Stony Brook University

From Tuples to Files: a Fast Transactional System Store and File System

A Thesis Presented

by

Pradeep J. Shetty

to

The Graduate School

in Partial Fulfillment of the

Requirements

for the Degree of

Master of Science

in

Computer Science

Stony Brook University

May 2012

Stony Brook University

The Graduate School

Pradeep J. Shetty

We, the thesis committee for the above candidate for the

Master of Science degree, hereby recommend

acceptance of this thesis.

Dr. Erez Zadok, Thesis Advisor
Associate Professor, Computer Science

Dr. Rob Johnson, Thesis Committee Chair
Assistant Professor, Computer Science

Dr. Donald Porter
Assistant Professor, Computer Science

This thesis is accepted by the Graduate School

Charles Taber
Interim Dean of the Graduate School

Abstract of the Thesis

From Tuples to Files: a Fast Transactional System Store and File System

by

Pradeep J. Shetty

Master of Science

in

Computer Science

Stony Brook University

2012

Traditional file systems are designed to store a moderate number of large objects. However, an increasing number of applications need also to store a large number of interrelated smaller objects, to query and update these objects and their relationships, and to maintain consistency and recoverability. Current approaches require applications to interact with multiple interfaces for different data types, making it difficult for programmers to develop error-free, efficient, and portable applications. Researchers have tried to solve this problem by using additional layers of abstraction to unify these disparate interfaces but continue to use traditional storage formats and algorithms that are optimized only for specific workloads.

We have built a transactional system store that can efficiently manage a continuum of interrelated objects from small to large. Our system is based on a data structure, the VT-tree, which is an extension of the log-structured merge-tree data structure (LSM). In this thesis we describe a transactional system store design and implementation that supports high levels of concurrency and larger-than-RAM snapshot-based transactions. We then describe the design of a new transactional file system, KVFS, which is based on our transactional VT-tree. In our system, applications can perform key-value storage and POSIX file operations in the same ACID system transaction, providing support for operations such as file indexing and tagging, meta-data search, and package installation—all in a generic and flexible manner. Our experiments indicate that KVFS's performance is comparable to that of existing native file systems and its elegant transactional interface adds a minimal overhead and supports highly concurrent transactions.

To my Mom, Dad, Sowmya, Praveen, and my family and friends: for their support all through out my life.

Contents

List of Figures vii

List of Tables vii

Acknowledgments ix

1 Introduction 1

2 Background 3

3 Design and Implementation 5
3.1 KVFS Architectural Overview . . . 5
3.1.1 FUSE write-back cache support . . . 6
3.2 VT-trees in KVFS . . . 7
3.2.1 Fragmentation . . . 9
3.3 Snapshots and Transactional Support . . . 13
3.3.1 Transactional Support in KVDB . . . 13
3.3.2 Locking Policy . . . 16
3.3.3 Transactional Support in KVFS . . . 16

4 Evaluation 18
4.1 Experimental Setup . . . 18
4.2 Transactional Performance in KVDB . . . 19
4.2.1 Concurrent Transactions . . . 19
4.2.2 Overlapping Transactions . . . 20
4.3 File System Performance . . . 23
4.3.1 File System Micro-benchmarking . . . 24
4.3.2 File System Macro-Benchmarking . . . 25
4.4 Real Workload and KVFS's Transactional Performance . . . 27
4.4.1 Single Transaction . . . 28
4.4.2 Parallel Transactions . . . 30
4.5 Defragmentation in KVFS . . . 31
Evaluating the Defragmentation Algorithm . . . 31
Analysis of Fragmentation in KVFS . . . 32
4.6 Evaluation Summary . . . 34

5 Related Work **36**

6 Conclusions **39**

Bibliography **40**

List of Figures

2.1 LSM-Tree 4

3.1 KVFS Architecture 6
3.2 Comparing write path vs. scan in LSM and LFS Approaches 8
3.3 VT-tree zone usage before and after compaction, and after defragmentation within a list 10
3.4 Transactional Architecture: showing the dependency list for both isolation modes 15

4.1 Measuring the overhead and the concurrency provided by transactions running in snapshot-isolation mode when the workload is partitionable. 20
4.2 Time taken for completing the operations by the transactions running in serializability mode with varying percentage of overlap 21
4.3 Time taken for commiting the transactions running in serializability mode with varying percentage of overlap 22
4.4 Filebench results for micro-benchmark workloads on SSD 24
4.5 Filebench macro-benchmark workloads result on SSD. Workloads include videoserver, fileserver, varmail, and webserver. 26
4.6 Samba compilation results 29
4.7 Linux kernel compilation results 30
4.8 Parallel Linux kernel compilation results 31
4.9 Percentage of under-utilized zones ($< 75\%$) with different stitching thresholds after inserting 34GB of data in KVFS 33
4.10 Zone usage distribution for runs with different stitching thresholds 34

List of Tables

3.1 Three dictionary formats used within *fs-schema* 6

4.1 Results of Filebench micro-benchmarks in ops/sec 25
4.2 Results of Filebench macro-benchmark workloads in ops/sec 26
4.3 Size and number of files in Samba and Linux source directories before and after compilation. 28
4.4 Results before and after defragmentation 32

Acknowledgments

Professor Erez Zadok has backed me all throughout my stint here at File Systems and Storage Lab. Other than just helping me with technical issues, he also sponsored me to attend lot of prestigious conferences, meet other researchers and shape my research at par with other research projects.

My mother, father, sister, and brother have all always supported me whether it is related to education, sports or financial matters.

Dr. Richard P. Spillane was my mentor here for the first year, and I learned so many technical information from him. This includes transactions, LSM-trees, and even about the basic research. He has helped me become a better researcher.

Ravikant P. Malpani has been a real hard worker on this project. He helped me integrate the locking support for transactions in our database in quick time. He was also responsible for getting most of the benchmarking done for our system.

Binesh Andrews joined this project recently and helped me implement and benchmark the defragmentation tool for our file system. He, along with Ravikant, also helped me implement write-back cache support for FUSE file system.

Justin Seyster has helped me implement the locking code for database when I was undermanned. His code runs amazingly without a single bug.

I would like to thank all the co-authors of our paper related to Quotient Filters and Cascading Filters. This includes Dr. Robert Johnson, Dr. Michael Bender, Dr. Erez Zadok, Dzejla Medjedovic, Pablo Montes, Dr. Richard P. Spillane, and others. This has been one of the coolest research I have involved in.

I would also like to thank all the anonymous reviewers of USENIX ATC who have taken the time to read my paper and gave valuable critics and suggestions. The reviewer comments on this work provided us with a number of excellent directions for advancing our research.

Finally I would like to thank my committee Drs. Robert Johnson and Donald Porter, and my adviser, again, Dr. Erez Zadok.

This work was made possible in part thanks to National Science Foundation awards CCF-0937833 and CCF-0937854, a NetApp Faculty award, and an IBM Faculty award.

Chapter 1

Introduction

Many modern applications need to store structured data along with traditional files to help categorize them. Traditional files includes video clips, audio tracks, and text documents. Structured data represents large numbers of interrelated smaller objects. Examples include media tags such as a photograph light conditions, or an MP3's performer's name, album cover, etc. File systems handle traditional file-based data efficiently, but struggle with the structured data. Databases manage structured data efficiently, but not for large file-based data. Storage systems supporting both file system and database workloads are architecturally complex [11, 20]. They use separate storage stacks or external databases, use multiple data structures, and often are inefficient [41].

