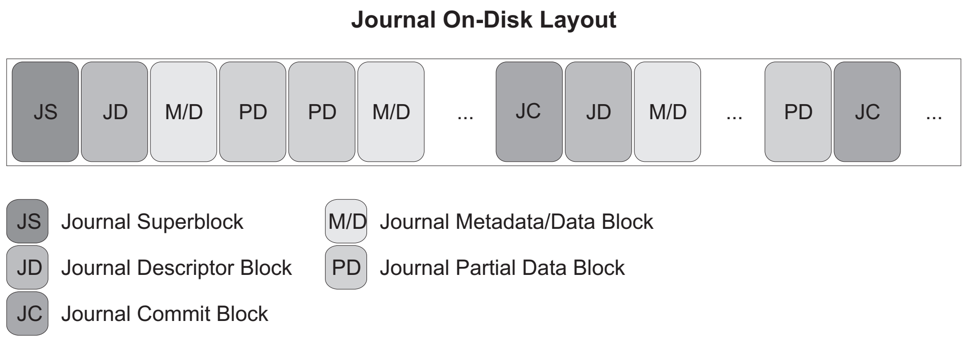


Figure 5.1: In differential data journaling, the on-disk layout of the journal has one new feature; the partial data blocks. These blocks are used to accumulate the modifications of multiple write operations in a reduced number of journal blocks.

5.1 Partial Blocks

The original journaling process of Ext3 transfers a full copy of each modified block buffer from memory to journal. This is true for both data and metadata blocks when they are journaled according to the mount options of the file system. Thus, even a single bit change in a bitmap results in the entire bitmap block being logged. In case of small writes that modify only a part of a block buffer, the logging of full blocks can have a multiplier effect at the throughput required by the journal device, as we have already observed in Figure 4.1. The actual waste in journal device throughput depends on the fraction of the block buffer that is left unmodified by each write operation. Ideally, only the modified part of the block should be written to the journal. Subsequently, at the uncommon case that the recovery process is initiated, the original block should be read from the final on-disk location and then written back, updated with the difference retrieved from the corresponding journal record.

In order to implement differential data journaling, we introduce a new type of journal block that we use to accumulate the modifications of data blocks from multiple write operations (Figure 5.1). We call this type of journal block *partial*, to differentiate it from *full* blocks, which are blocks fully modified by a single write operation. Partial blocks are only used to gather the partial updates of data blocks, rather than metadata modifications. In summary, the commit process treats data blocks differently than the metadata ones, while two different types of data blocks are distinguished; partial that store writes smaller than the default block size, and non-partial that correspond to fully written buffers.

5.2 Journal Heads

As we have already explained in paragraph 3.2, for each journal block buffer there is a corresponding journal head that associates the block with a transaction. Additionally, the journal head points to a buffer head that links the buffer to a buffer page and other information required for the transfer to the journal device.

For writes that only modify part of a block, we expanded the journal head with two extra fields, the offset and the length, respectively, of the partially modified block pointed to by the buffer head. As we see below, we make use of the journal head in order to prepare the blocks that we actually send to the journal.

5.3 Tags

As the commit process is started, a buffer for the journal descriptor block is allocated. In data journaling, the transaction logs both data and metadata modifications. The journal descriptor block contains a list of fixed-length tags, where each tag corresponds to one write. Originally, each tag contains two fields:

- The final disk location of the modified block.
- Four flags for journal-specific properties of the block.

In our design, we introduce three new fields in each tag:

- A flag to indicate whether the corresponding block is partially modified or not.
- The length of the new bytes written in the partial block.
- The starting offset in the data block of the final disk location.

This data is persistent and can be used for recovery if a failure occurs.

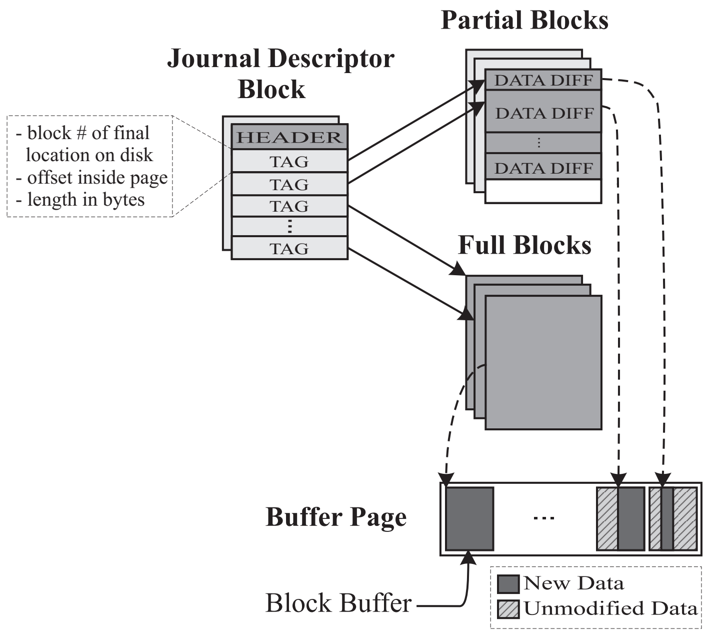


Figure 5.2: In the differential data journaling we use a new type of journal blocks, the partial journal blocks, to accumulate the data modifications from multiple writes. Full journal blocks are still used for metadata or blocks that are completely modified by write operations. The descriptor's tags are used to keep the correspondence between final location and journal blocks, and also to describe the partial modifications inside the partial journal blocks.

Once the tags fill up a journal descriptor block, the descriptor block and all the corresponding data and metadata blocks are written consecutively to the journal. Furthermore, additional journal descriptor blocks are allocated as required by the transaction.