Applications want to use system transactions to group accesses to both file-based and structured data consistently. Package managers (e.g., apt-get, yum) can use system transactions to roll back an unsuccessful software upgrade. System transactions also let programmers write highly concurrent code more simply, and improve application security [27, 39]. Existing systems with transactional support require complex kernel modifications [27, 46], need a complete redesign [33], or have high overheads [22]. An architecture that supports system transactions and key-value storage, and can also efficiently process varying workloads, is going to be more complex if its storage design uses multiple, heterogeneous data structures. No known, single data structure is flexible enough to support both database and file-system workloads efficiently.

Read-optimized stores using B-trees have a predictable performance as the working set increases. However, their performance suffers by randomly writing on each new insert, update, or delete operation. Log-structured stores using B-trees efficiently insert, update, and delete; but they randomly read data if their workload is highly random or larger than RAM. Log-structured merge-trees [23] (LSM-trees) have neither of these problems. As with any log structured systems, in LSM-trees random writes become sequential append to a log. Also, commercially used LSM-trees uses Bloom filters [6] to accelerate point queries. Furthermore, LSM-based transactions are inherently log structured. That is why LSMs are widely used in many write-optimized databases [2, 17]. Still, current LSMs do not efficiently process sequential insertions, large key-value tuples, or large file-based data.

We use an extended LSM that we co-developed, called *VT-tree* [38]. The VT-tree supports highly efficient mixes of sequential and random workloads, and any mix of file-system and database workloads. In this thesis we also address several new issues such as fragmentation

faced in VT-tree. Based on VT-trees, we built a database storage engine called *KVDB* and a new file system called *KVFS*—incorporating KVDB into KVFS's architecture with relative ease. KVFS supports both a traditional POSIX API as well as a MYSQL- or BDB-like API for structured data access. Both APIs can be accessed in the *same* system transaction, allowing for more flexible and versatile operations. This enables operations such as file indexing and tagging, meta-data search, and package installation in a generic and flexible manner.

KVFS supports larger and faster system transactions by avoiding double-writes and by indexing its journal. KVFS runs in user-level (using FUSE [44]), is portable and easily maintainable, and can be restarted and recovered independently of the OS. KVDB is cache friendly, I/O efficient, and works well for random and sequential workloads. Excluding inherent FUSE overheads [28], KVFS's performance is comparable to Ext4.

The rest of this thesis is organized as follows. Chapter 2 describes some of the data structures used in databases and file systems. We introduce KVFS's design and describe the VT-tree and KVFS's transactional architecture in Chapter 3. Our evaluation in Chapter 4, compares the performance of KVFS to Ext4, and also measures the efficiency of KVFS's transactional interface. Related work is discussed in Chapter 5. We conclude and discuss future work in Chapter 6.

Chapter 2

Background

KVFS is a file system implemented as a FUSE [44] driver on top of a user-level database called KVDB. To ensure that KVDB has the highest insertion, update, and delete throughput possible, we chose the log-structured merge-tree (LSM-tree) as our basic data structure. LSM-trees write sorted and indexed lists of tuples to storage, which are then asynchronously merged into larger sorted and indexed lists through a process called *compaction*. LSM-trees can sustain high random update, insert, and delete throughputs while deamortizing compaction more effectively than a naive log-structured or copy-on-write [15] approach [45]. A log-structured approach allows for efficient caching and a highly concurrent transactional design, which we describe in Chapter 3.

However, the LSM-tree is not as efficient for sequential insertions [37] (see Chapter 3). VT-tree [38] is an extended LSM-tree with optimizations for sequential insertion. We first introduce the LSM-tree here and then describe VT-tree and its usage within KVFS in Chapter 3.

LSM-Trees The LSM-tree is an alternative to the B-tree and the log-structured radix [15] or B-tree [13, 47] for storing structured data in secondary storage devices [3, 23]. LSM-trees offer one to two orders of magnitude faster insertions in exchange for one to ten times slower point queries and scans [3]. They are typically used in scenarios where large data sets are automatically generated and queries can be parallelized, such as Web search [8]. LSM-trees rely on asynchronous compaction to keep lookup and scan times bounded as the number of elements in the LSM-tree increases. There are a variety of compaction strategies [40], and when used with Bloom-filter–like data structures [4, 6], losses in lookup latency with respect to B-trees can be recovered.

Figure 2.1 illustrates the operation of a basic LSM-tree. An in-RAM buffer called a *memtable* (e.g., a red-black tree or skip list) holds recently inserted items. When the buffer is sufficiently full, it is flushed to disk as a single list of sorted tuples. Along with this sorted list is a secondary index and a Bloom filter to accelerate scans and point queries. A list plus its secondary index and Bloom filter is called an *SSTable* [8]. Point queries simply search all lists for the value belonging to a key, using the Bloom filters and secondary indexes to avoid fruitless searching and I/Os wherever possible. LSM-trees also allow us to read many tuples in series, or find the tuples that come immediately after a queried key. This kind of query is called a *scan*. To perform a scan, cursors are placed in each list; the buffer and all lists are

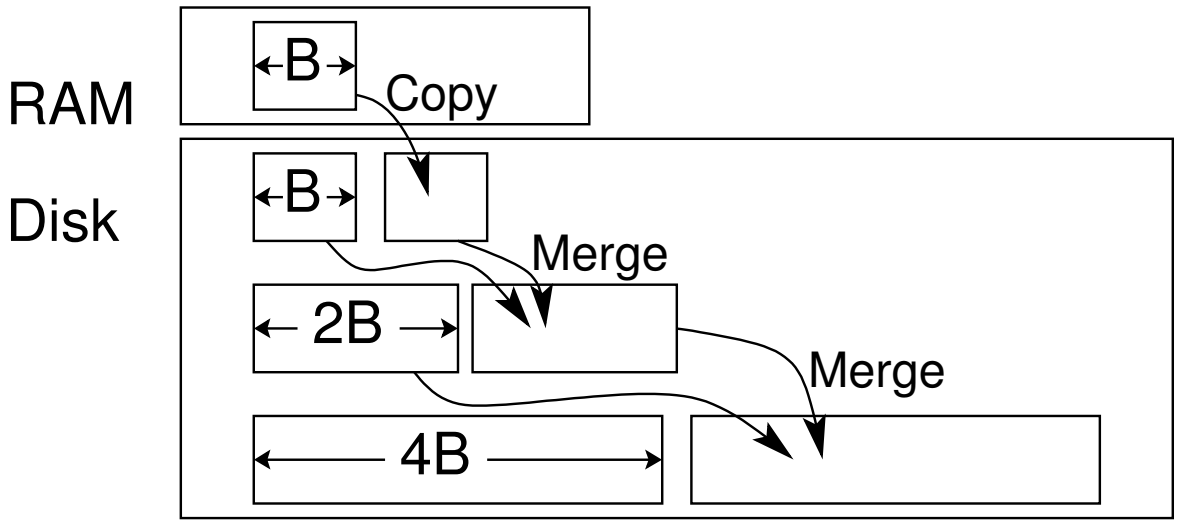


Figure 2.1: LSM-Tree

merged into a single sorted view by incrementing and merging the result of the cursors.

As the number of tuples inserted increases, the number of on-disk lists increases too. To ensure that lookup and scan performance is always bounded, some or all of the on-disk lists may be compacted into one sorted list when a buffer is flushed. This is called a *minor compaction*. The exact lists chosen to compact at the time of buffer flush depends on the compaction algorithm. The compaction algorithm also affects how many lists must be searched to place cursors for a scan or to find a tuple [40].

The main problem with the LSM-Tree is that tuples must be copied for each compaction. Each tuple is copied anywhere from $\lceil \frac{\log_K N}{B} \rceil$ to $\frac{(K-1)\log_K N}{B}$ times depending on the compaction algorithm. Here, B is the number of tuples that can fit in the Disk-Access Model (DAM) [3] block size for the disk; K is the maximum allowed number of on-disk lists in a given size class before a compaction is required. This is depicted in Figure 2.1 where the contents in RAM are first serialized to an on-disk SSTable of size B; then the contents are re-written to larger SSTables by subsequent minor compactions. These repeated copies are a problem if we wish to use LSM-trees for file system data. Storing 16 times the size of the in-RAM buffer in a day would require that data to be written 2–8$\times$. A traditional file system would write it only once and would not move it until the next defragmentation. KVFS tries to achieve the same by using VT-tree and supporting defragmentation in VT-tree.

Chapter 3

Design and Implementation

We have two design goals for KVFS: (1) versatile support for mixed workloads while maintaining simplicity and (2) efficiency especially given its transactional design. First, we designed a storage system capable of supporting both file system and database workloads efficiently yet remain architecturally simple. KVFS achieves this by using the *VT-tree* [38] data structure, an extension to LSM-trees, to improve sequential-workload and large tuples performance. Second, we designed an efficient transactional architecture to avoid most double writes, be highly parallelizable for partitionable workloads, and support larger-than-RAM transactions. Application using KVFS can group together a sequence of POSIX and key-value storage operations into a single atomic transaction.

We describe KVFS's architecture first in Section 3.1, then discuss how we use VT-trees in KVFS in Section 3.2, and our transactional architecture in Section 3.3.

3.1 KVFS Architectural Overview

Figure 3.1 shows KVFS's basic architecture. KVFS uses FUSE [44] to provide a POSIX-compliant file system interface to traditional applications. A user read or write request is passed on to FUSE by the VFS. The request goes to KVFS layer running as a user-space daemon. KVFS translates the request into one or more key-value operations sent to KVDB, which implements a transactional database storage engine based on the VT-tree. KVDB performs all necessary I/Os using a series of `mmap`s of a disk file stored on a back-end Ext4 file system. The response follows the same path as the request, in reverse. FUSE provides kernel-level caching for file data pages and attributes; cached information thus eliminates the need for user-level upcalls; this results in maintaining KVFS's high performance for in-cache workloads. However, transaction managers typically require notification for all reads, a challenge for consistency if notifications are missed for cached data. In Section 3.3 we describe how KVFS addresses this challenge. We also modified FUSE to support write-back caching and we explain this in detail in Section 3.1.1.

When initializing KVDB, we create a schema with one or more dictionaries, each backed by a VT-tree. KVFS defines three VT-trees in a single KVDB schema, called *fs-schema* to support file system operations. The three dictionary formats, also shown in Table 3.1, are:

1. *nmap* for namespace entries, similar to `dentries`; here, the *path-component* of a file

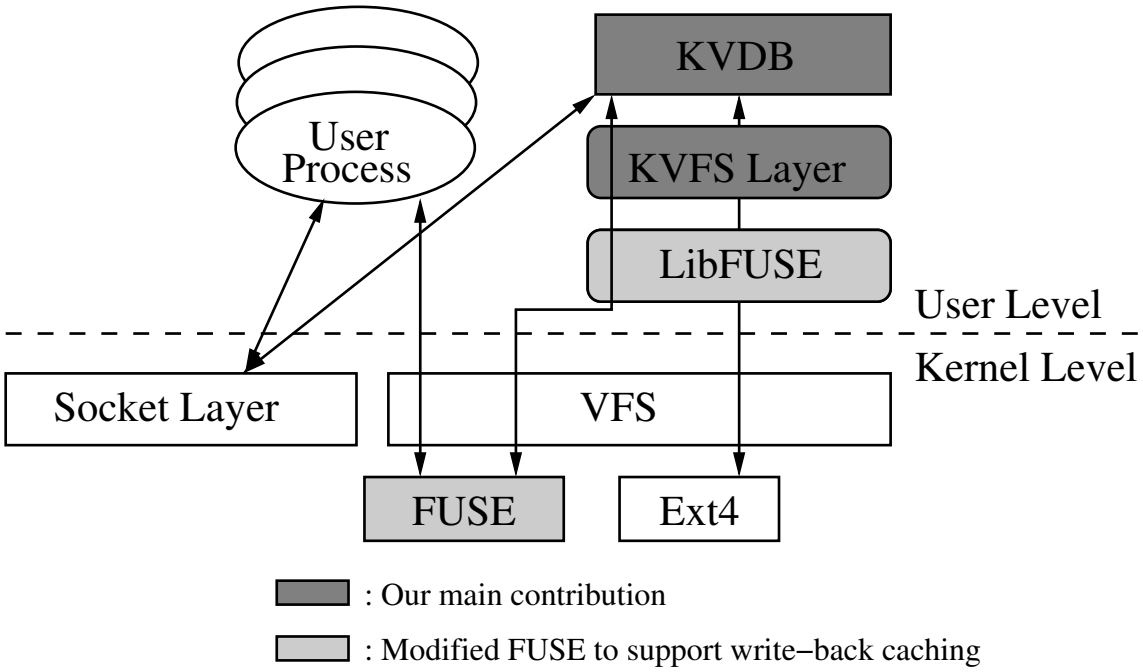


Figure 3.1: KVFS Architecture

VT-tree	Format
nmap	$\langle\{$*parent-inode#*, *path-component*$\}$, *inode#*$\rangle$
imap	$\langle$*inode#*, *inode*$\rangle$
dmap	$\langle\{$*inode#*, *offset*$\}$, *data-block*$\rangle$

Table 3.1: Three dictionary formats used within *fs-schema*

and its parent directory's inode# form the key, and the value is the file's inode#.

2. *imap* for storing inode attributes; and

3. *dmap* for the files' data blocks.

KVFS supports storing any external meta-data associated with a file by creating a new dictionary format within the same schema in the KVDB.

KVFS also provides a key-value interface directly to KVDB using sockets; this is useful for databases and applications that process random-access workloads. KVFS supports access to both the POSIX and key-value interface in the *same* system transaction. This allows an application, for example, to create a new MP3 file iff its corresponding ID3 tag was successfully added to a music-search database. We explain this further in Section 3.3.

3.1.1 FUSE write-back cache support

KVFS uses FUSE [44] to support POSIX operations. Using FUSE requires two additional context switches and buffer copies than running on a native file system. This results in around

2× overhead compared to the native file system performance [28]. The FUSE kernel module caches read pages, but writes are immediately sent to the FUSE server running in user space. Supporting a generic write-back cache in FUSE kernel is complex because of two reasons:

1. The entity responsible for accepting or rejecting the writes is not the FUSE kernel, but the user-level file system instead. A deadlock situation could arise while flushing dirty pages if the user-level file system is swapped out due to memory pressure [28].

2. If the user-level file system intends to be POSIX-compliant, all the dirty pages for the file should be synchronized to disk during a close request.