5.4 Commit Policy

The commit process of differential data journaling differs from the original approach in that it makes further use of partial blocks. In particular, a new partial data block is allocated when a new transaction is started and it is used to accumulate all the modifications with size smaller than the default file system block size. The journal descriptor block stores the mapping of each journal block to its actual on-disk location in the form of tags. In our prototype, it additionally includes tags that describe the partial writes (Figure 5.2). If a write updates part of a data block, the modified bytes are copied to the current partial block buffer of the transaction. When the available space of a partial data block is not sufficient to store a new incoming update, then a new partial block is allocated to serve the next partial modifications. In case that a write system call modifies a metadata block or fully writes a data block, we log the corresponding full block instead.

We might still need to create a copy of the full block in order to freeze the version that we send to the journal, if the block is going to be modified shortly by another transaction. Once all data and metadata is on safe storage, the transaction needs to be marked as committed so that it can be guaranteed that all its updates are safe in the journal. Eventually, the commit process completes right after the journal commit block is synchronously written to the log.

5.5 Recovery Policy

During the recovery process, the data modifications are retrieved from the journal, and are subsequently applied to the blocks corresponding to the final on-disk location.

Initially, when a descriptor block is read from the log, we extract its included tags. Each tag can describe either a partial or a full log block. When we meet the first tag that describes a partial write modification, the next log block is retrieved from the journal, and from that point on it is used as the partial block of the current transaction. Since the data of consecutive writes are placed next to each other in the partial block, their corresponding starting offsets can be deduced from the length field in the tags. In case that the length field of a tag exceeds the end of the current partial block, the next block is read from the journal and becomes the new partial block of the transaction. We use the starting offset tag field to read into a kernel buffer the disk block that we will modify in order to apply the data modifications.

However, if the partial block flag is not set, then the next block is retrieved from the journal, which is eventually treated as a metadata or a full data block. Obviously, the full block is directly written to the final disk location without reading first the previous version from the disk.

Chapter 6

Experimental Results

In the present chapter, initially, we introduce the hardware configuration that we used in our performance measurements. Afterwards, we study the requirements and performance of our differential data journaling implementation with respect to the ordered, the writeback and the default data journaling modes of Ext3, and we graphically present our experimental results.

6.1 Experimentation Environment

We implemented the differential data journaling in the Linux kernel version 2.6.18. We evaluated our prototype implementation using x86-based server nodes running the Debian Linux distribution. For the majority of the experiments we used nodes with a quad-core 2.66GHz processor, 2GB RAM, and two SAS 15KRPM disks, each of 300GB storage capacity and 16MB internal buffer. Additionally, for one set of the experiments, a 2.33GHz quad-core processor and two SATA 7.5KRPM disks, each of 250GB and 16MB on-disk cache, were used.

In the general case, two separate disks are used; one for the journal and another one for the actual file system structures, except for one case that is explained later in this chapter. Furthermore, we use the default file system parameters of Linux that set the page and the block size to 4KB. We also keep the default journal size of 128MB, but manually tune for best performance the writeback period and expiration period of the dirty page flush process. In our measurements, we assume that write operations are followed by the fsync system call for synchronous completion.

Previous research reports that, by default, a synchronous write operation returns as soon as the data reaches the on-disk write cache, rather than the storage media. This behavior renders the system unreliable unless we disable the on-disk buffer cache or use controllers with battery-backed cache [23]. In most of our experiments, we kept enabled the disk write cache, which essentially emulates devices with battery-backed memory. However, we also evaluated our system with the write caches disabled. As we explain, the disk write cache adds no benefit to streaming workloads but leads to significant performance advantages in traditional applications.

In order to study the characteristics of our system and evaluate our implementation, we did extensive performance measurements. In particular, the first set of experiments is based on a microbenchmark that we have built for the needs of a streaming workload evaluation. This benchmark consists of multiple threads that periodically apply synchronous writes at a specific rate. In our evaluation, we examine the disk throughput requirements and the average latency of each write. During the next set of experiments, we used the Postmark benchmark to measure performance in an environment of temporary small files that is typical for electronic mail, newsgroups and web-based commerce [20]. Thus, we investigate the benefit of data journaling in applications other than streaming. Finally, we performed a series of experiments in order to examine the possible overhead of our prototype implementation. Therefore, we measure the time needed to recover the system to a consistent state after a crash, the CPU overhead that our approach incurs and perform some other experiments that are presented in the rest of this chapter.

At last but not least, our prototype implementation of differential data journaling is

Table 6.1: Various rates used from different types of streams.

Stream Type	**Estimated Average Rate**
Environmental Measurements (humidity, temperature etc.)	(tens of bits - hundreds of Kbits)/sec
Audio Streams (telephone quality, mp3 etc.)	(hundreds of bits - hundreds of Kbits)/sec
Video Streams (videophone quality, mpeg etc.)	(tens of Kbits - tens of Mbits)/sec

being used as a working environment over a period of three and a half months. The system has demonstrated a stable behavior during this entire period.

6.2 Streaming Workloads

In our first set of experiments, we evaluate the benefits and requirements of differential data journaling in a file system. We consider the case where the incoming data from a large number of concurrent streams is stored synchronously on the same disk. Actually, through the use of microbenchmark that we developed, we emulate the behavior of streaming workloads, where massive numbers of streams need to be stored synchronously at the same disk facility.

In digital multimedia, the data *rate*, or else *bitrate*, represents the amount of information of a recording that is stored per unit of time. Various factors can influence a stream's rate, such as the compression scheme that is used or the nature of the particular steaming application. For instance, some sensors may send video and audio streams of high quality at high rates, while others may generate environmental measurements at much lower rates. In Table 6.2, we present the range of different rates that are used according to the type of each stream.

Our microbenchmark tool allows us to examine the performance characteristics of streams with different rates, while varying the degree of concurrency. So, in order to press the system, we increase the total number of streams between the different runs. At each execution, a sequence of write updates is synchronously applied to the system for a specified amount of time, while according to the stream rate different record sizes are used. Typically, a low-rate streaming workload implies many small synchronous writes applied to the same storage media, while higher-rate streams typically correspond to larger ones. In particular, the rate of a low-rate streaming workload varies from tens of bits up to few tens of kilobits per second. Therefore, the corresponding write request size is much smaller than the default Linux kernel block size. On the other hand, high-rate streams send data over megabits per second, thus leading to request sizes that range from hundreds of kilobytes and on.

6.2.1 Flushing Policy

In streaming workloads, even though each stream simply appends data sequentially to the end of a separate file, the aggregate traffic is random. However, data journaling safely stores data on the journal at sequential throughput and lazily transfers it to the final location at a rate that we can control. Particularly, we manually tune for best performance the writeback period and the expiration period of the dirty page flush process, according to the rate and the number of the streams that are involved in each experiment's execution. The writeback period is used to define when the pdflush daemons wake up and write old data out to disk, while the expiration period defines when dirty data is old enough to be eligible for writeout by the pdflush daemons. Data which has been dirty in memory for longer than this interval will be written out next time a pdflush daemon wakes up. In Linux kernel, the writeback period is by default set to 5 seconds and the expiration period to 30 seconds.

Ideally, in case of low-rate streams we would like to accumulate multiple write updates in memory for a long period of time, in order to benefit as much as possible from the batching of related writes. We achieve this by delaying the awakening of pdflush daemons and increasing both the default expiration and writeback intervals. Nevertheless, the new time intervals should be carefully selected, to avoid overfitting either the journal device, or the memory. In general, when there is no available space left in the journal or the memory, the subsequent writes should block, waiting for the journaled updates to move from memory to their final on-disk location, through either the checkpointing or the

Table 6.2: Flushing Policy - Stream Rate of 1Kbps

Number of Streams	**Writeback Period (in seconds)**	**Expiration Period (in seconds)**
100	10	300
500	10	300
1000	10	150
2000	10	60
3000	1	30
4000	1	30
5000	1	5
6000	1	5
7000	1	5
8000	1	5

kernel's dirty page flush process. For this reason, we choose the expiration interval to be long enough for low-rate streams, but we wake up the pdflush daemons rather frequently to clean the memory from old updates. Additionally, when the number of low-rate streams increases, so does the total amount of data written and hence, we lessen the expiration interval to avoid the checkpointing and the dirty page flush process. Tables 6.2 and 6.3 present the particular tuning of the dirty page flushing parameters that we use in our measurements, for low-rate streams of 1Kbps and 10Kbps respectively.

Multiple high-rate streams generate large volumes of data that need to be stored on

Table 6.3: Flushing Policy - Stream Rate of 10Kbps

Number of Streams	**Writeback Period (in seconds)**	**Expiration Period (in seconds)**
50	10	300
100	5	100
500	5	60
1000	1	30
1500	1	10

Table 6.4: Flushing Policy - Stream Rate of 1Mbps

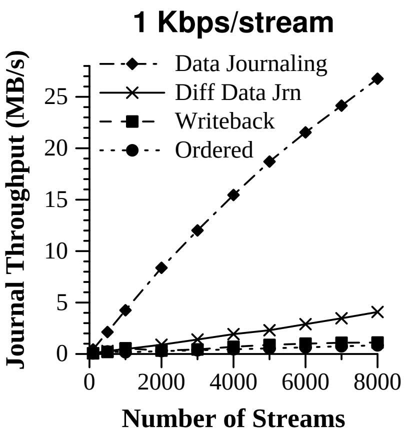
Number of Streams	**Writeback Period (in seconds)**	**Expiration Period (in seconds)**
10	5	20
25	1	5
50	1	3
75	1	1
100	1	1

the same disk facility. The benefit of batching together such updates is insignificant due to their size. Therefore, we don't need to keep them in memory for long time. In these cases, we can either use the default expiration and writeback periods, or slightly reduce them according to the generated amount of data. Once again, when the number of streams increases we can reduce the intervals even more, in order to prevent the memory structures from getting full. Table 6.4 presents the configuration of the writeback and expiration periods in case of high-rate streams of 1Mbps.

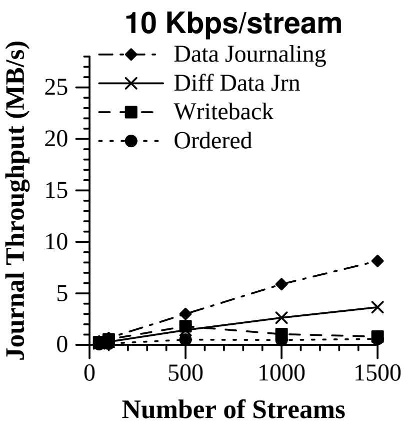
Finally, since we fsync every individual write, we use the default journal commit interval of 5 seconds to wake up the kjournald daemon, as it eventually does not influence our measurements.

6.2.2 Journal Traffic

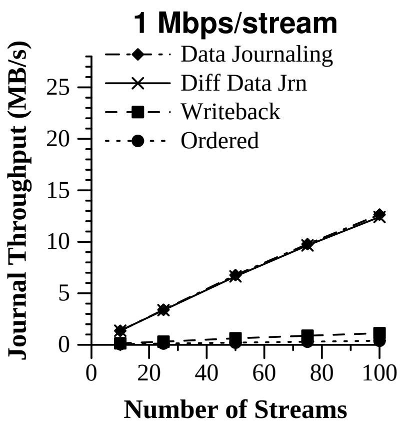
In Figure 6.1 we measure the journal device throughput across different numbers of streams and rates of 1Kbps, 10Kbps and 1Mbps. In Figure 6.1(a), we observe that when the number of streams reaches several thousands, data journaling sends around 30MB/s of log records to the journal. Instead, differential data journaling keeps the traffic lower than 5MB/s. This behavior is less intense as the stream rate increases from 1Kbps to 10Kbps (Figure 6.1(b)), and in fact the two data journaling modes overlap for streams of 1Mbps (Figure 6.1(c)). As expected, in all three cases the two metadata-only journaling modes keep the overhead of the journal device at the low levels, since only a small amount of information is finally logged.