In earlier versions of FUSE, all file write requests were split into 4KB (page size) writes, requiring additional context switches for every 4KB of writes. FUSE started supporting *big-writes* option from the 2.8.0 FUSE library version (libfuse) and the 2.6.27 Linux kernel version. The *big-writes* option allows file writes to be transferred to user-level file system in larger chunks. However, this write needs to be a single contiguous file write from the application. The reason is that even the recent versions of FUSE do not cache the writes; it only transfers them as a big chunk. FUSE limits the size of these chunks to 128KB to reduce the kernel memory consumption and the possible deadlock situation. If the kernel is under memory pressure due to a large number of such in-transit pages, a deadlock may occur if the user-level file system process has been swapped out [28]. The *big-writes* option significantly reduces the number of context-switches for applications that write to a file with write sizes larger than 4KB. The *big-writes* option is not of much help for applications writing to random files using smaller writes.

We extended the *big-writes* option to support basic a write-back cache in kernel. When a write request comes for a file residing in the user-mode file system, we now cache the writes in the file's page-cache. This avoids immediate write-back to the user-mode file system. The page is marked as dirty in the page-cache and stays as a hot page in the kernel's page-cache if it gets updated frequently. The page is written back by the `pdflush` kernel thread when the page becomes eligible for a write-back. Additionally, we ensure that all the dirty pages associated with the file are flushed to the user-mode file system when the file is closed. When the dirty pages become eligible for write-back, we transfer them in chunks of 128KB. The pages within the chunk need not be contiguous and they can also belong to different files. We also limit the total page cache used by the FUSE user-file system for dirty pages to 256KB to avoid the possibility of a deadlock during write-back by `pdflush`. This write-back implementation is not ideal and does not provide all the benefits of a write-back cache as used by existing in-kernel file systems like Ext3 and XFS; still, results from our initial evaluation are promising.

3.2 VT-trees in KVFS

KVFS uses *VT-trees* [38] that we co-developed to achieve its first design goal: build a storage system capable of supporting both file system and database workloads efficiently and yet remain architecturally simple. LSM-trees are widely used in many write-optimized databases [2, 17]. We call an LSM-tree with extensions for efficient sequential insertions

and large tuples, a *VT-tree*. During a minor compaction, The VT-tree merges two lists into a larger list without copying every single tuple as a naïve LSM-tree would, a process called *stitching*. VT-tree's stitching avoids copying largely unmodified or sequential data during a minor compaction, allowing it to handle file system data more efficiently. In KVFS, the *dmap* benefits from VT-tree's stitching property as the *dmap* holds file data that can be large and sequential. File system meta-data like *imap* and *nmap* take advantage of the LSM-tree nature of the VT-tree. *nmap* holds the directory entries (dentries). Entries in *nmap* are sorted and always get merged as with the LSM-tree. This allows the dentries under a directory to be laid out contiguously on the disk even if they are created at different point in time. This makes the listing of large directories (e.g., with `/bin/ls`) in KVFS faster.

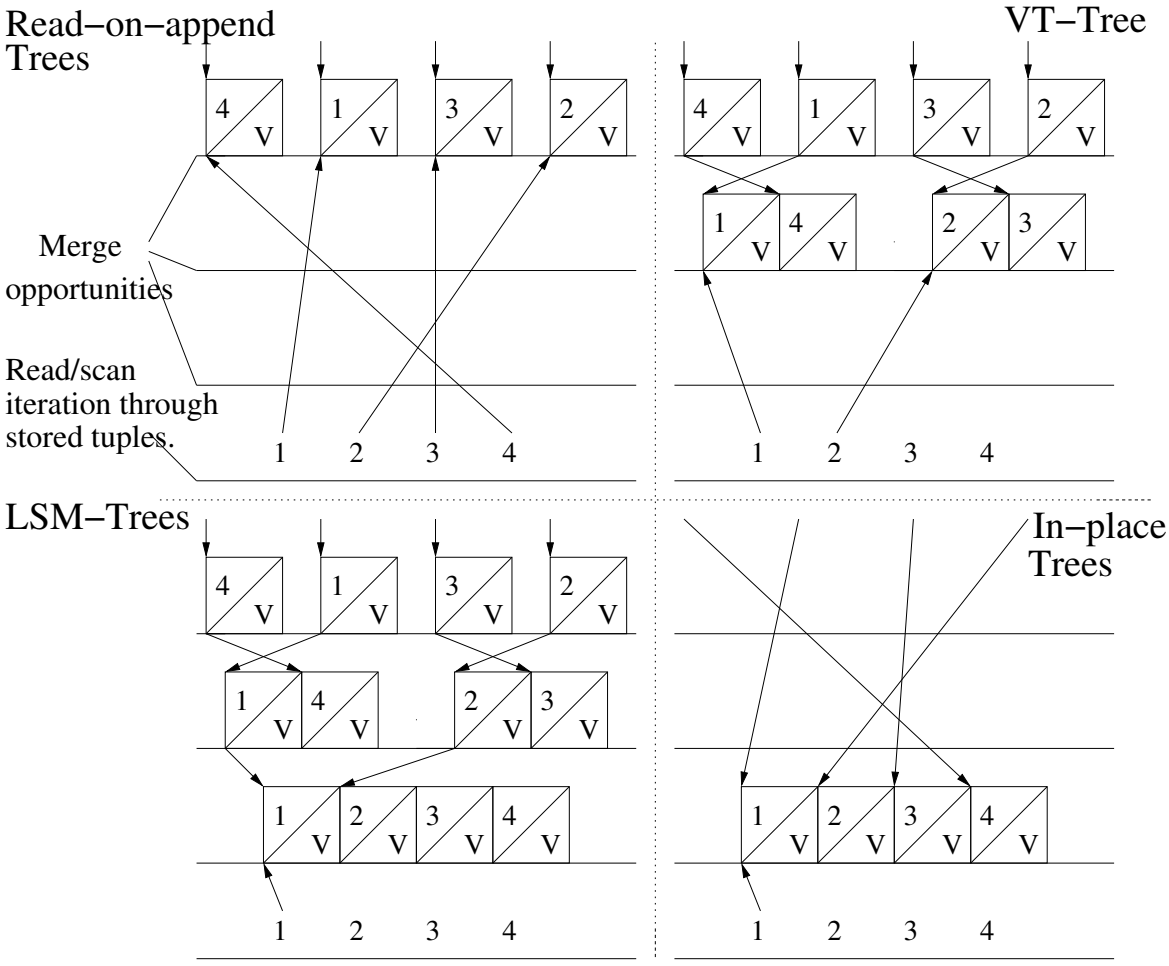


Figure 3.2: Comparing write path vs. scan in LSM and LFS Approaches

A VT-tree can be tuned to behave like a log-structured file system (LFS) and avoid writing more than once; alternatively, it can behave like a log-structured merge tree, and guarantee upper bounds on the number of seeks required to perform a scan. Also, intermediate configurations are possible, to trade off seek for scan performance for repeated writes performance. Thus, a VT-tree can span the entire continuum from an LSM-tree to an LFS.