(a)



(b)



(c)

Figure 6.1: We examine the journal device throughput across different numbers of streams and rates of 1Kbps, 10Kbps and 1Mbps. For low-rate streams, the disk overhead of differential data journaling is comparable to that of ordered and writeback modes, unlike the default data journaling mode which leads to journal device throughput by several factors higher. Nevertheless, at high rates, differential data journaling overlaps with the default data journaling mode in terms of journaling throughput.

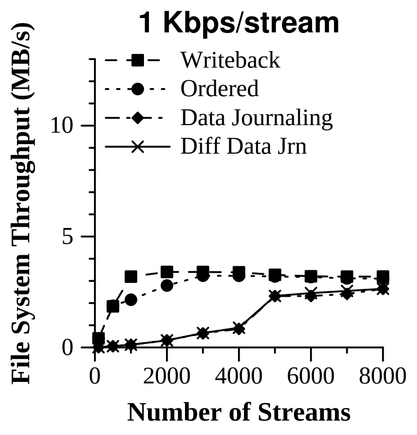
In general, we observe that at low rates, the journal throughput of differential data journaling is close to that of ordered and writeback modes. The corresponding throughput in the case of the default data journaling mode is several factors higher. Particularly, a low-rate streaming workload implies many small synchronous writes applied to the same storage media, while higher-rate streams typically correspond to larger ones. In the case of low-rate streams, differential data journaling manages to reduce substantially the journal throughput. This is achieved through the accumulation of multiple write updates into a single journal block. On the other hand, default data journaling incurs significant journal overhead because of the full-block logging scheme. Even though a corresponding increase in memory copy activity is likely, this is hardly a problem as we see later. Therefore, we can reliably store the data of low-rate streams without excessive journaling cost.

Nonetheless, at high rates, differential data journaling overlaps with the default data journaling mode in terms of journaling throughput, while the required journal disk overhead of metadata-only modes remains significantly low. As the total amount of data written increases, the benefit of partial writes becomes nominal and large volumes of data are finally sent to the journal.

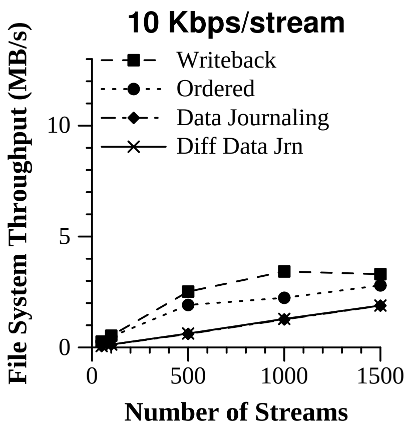
6.2.3 Final Location Traffic

In Figure 6.2 we measure the disk throughput for the update of the final location on the file system. We notice that the ordered and writeback methods, that only journal metadata, incur consistently higher throughput to the final disk location, especially at low-rate streams. Besides, metadata-only journaling allows synchronous updates to complete by first forcing data blocks to their final on-disk location, before the corresponding metadata blocks are synchronously written to the journal. Instead, the two data journaling modes append both the metadata and data updates synchronously, but efficiently to the journal, and keep the corresponding data blocks in memory for some time. There, each block has the chance to receive the updates from multiple writes, before it is transferred to its final location on disk. Furthermore, we tune the parameters of the dirty page flush process in order to gain as much as possible from the opportunity of batching. Hence, for low-rate streams we open enough the expiration interval and allow many small modifications of single blocks to be accumulated.

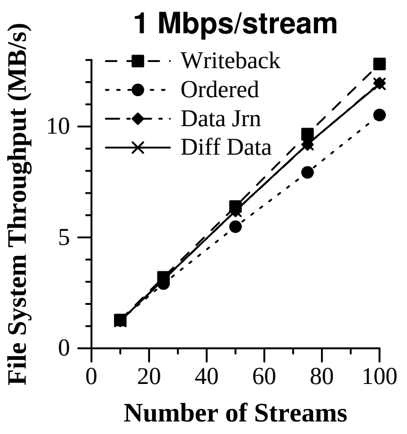
On the other hand, for high rate streams, we have reduced considerably the expiration and the writeback periods, in order to prevent the journal device from becoming full. Generally, when the journal fills up, a checkpointing process is initiated and all the subsequent writes are blocked. However, this tuning, in the long run, prevents us to benefit from the batching opportunities offered during small writes. Thus, the same number of write updates are applied to the final on-disk location, regardless of the journaling mode.



(a)



(b)



(c)

Figure 6.2: We examine the throughput of the file system device across different numbers of streams and rates. For low-rate streams, the two metadata-only journaling modes require up to several factors higher throughput than the two data journaling modes. Nevertheless, in case of high-rate streams, the final location disk overhead is comparable across all the four modes.

Summarizing, at low rates, the writeback and ordered modes tend to require up to several factors higher throughput than the two data journaling modes. We attribute this benefit of the two data journaling modes to the aggregation of multiple writes that update the same block. Since journaling keeps each update safe on disk, dirty pages can remain for a configurable time period in memory before they are flushed to the file system disk. Nevertheless, in case of high-rate streams, the final location disk overhead is comparable across all the four modes since, due to the large amount of data written, there is no benefit from batching together related writes.