Figure 3.2 compares four types of trees: read-on-append trees (e.g., log-structured B-trees), LSM-trees, in-place trees (e.g., B-trees), and the VT-tree. Each tree is depicted show-

ing the path a tuple took to its final destination. Horizontal lines represent an opportunity to sequentially copy a tuple into a larger list during a merge. Only LSM-trees take every opportunity to do this; they copy every tuple $\log_2 N$ times until all tuples are physically contiguous on disk (bottom level). None of the other trees take every opportunity to compact tuples; this allows them potentially to write more efficiently. In-place trees directly flush each tuple to its final location, but perform random writes to do so. Read-on-append trees sequentially write every tuple to disk initially, but require random reads to scan in sorted order. LSM-trees sequentially read as efficiently as in-place trees; and, although LSM-trees perform $\log_2 N$ writes for each tuple, this can still be much faster than a single disk seek.

The VT-tree is a compromise between an LFS-based approach and a naïve LSM-tree approach: contiguous tuple sequences smaller than the stitching threshold are copied, but once the sequence length exceeds this threshold, tuples are no longer copied. Users can set the stitching threshold large enough so that the cost to perform a seek is equivalent to the cost of sequentially scanning a stitched region: thus performing sequential reads is never worse than a factor of two of reading the same amount of data laid out contiguously on disk.

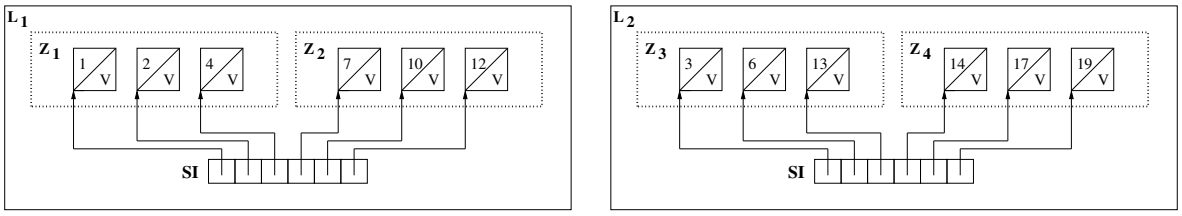
3.2.1 Fragmentation

Using VT-tree in KVFS comes with a cost: it results in fragmentation as with any similar log-structured file systems [30]. An LSM-tree typically reads in lists of sorted tuples, and then writes a sorted tuple stream to a new merged list through a process called compaction. In a widely used compaction algorithm [40], a merge occurs $\log_K N$ times, so a tuple is copied up to $\log_K N$ times. It is necessary to merge smaller lists into larger lists to bound the total number of lists in each size category. Stitching in VT-tree helps avoid such expensive list merging and copying in some cases. If the data is already sorted or sequential, stitching leaves the data in-place and uses back-pointers to refer to that data from the new list. However, stitching every region containing contiguous tuples that is above some threshold can introduce fragmentation. This is because although some tuples are left in place, others are copied into the new list, and their original physical location remains unused: this is shown in Figure 3.3. To use VT-tree in a file system, a defragmentation tool is necessary, but the existing VT-tree implementation does not have support for defragmentation [38]. We have designed a simple defragmentation tool that can be run in online or offline mode. Our defragmentation tool mainly focuses on reclaiming the unused space and also preserves the sequentiality of the existing data.

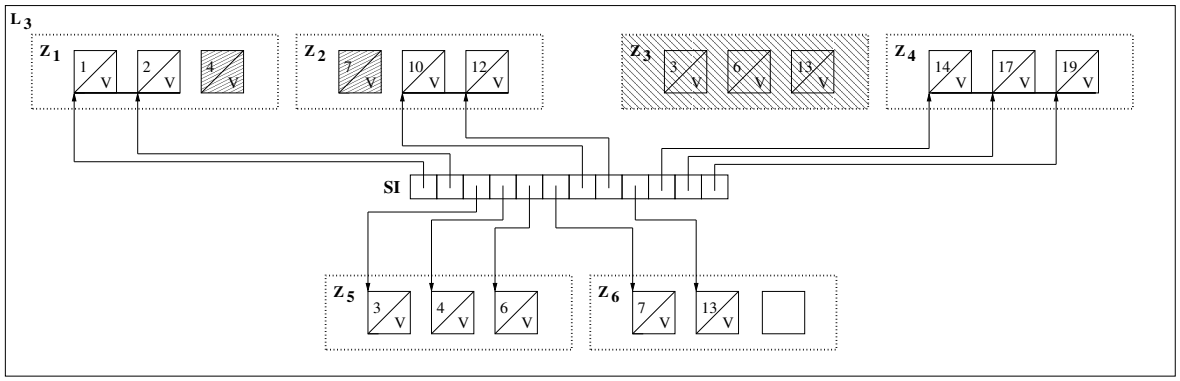
As shown in Figure 3.3, each list (L_is) in VT-tree is made up of one or more *zones*, where a zone is a minimum allocation unit in our implementation. Zones are usually 8MB large. Each list has an associated secondary index to help reduce I/Os during lookups. This index refers to the same original tuples, using the same original key, but whose value is the physical offset or back pointer to the original tuple's location. Typically, a secondary index entry points to a contiguous scan (extent) of two or more elements in the original list. The number of elements pointed to by a secondary index entry is one in this example and the stitching threshold is of two elements.

During the compaction process in VT-tree with stitching enabled, lists L_1 and L_2 are merged and L_3 is the resulting compacted list. List L_1 is made up of two zones, Z_1 and Z_2; L_2 is made up of zones Z_3 and Z_4. Each of the lists have six tuples in them. After merging,

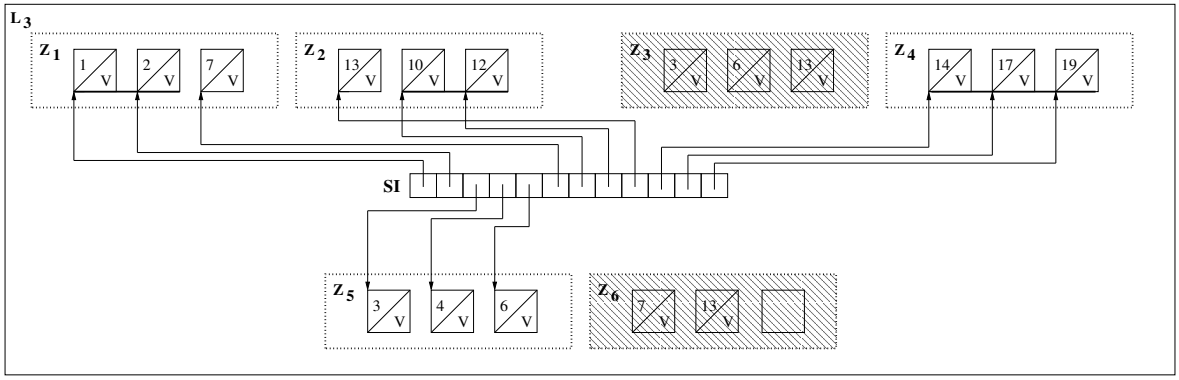
Before compaction



After compaction



After defragmentation



L : List Z : Zone SI : Secondary Index : Unused tuple space : Deallocated zone

Figure 3.3: VT-tree zone usage before and after compaction, and after defragmentation within a list

we stitch regions consisting of contiguous tuples whose length is at least that of the stitching threshold. In this example, the stitching threshold is two and the stitched tuples are $\{1, 2\}$, $\{10, 12\}$ and $\{14, 17, 19\}$. The rest of the tuples are copied to the newly allocated zones Z_5 and Z_6 in the list L_3. As seen in Figure 3.3, zones Z_1, Z_2, and Z_4 remain allocated and are used by the list L_3. Zone Z_3 has been freed as all the tuples in it have been copied to the zones Z_5 and Z_6 while compacting the lists L_1 and L_2. After the merge, zones Z_1 and Z_2 show some internal fragmentation. L_3 ends up consuming five zones instead of four with a 20% of space wastage.