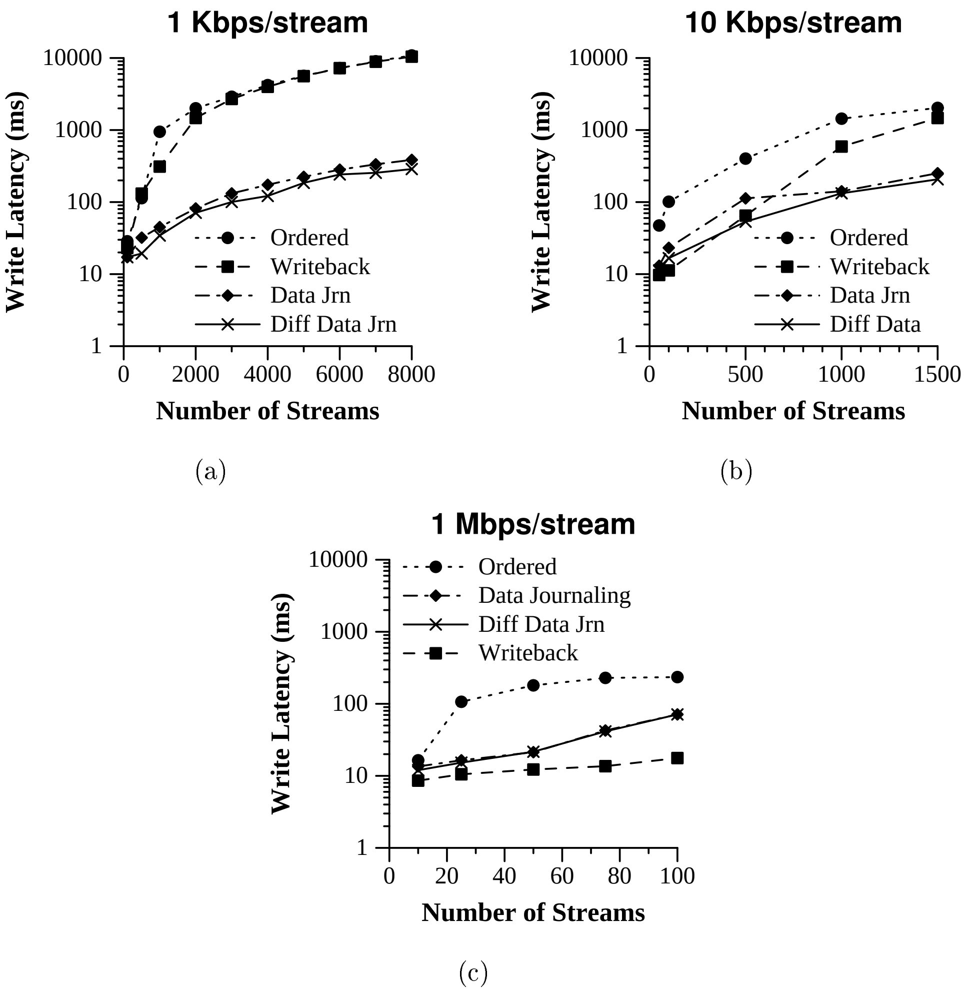


Figure 6.3: We measure the average write latency of synchronous updates at different rates and streams. Synchronous writes are usually avoided because they are known to incur high latency in typical file systems. However, data journaling modes can benefit from the sequential journal's throughput that eventually allows the system to safely and quickly store the incoming data.

6.2.4 Write Response Time

The benefits of the two data journaling modes are even more impressive, when we consider the average latency of the synchronous writes, as depicted in Figure 6.3. In order to demonstrate the differences across the different modes, we use logarithmic scale at the y axis. As we move from higher to lower rates, the write latency of the ordered and writeback modes appears from several factors up to orders of magnitude higher than those of the two data journaling modes. In particular, in Figure 6.3(a), we see that the ordered and writeback modes incur almost two orders of magnitude higher latency with respect to the other two modes, when serving large numbers of low-rate streams. Thus, a write operation that completes in tens of milliseconds with data journaling, takes as high as 10 seconds with ordered mode.

Data journaling modes force write updates synchronously to the journal. There the written transactions are appended sequentially and efficiently. However, in case of metadata-only journaling modes, data is flushed synchronously to the fixed location before the corresponding metadata blocks are synchronously written to the journal. Especially, when we have large numbers of streams, data blocks are distributed across random locations on disk, and hence incur seeking overhead and rotational latency when data writes are forced to the final location.

Such a high write latency in the default Ext3 journaling mode, the ordered mode, raises issues about the ability of the system to quickly and safely store incoming measurements. This is crucial, especially at critical time periods before physical catastrophes, when the arriving data matter the most. Synchronous writes are usually avoided because they are known to incur high latency in typical file systems. This is true even when the write cache of the disk is enabled. Nevertheless, the sequential throughput of the journal has a considerable impact to the ability of the system to store safely the incoming data in a short period of time.

6.2.5 CPU Utilization

A possible overhead of our prototype implementation is the CPU cost that is needed, so that multiple data modifications can be accumulated in single journal blocks. This is achieved through the memory copy of the modified block parts to the appropriate journal partial block.

In Figure 6.4 we evaluate the impact of the four journaling modes to the total CPU utilization of the system. We observe that the system utilization always remains less than 10%. At both low and high rates, the CPU remains mostly idle, whether doing nothing or waiting for the I/O operations to finish. Therefore, the processing cost of differential data journaling remains comparable to that of the other three mount modes.