In KVFS, the stitching threshold is usually between 32–256KB and zones are 8MB in size. A typical file system supporting capacity of 128GB will have 16K zones. Our experiments, described in Chapter 4, show that most of the zones in a half filled KVFS contain only few stitched regions, and are only 20–40% utilized resuling in a lot of fragmentation.

Defragmentation algorithm. A defragmentation process needs to consider at least the following three parameters: the amount of data needs to be moved, preserving the sequentiality of the existing data, and the total time taken for the completion. Other important parameters to measure the effectiveness of a defragmentation algorithm are the amount of space reclaimed and any improvement in subsequent scan performance. As seen in Figure 3.3, after compaction, zones Z_1 and Z_2 in list L_3 are fragmented, and we consume an extra zone Z_6 to store the copied tuples: a 20% space wastage. So, VT-tree's defragmentation algorithm first needs to find the amount of space that can be reclaimed and the candidate reclaimable zones. In LFS [30], the defragmentation process is called segment cleaning and segments used within LFS are similar to zones in KVFS. Segments in LFS can have some dead or stale data along with the live or active data. In the *greedy* algorithm used by LFS, the reclaimable segments are chosen based on the amount of active data they hold. If a segment has less active data, it is easy to clean that segment as it has to move only the active data to some other segment. As noted by LFS's authors, this greedy algorithm suffers when the rate of change of data follows a bi-modal distribution between hot and cold segments. They proposed another algorithm called *cost-benefit* that considers the cost-to-benefit ratio of cleaning a particular segment. There, the authors looked at the youngest block within the segment to estimate how long the segment is going to be untouched (remain clean); segments containing older data have a higher benefit if cleaned.

As with any log-structured systems, in VT-tree, data within a zone never gets modified; if required, it is copied to another zone in a new list during compaction. A zone can get fragmented during a compaction in two cases: (1) when only few stitched regions can be formed in this zone or (2) if an interleaving tuple is found for some of the existing stitched regions in this zone and at least one stitched region stays unaffected. This interleaving tuple can be anything: an update (i.e., block in a file being overwritten) or a deletion of an existing tuple, or even an unrelated tuple to the existing tuples in the stitched region. These two cases can happen with zones that are either hot or cold. In VT-tree, hot zones are part of list at the top, which are small and recently created. Cold zones are usually part of larger and older lists. Lists at the top participate in the compaction process more often than the older lists at the bottom. Thus, cleaning the zones of these lists at the top may not be beneficial. Also, lists at the top contain a smaller number of zones and do not contribute to the majority of the fragmentation anyway.

We devised an algorithm that orders the lists based on the amount of space that can be reclaimed. We also consider the location of the list in the hierarchy. Within each list selected for cleaning, the algorithm finds the candidate zones for reclaiming. To do that, we scan through all the zones and assigns a weight to each of them. This weight is calculated based on the amount of sequential data each of these zones already have. We use the secondary indexes to find the range of sequentiality of the data and we can calculate the weight without performing any I/O. The weight is calculated as the ratio of total length of contiguous scans to the number of contiguous scans in the zone. This ensures that having one large stitched region has more weight than many small stitched regions in a zone. Zones with a lower weight are the candidate zones for reclaiming space. The data from these candidate zones is moved to the other zones in the order explained below. Using this algorithm, we do not need any extra zones than what is already being used, allowing us to use the same methodology in an out-of-space condition that could result due to fragmentation. We also avoid overwriting any existing data in a used zone as that would otherwise make the recovery complex and also require journalling each overwrite. Here, we also preserve the sequentiality of the existing data and perform only the minimally required amount of data movement to reclaim the maximum space possible.

With our method, a zone containing lot of active data may get chosen for cleaning. This is possible only if the zone contains lot of random data that interleaves between the stitched regions in some other zones: this is less likely in practice. In Figure 3.3 we show only one list, L_3, participating in the defragmentation. Within this list, zones Z_4 and Z_5 have the highest weight of three as both have only one large contiguous scan of three tuples. Zones Z_1 and Z_2 have a weight of two, and zone Z_6 has a weight of one. Since we can only reclaim one zone here, Z_6 is our candidate. We end up moving the tuples from the zone Z_6 to zones Z_1 and Z_2, each of which has one unused tuple space. While selecting zones for the data to be copied, we can either use a first-fit or a best-fit method. The best-fit technique improves the sequentiality and it only requires us to go through secondary indexes to determine the best fit without causing any I/Os, but may result in more random writes while copying the data. Whereas first-fit is easy to implement, it may sometimes spread the data more randomly if the holes are small and spread across many zones.

Our current implementation of the defragmentation runs in an offline mode. During the normal run, KVFS keeps track of the fragmented zones, allows users to get the information on when to run the defragmentation tool. KVFS needs to be booted in *defrag* mode to kick off the defragmentation process. While in defrag mode, KVFS goes through each of the VT-trees in its schema and runs the aforementioned defragmentation algorithm. Within each VT-tree, we run the defragmentation algorithm on all the fragmented lists and in parallel. Our experiments, described in Chapter 4, show the efficiency of this algorithm. Although this simple defragmentation algorithm works well, the framework we have implemented allow us to use a different method for weight calculation if required. Our defragmentation tool can be easily extended to run online. KVFS already keeps track of the zone usage during a normal run. After each minor compaction, we can check if the resulting new list is fragmented and kick off a defragmentation thread in the background if the list is fragmented. As we never overwrite any existing data in the zone, reads can be served in parallel. Once the data has been moved and a new secondary index is constructed, we can stop the reads, swap the secondary indexes, and then let the reads use the new secondary index. However, any future

compactions involving this particular list have to wait until the defragmentation is completed.

Another simple method of defragmentation is to cause a minor compaction of a fragmented list with that of an empty list, with a stitching threshold set to some large value. The resulting output list has no stitched regions and hence no fragmented zones. All the zones used by previous list become free and reusable. This method in fact undoes the effects of the stitching that took place during previous minor compactions, before the data reached this list. Although it appears to defeat the purpose of having stitching in the first place, this is not true. Stitching would already have saved some extra copies that would have otherwise happened by now. The only disadvantage is that this method requires free zones to perform a minor compaction, which may not be always available. Comparing these methods and exploring others are subject to future work.

3.3 Snapshots and Transactional Support

KVDB is based on the VT-tree. Snapshots and transactions are an inherent feature in VT-tree, as with any other log-structured system [36]. Snapshots in KVDB are transaction aware. We implemented them using the same machinery used in KVDB.

KVDB initially contains a single schema called the *main-line* schema, which consists of one or more dictionaries, each backed by a VT-tree. KVDB supports snapshots by snapshotting the main-line schema, which implicitly snapshots all the VT-trees part of it. Creating a snapshot makes the main-line schema read-only as expected and creates a new writable schema with empty VT-trees: this is now the new main-line. The new main-line schema hosts only new data and any modification to the existing old data. Lookups and scans may need to visit older snapshots, requiring KVDB to maintain a dependency of this new main-line schema with the older snapshot. A lookup first visits the current main-line schema and on an unsuccessful search, it will follow this dependency list, doing lookups on the snapshots from recent to old until the search is successful or the end of the list is reached.