Consequently, the accumulation of multiple write updates in one block in differential

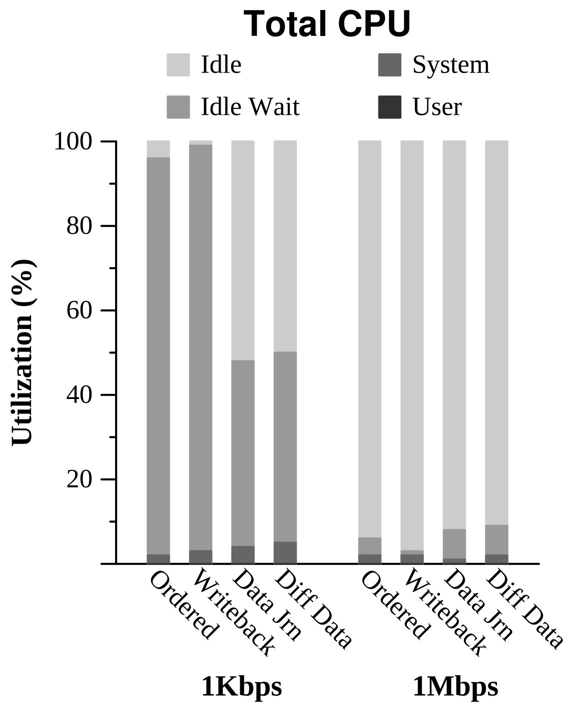


Figure 6.4: We investigate the total CPU utilization of the system across the different journaling modes. In all the four cases, at both low and high rates, the CPU remains mostly idle, whether doing nothing or waiting for the I/O operations to finish. Thus, the extra CPU cost of differential data journaling due to memory copy operations is nominal, in comparison to the other three modes.

data journaling does not create an overhead, for the memory copy, much higher than the other modes.

6.2.6 Mixed Workload

Finally, a number of experiments with workloads that consist of mixed set of streams with different rates were performed and lead to measurements similar to the above. The results of the mixed workload tend to approach respectively the behavior of streams with low or high rate, depending on the prevalence of the corresponding type of stream in the workload.

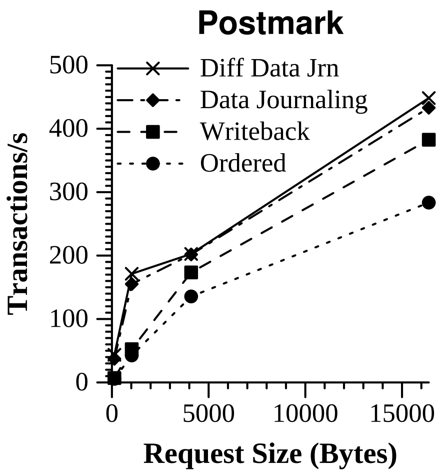


Figure 6.5: We evaluate the Postmark benchmark results. Both data and differential data journaling modes perform several factors better from the metadata-only journaling modes. In particular, due to low write latency, data journaling modes manage to serve a larger number of transactions per second.

6.3 The Postmark Benchmark

In Figure 6.5, given the very encouraging results that we obtained for workloads with low-rate streams, we evaluate data journaling with Postmark. This benchmark is typically used to study the performance of small writes [17]. It is designed by Jeffrey Katcher in order to replicate the small file workloads seen in electronic mail, netnews, and web-based commerce under heavy load.

We measure the achieved transaction rate with a workload of 10000 transactions over 500 files, and a mix of read, append, create and delete file operations. We run Postmark with 100 threads and file ranges from half kilobyte to a hundred kilobyte.The actual duration of the experiment varies depending on the efficiency of the requested operations. We run the benchmark in a range of block sizes from 128 bytes to 16KB. During our experimental measurements, we use the kernel's default dirty page flushing parameters that are presented in Table 6.5. In Figure 6.5 the x axis refers to the request size of the read and write operations, while the y axis is the number of transactions that can be served per second.

Our main observation is that the two data journaling modes perform several factors better than the metadata-only journaling modes. The performance improvement is higher for small block sizes. However, even with the block size equal to 16KB, the data journaling

Table 6.5: Flushing Policy - Postmark

Writeback Period	**Expiration Period**	**Commit Interval**
5 seconds	30 seconds	5 seconds

modes double the measured transaction rate. This behavior comes from the low write latency that the two data journaling modes incur, in contrast to the metadata-only modes. Thus, within the same time period, data and differential data modes manage to serve much more transactions than the other modes.

Consequently, if somebody uses differential data journaling to keep low the extra journaling throughput, one can improve substantially the performance of applications that need synchronous small writes.

6.4 Recovery Time

In a different experiment, we evaluate the ability of the system to recover quickly after a system crash that leads to log records appearing in the journal during the reboot. In this setting, we have 100 threads that apply 100 write updates with request size 125 bytes. Furthermore, we disable the writeback and expiration time periods of the pdflush kernel thread, in order to ensure that the transactions commit to the journal, but don't checkpoint the updates to the final location on disk. Then we cut the power to the system. During the reboot, we measure, within the kernel, the time period of the file system recovery.

In Figure 6.6, we breakdown the total recovery across the three passes that scan the transactions, revoke blocks, and replay the committed transactions. We notice that the scanning period for differential data journaling is much lower than that of default data journaling and actually similar to those of ordered and writeback. This is reasonable, due to the new type of journal blocks that we introduced, the partial data blocks. Thus, gathering small updates into a small number of journal blocks, differential data journaling logs much fewer blocks than default data journaling, which for each update sends a full block to the journal. Instead, in the metadata-only journaling modes, the amount of journaled blocks is even smaller since data blocks are not logged at all.

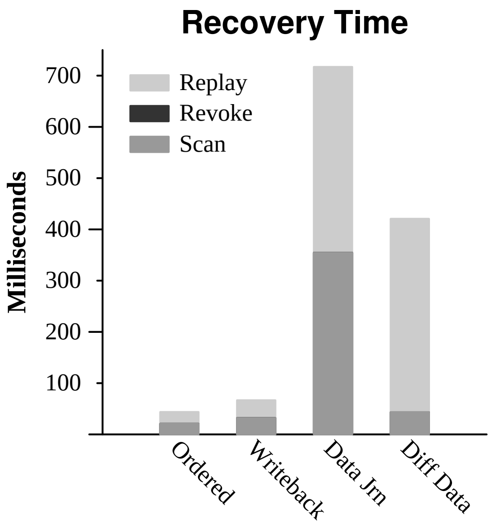


Figure 6.6: We measure the recovery time across the four journaling mount modes. We observe that differential data journaling requires much lower time for the scan pass than the default data journaling mode, while the replay pass takes comparable time across the two modes.

For the revoke phase, as expected, the time period needed is comparable to all the four modes. During the last phase, in differential data journaling extra block reads from the disk are required so that the modifications from the journal partial blocks can be applied to the corresponding final disk blocks during replay. On the other hand, in the default data journaling case, this is avoided since whole blocks are logged, and during replay these blocks can directly replace the existing final disk blocks without first reading them. Nonetheless, despite the extra block reads involved in the replay of differential data journaling, the time the replay phase takes ends up comparable to that of the default data journaling.

6.5 Other Issues

Since the ordered mode does not take full advantage of the separate journal device, we also investigate the case where we use the two SAS disks in RAID0 configuration with hardware controller support. For the configuration of this set of experiments, we use as journal a normal file within the same file system device rather than a separate partition. From our measurements (not shown) we observe that the write latency drops to half in

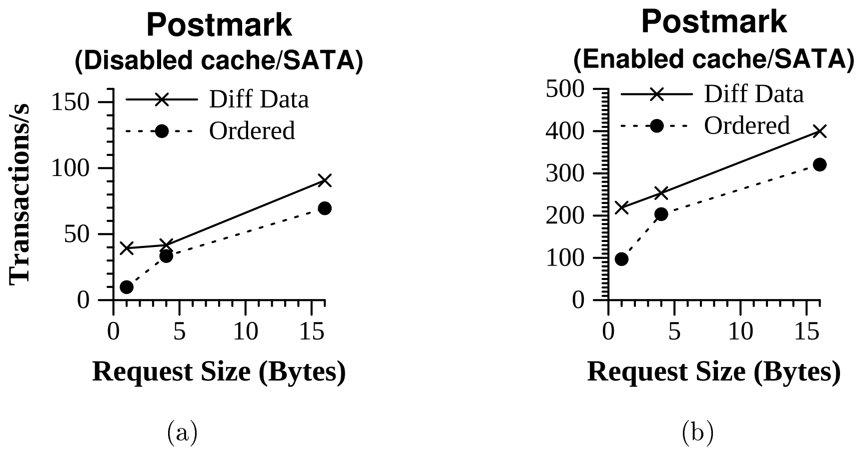


Figure 6.7: We figure the Postmark results while enabling and disabling the on-disk write cache. We notice that the two data journaling modes almost double the transaction rate with respect to the ordered mode that is commonly used by default.

the ordered mode, when compared to the case where we dedicate one disk to the journal. After the change, the write latency of differential data journaling remains about the same as before. The relative difference between the latencies of the two modes is still high across the different streams rates and in excess of a magnitude order for 1Kbps streams.

In a different experiment, we examine the effects from disabling the write cache of the disks. For these measurements, we use a server with two 250GB SATA disks. We find that the disabled write cache of the disks makes no difference to the streaming workload measurements in comparison to the case that the cache is enabled. However, in the case of the Postmark benchmark with 5000 transactions, disabling the write cache scales down the performance of the different mount modes, as shown in Figure 6.7.

Specifically, we disable the on-disk write cache to ensure that the writes only return after they reach the media. The advantage of differential data journaling is evident especially with small read and write requests. Furthermore, when we enable the on-disk write cache, performance scales similarly for the ordered mode and differential data journaling, while the relative difference remains. Overall, differential data journaling still maintains a significant advantage with respect to the ordered mode, especially at low stream rates.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

The unique demands placed by high-volume stream storage indicate that neither existing databases nor file systems are directly suited to handle their storage needs. In our vision, a general-purpose stream storage facility could serve as a building block for a variety of applications in the entire range from network packet monitoring to urban traffic control with the appropriate indexing functionality built separately at a higher level when needed. The operation reliability in such applications is a primary challenge, especially when public safety concerns are involved. In order to improve their operation reliability, general-purpose file systems apply journaling techniques to preserve metadata consistency across system crashes at minimal recovery time. Motivated from the emerging need to reliably store and handle large numbers of streams for real-time or retrospective processing, we have taken a fresh look at file systems that support data journaling.

We have used a widely known file system mounted with data journaling mode and, after applying synchronous writes, we demonstrated that the journal device throughput is high because the journal log records store entire blocks rather than their modified part.

Then, we introduced the differential data journaling mode, based on the idea of accumulating the updates from multiple writes into a single journal block. In order to implement differential data journaling, we designed a new type of journal block that we call partial data block. Additionally, we tune the timing of dirty page flushing to complete in the background rather than synchronously with the write operations. Using streaming workloads, we found that differential data journaling reduces the journal traffic substantially in comparison to the default data journaling mode, especially for streams with low rates. The sequential throughput of the journal reduces the write latency up to orders of magnitude for the data journaling modes with respect to metadata-only journaling. Finally, we have experimented with a typical small-write workload and measured substantial improvement in the supported transaction rate. Overall, differential data journaling offers fast storage across streaming and traditional workloads at relatively low disk throughput requirements.

7.2 Future Work

There are many directions for future work, mainly regarding the performance evaluation of our implementation. In the future, we primarily plan to extend the experimental measurements of our prototype implementation, to validate further the contributions of our study and emphasize the offered performance gains.

Only experimentation in a real streaming environment can reveal the potential of our approach. Therefore, initially, we aim to examine the behavior of differential data journaling in the context of a distributed file system that we are currently building for the needs of streaming data storage. In particular, a real workload with varying number of clients applying concurrent writes of stream data to the same storage server, will provide a more realistic environment in terms of the ability of differential data journaling to serve streaming workloads.

Regardless of the possible performance loss under certain circumstances, given the nature of the load for which our system is designed, a direct comparison with the log-structured file system or other journaling file systems would also be valuable in order to demonstrate the benefits of our architecture.