Since snapshots in KVDB only contain the modifications from the previous one, supporting a snapshot-based incremental backup and replication is efficient and easy—a feature that many modern storage systems try to include.

3.3.1 Transactional Support in KVDB

KVDB supports ACID transactions in two different modes of isolation:

- Serializability: this is the highest degree of isolation [14]. KVDB supports serializability using strong, strict two-phase locking (SS2PL). We discuss the choice of SS2PL later in this Section.

- Snapshot isolation: here, transaction reads see a consistent snapshot of the KVDB that was taken when the transaction started. Conflict resolution here can be slow or the applications using this may encounter lost updates. So, this mode is provided only for applications to run in parallel, as long as they have no overlap in their operations.

Transactions using these two modes differ in the locking semantics they follow, and also in their snapshot-dependency list. In snapshot isolation, the dependency list begins with the snapshot that was taken at the time the transaction started; all the reads in this transaction see only the data generated before the transaction's start time. In serializability mode, the dependency list always starts with the current main-line schema, allowing the transaction to read the most recently committed data by other transactions, even if these transactions were started later.

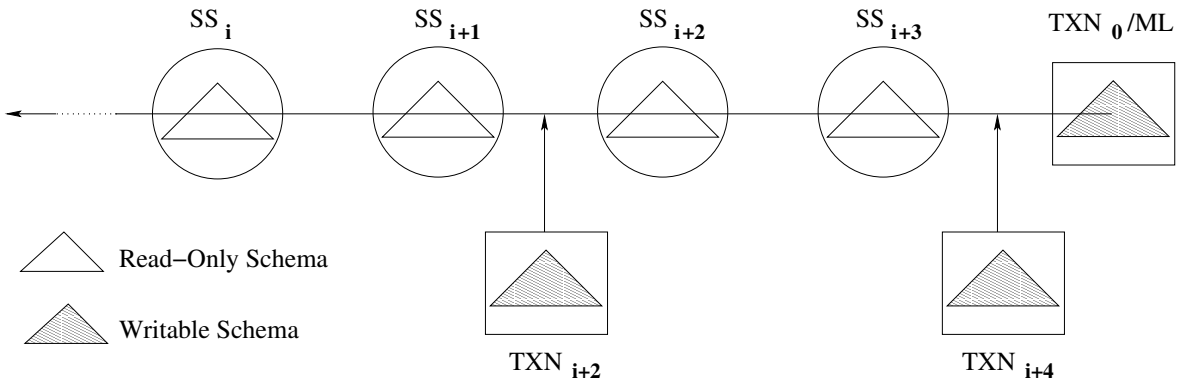
KVDB provides system transactions to any process that calls `txn-begin`. This branches the current system state into a private writable schema, and the *main-line* or the writable portion of the system; see Figure 3.4(a). The *main-line* is visible to all other processes. The process can now modify this writable snapshot without affecting any other process in the system. When the process calls `txn-commit`, the separate branch is merged with the main-line, and locking within KVDB eliminates merge conflicts. Having private snapshots and caches for each transaction improves KVDB's concurrency and parallelism. This transactional architecture is also best suited for applications running on multi-core architectures, such as with the Corey [7] kernel.

When KVDB is initialized, the main-line schema is part of a default, main-line transaction. Clients wanting to enforce ACID properties need only begin a new transaction and commit it once all operations are completed. Transactions can be multi-threaded: KVDB ensures consistency using per-transaction locks. Writable snapshots can be larger than RAM, and can continue to exist after the process that initially branched them terminates (supporting check-pointing). Since writes to the private writable schema are updates to a VT-tree, which is an LSM-tree, no redo records are needed to abort a transaction: simply remove the newly created lists within the schema. Also, transactions can exceed RAM sizes without blocking on packing or segment cleaning, because the LSM-tree maintains an implicit index to support lookups in no worse than $K \log_K N$ I/Os (often only one I/O).

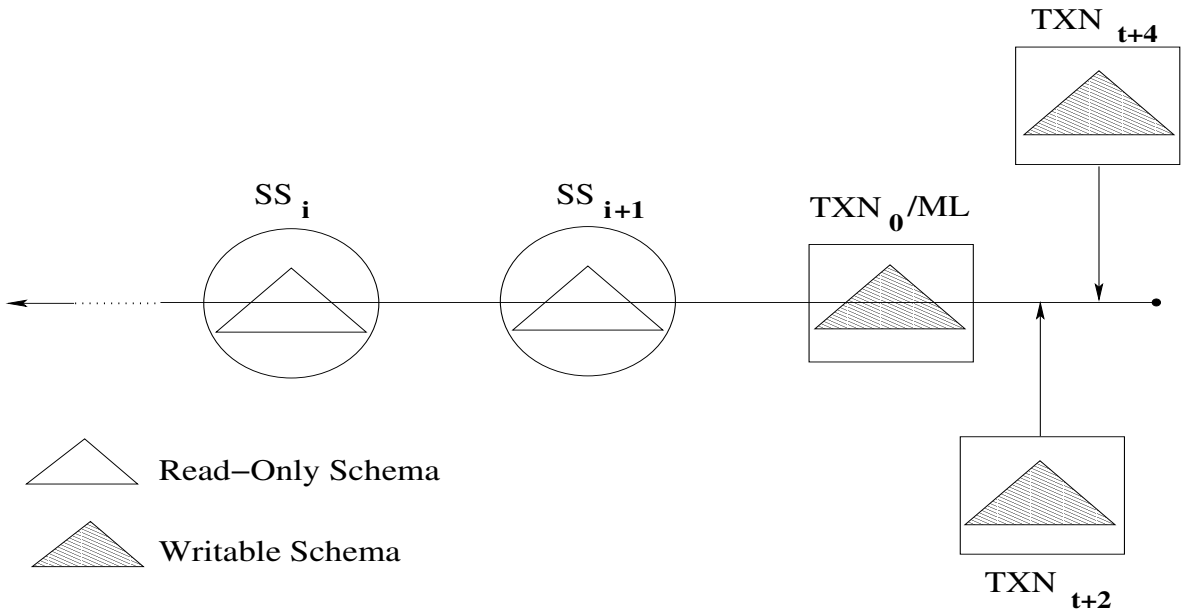
Figure 3.4(a) shows the KVDB in snapshot-isolation mode with two active transactions—TXN_{i+2} and TXN_{i+4}—and a default main-line transaction, TXN_0. All the writes and updates in the transaction go to its writable schema. Reads and scans within the transaction may need to visit all the dependent snapshots. For reads and scans, we first construct a *transaction-specific view* (TSV) containing all the schemas the transaction depends on. For TXN_{i+2}, the TSV contains its writable schema, snapshots SS_{i+1}, SS_i, and any dependent snapshots. Reads within TXN_{i+2} are looked up in all the snapshots (as ordered by the TSV) until the desired element is found or no more snapshots are left. We reduce I/Os during lookups using Quotient Filters (QFs) [4]. Similarly, the TSV for the main-line transaction TXN_0 consists of its writable schema, snapshots SS_{i+3}, SS_{i+2}, SS_{i+1}, SS_i, and any dependent snapshots. If no transaction is directly dependent on a particular snapshot, that snapshot can be merged with the next one to improve read performance; here, SS_i can be merged into SS_{i+1} unless it is a user-created snapshot.