Furthermore, heterogeneity, a main feature of most streaming storage systems, is itself a challenging problem to be handled by the existing implementations. We have already performed a series of measurements across mixed workloads, where low and higher rate streams coexisted. Yet, we need to examine further how differential data journaling performs in such heterogeneous scenarios.

Moreover, we intend to examine the behavior of differential data journaling under some database workload. TPC-C simulates a complete computing environment where a population of users executes transactions against a database [9]. The benchmark that we are going to use constitutes a realistic implementation of order-entry built on top of Postgres.

Finally, a possible extension of our work would investigate the automatic tuning of system parameters related to the timing of dirty page flushes.

Bibliography

[1] Stergios V. Anastasiadis, Rajiv G. Wickremesinghe, and Jeffrey S. Chase. Circus: Opportunistic block reordering for scalable content servers. In *USENIX Conference on File and Storage Technologies*, pages 201–212, 2004.

[2] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In *ACM Symposium on Principles of Database Systems*, pages 1–16, New York, NY, USA, 2002. ACM Press.

[3] Hari Balakrishnan, Magdalena Balazinska, Don Carney, Ugur Cetintemel, Mitch Cherniack, Christian Convey, Eddie Galvez, Jon Salz, Michael Stonebraker, Nesime Tatbul, Richard Tibbetts, and Stan Zdonik. Retrospective on aurora. *The VLDB Journal*, 13(4):370–383, 2004.

[4] Pere Barlet-Ros, Gianluca Iannaccone, Josep Snjuas-Cuxart, Diego Amores-Lopez, and Josep Sole-Pareta. Load shedding in network monitoring applications. In *USENIX Annual Technical Conference*, pages 59–72, Santa Clara, CA, 2007.

[5] 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.

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

[7] Don Carney, Uğur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Greg Seidman, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Monitoring streams - a new class of data management applications. In *International Conference on Very Large Data Bases*, pages 215–226, Hong Kong, China, 2002.

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

[9] Transaction Processing Council. Tpc benchmark c standard specification, revision 5.9. Technical report, 2007.

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

[11] Manuel Esteve and Carlos E. Palau. A flexible video streaming system for urban traffic control. *IEEE Multimedia*, 13(1):78–83, January 2006.

[12] Ricardo Galli. Journal file systems in linux. *Upgrade*, 2(6):50–56, December 2001.

[13] Gregory R. Ganger, Marshall K. McKusick, Craig A. N. Soules, and Yale N. Patt. Soft updates: a solution to the metadata update problem in file systems. *ACM Transactions on Computer Systems*, 18(1):127–153, February 2000.

[14] 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.

[15] 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 System Design and Implementation*, San Diego, CA, USA, 2008.

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

[17] 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.

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

[19] Gianluca Iannaccone, Christophe Diot, Derek McAuley, Andrew Moore, Ian Pratt, and Luigi Rizzo. The como white paper. Technical Report Technical Report IRC-TR-04-17, Intel Research, 2004.

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

[21] Purushottam Kulkarni, Fred Douglis, Jason LaVoie, and John M. Tracey. Redundancy elimination within large collections of files. In *USENIX Annual Technical Conference*, pages 59–72, Boston, MA, 2004.

[22] Darrel D. E. Long, Patrick E. Mantey, Craig M. Wittenbrink, Theodore R. Haining, and Bruce R. Montague. Reinas: the real-time environmental information network and analysis system. In *IEEE COMPCON*, pages 482–487, March 1995.

[23] Edmund B. Nightingale, Kaushik Veeraraghavan, Peter M. Chen, and Jason Flinn. Rethink the sync. In *Usenix Symposium on Operating Systems Design and Implementation*, pages 1–14, Seattle, WA, 2006.

[24] Juan Piernas, Toni Cortes, and Jose M. Garcia. Dualfs: A new journaling file system without meta-data duplication. In *ACM International Conference on Supercomputing*, pages 137–146, New York, NY, 2002.

[25] 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.

[26] Sean Quinlan and Sean Dorward. Venti: a new approach to archival storage. In *USENIX Conference on File and Storage Technologies*, Monterey,CA, 2002.

[27] Mended 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.

[28] Margo Seltzer, Keith A. Smith, Hari Balakrishnan, Jacqueline Chang, Sara Mcmains, and Venkata Padmanabhan. File system logging versus clustering: A performance comparison. In *Usenix Annual Technical Conference*, pages 249–264, 1995.

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

[30] Carl A. Waldspurger. Memory resource management in vmware esx server. *SIGOPS Operating Systems Review*, 36(SI):181–194, 2002.

[31] 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.

[32] Wenguang Wang, Yanping Zhao, and Rick Bunt. Hylog: A high performance approach to managing disk layout. In *USENIX Conference on File and Storage Technologies*, pages 145–158, Berkeley, CA, USA, 2004. USENIX Association.

[33] Zhihui Zhang and Kanad Ghose. yfs: A journaling file system design for handling large data sets with reduced seeking. In *USENIX Conference on File and Storage Technologies*, pages 59–72, San Francisco, CA, 2003.

Author's Publications

Andromachi Hatzieleftherou, Stergios V. Anastasiadis, Okeanos: Fast and Reliable Stream Storage Through Differential Data Journaling, Technical Report DCS2008-8, Department of Computer Science, University of Ioannina, November 2008.

Andromachi Hatzieleftheriou, Stergios V. Anastasiadis, Okeanos - Reliable Archival Storage for Heterogeneous Stream Data, EuroSys, Glasgow, Scotland, UK, April 2008 (poster).

Short Vita

Andromachi Hatzieleftheriou was born in Serres, Greece in 1985. She was admitted at the Computer Science Department of the University of Ioannina in 2002. She received her BSc degree in Computer Science in 2006 and she is currently a postgraduate student at the same department. She is a member of the Systems Research Group of the University of Ioannina since 2007. Her main research interests lie in the field of file and storage systems.