In Figure 3.4(b), we show transactions in the serializability mode. As we see, all the transactions have dependency on the current main-line transaction. The TSV for TXN_{t+4} contains its writable schema, main-line transaction TXN_0's writable schema, snapshots SS_{i+1}, SS_i, and any dependent snapshots.

We now discuss the effect of operations `txn-begin`, `txn-commit` and `txn-abort` supported by KVDB for transactions.



(a) Transactional view in snapshot-isolation mode



(b) Treansactional view in serializability mode

Figure 3.4: Transactional Architecture: showing the dependency list for both isolation modes

KVDB txn-begin. In snapshot-isolation mode, this operation creates a new snapshot, with two effects. (1) We mark the main-line schema read-only, which may flush in-RAM VT-tree buffers to on-disk lists. KVDB can defer this until memory is full. (2) We create a new writable schema and make it part of the main-line transaction. `txn-begin` also creates a new writable schema for itself; lookups and scans on this transaction may visit older snapshots. So we register the new schema's dependency on the previous main-line schema.

In serializability mode, txn-begin need only to create a private writable schema for itself and register the dependency with that of the current main-line. We do not need to snapshot the main-line.

KVDB txn-commit. This operation evicts dirty key-value pairs from in-RAM buffers of each VT-tree's in its writable schema, into the main-line schema. Then, we move all on-disk lists from each of its VT-tree's to the corresponding main-line schema's VT-tree. We quiesce the main-line transaction while moving each list (a simple pointer-swap operation). The most recently committed transaction's lists are placed highest, and so they are found before their other (now stale) versions. We then discard the writable schema and the transaction structure. Finally, we flush the journal to ensure durability and recoverability.

KVDB txn-abort. This operations requires freeing up the space used by in-RAM buffers and then discarding the writable schema and the transaction itself.

3.3.2 Locking Policy

Locking within KVDB is pessimistic and two-phase. Processes acquire locks on resources before reading from or writing to them, and they release these locks on commit or abort. Locking need not be pessimistic: read operations could be serialized to disk in sorted order along with inserts, and conflicts could be detected by merging the lists of a committing writable snapshot (transaction) with the main-line, and thus detect conflicts and lost updates. Since KVDB supports larger-than-RAM transactions, these lists can be large and conflict resolution can be time consuming as well. However, the trade-off between serializing reads and avoiding immediate contention with a pessimistic model is unclear; it is outside the scope of this thesis and a subject of future work.

Locking is further optimized by using a range tree to store locks. Processes can lock subsets of the key-space, and can then avoid checking locks for all operations within this subset. This allows partitioning of the caches and permits completely parallel operations on transaction caches and the main-line cache—improving concurrency—without even synchronization on locking primitives. We use this feature to lock directories in KVFS, by specifying a prefix, and then no further locking is required to transact on the contents of that directory. Lock denial is handled by allowing transactions with a higher priority to proceed on contention. In this case, the lower-priority transaction can choose to wait until the conflicting higher-priority transaction commits or aborts, or the lower-priority transaction can choose to restart at a later point in time. Transaction priority is the same as that of the process that started it. Locking support in KVDB comes with deadlock detection. On detecting the deadlock, a transaction with lower priority is aborted and all its resources are released along with all of the locks it has acquired until then.

3.3.3 Transactional Support in KVFS

The KVFS layer uses the FUSE [44] low-level interface to translate FUSE file system requests into key-value requests. With KVDB providing an efficient transactional interface, making KVFS a transactional file system required no kernel changes. KVFS provides three interfaces that we implemented using `setxattr`: (1) `txn-begin`, (2) `txn-commit`, and (3) `txn-abort`. These are available as a user library for user applications.

KVFS txn-begin. Each `txn-begin` creates a snapshot of the file system. An application can then safely modify this private snapshot. Applications must `chroot` into the snapshot, which is automatically done during `txn-begin`; child processes inherit the parent's current working directory as per POSIX [16]. This enables multiple-process transactions (e.g., a transactional shell script): child processes automatically inherit their parent's locks and isolation. No extension to the `task` struct or other process information are required within the OS [27, 39]. Other than calling these APIs, applications require small changes (often to remove obsolete code).n Our system manages all remaining transactional aspects. Upon `txn-begin`, we embed a transaction ID (TID) into the root directory inode# of new snapshot. Future operations start from this root directory, allowing KVFS to find the TID from the inode# part of the requests. KVFS uses two types of inode numbers. (1) The *advertised inode#*, with the TID embedded. These are the inode numbers that application see and all system requests use. They have a side benefit of creating a separate kernel inode cache for each transaction. (2) A *plain inode#*, used internally by KVFS to organize, store, and retrieve files, pages, and directory entries. This allows reads to span across multiple snapshots and also merge the snapshots to improve the performance, without going through the entire data within the snapshot.

KVFS txn-abort and txn-commit. `txn-abort` recursively unlinks this temporary snapshot once all processes have `chdir`ed out of the snapshot. Conversely, `txn-commit` merges the snapshot's contents into the main file system view shared by other and non-transactional processes. In both cases, KVFS invalidates the corresponding kernel cache before returning control back to the application. This ensures that applications have to acquire fresh read locks on committed/aborted items, and not use stale caches.

FUSE cache support. FUSE's read performance is comparable to in-kernel file systems, but only when its kernel-caching feature is enabled [37]. Kernel caching, however, conflicts with user-level transaction managers that must ensure proper locking before reads. KVFS is able to leave read caching enabled in FUSE while still maintaining consistency, and avoid lost updates. In KVFS, reads are specific to a particular writable snapshot, or the main-line. If the read is to a writable snapshot, then either the item is not cached, in which case KVDB is notified and locks it accordingly—or the item is cached, but only for this specific writable snapshot, and must already be locked. By using a separate snapshot with a separate cache for each transaction, cache faults automatically have a one-to-one correspondence with taking read locks on their items.

Chapter 4

Evaluation

Our experimental setup used to evaluate KVDB, KVFS, and other systems is described in Section 4.1. In this chapter, we mainly focus on evaluating the two design goals mentioned in Chapter 3. First, with KVDB's simple and efficient transactional architecture, we show in Section 4.2 that transactions comes with minimal overhead and are highly concurrent. Second, in Section 4.3 we compare KVFS's performance against Ext4, concluding that it is practical to use KVFS. We then evaluate the performance of KVFS's transactional interface in Section 4.4. We measure the effectiveness of our defragmentation algorithm devised for VT-trees in KVFS at the end in Section 4.5.

4.1 Experimental Setup

We conducted experiments on three identically configured machines running Linux Ubuntu 10.04.3 LTS. Each machine includes a single-socket Intel Xeon X5650 (4 cores with one hyperthread on each core, 2.66GHz, 12MB L2 cache). The machines have 64GB of RAM; to test the out-of-RAM performance, we booted them with 4GB each. Each machine has a 146.2GB 15KRPM SAS disk used as the system disk and a 160GB SATA II 2.5in Intel X25-M Solid State Drive (SSD) used to store the out-of-RAM part of the data. We use only a 95GB partition of the SSD to minimize the SSD FTL firmware interference. We measured 10 thread random-read throughput on the Intel X-25M of 15,000 IOps, a random-write throughput of 3,500 IOps, a sequential read throughput of 245MB/sec, and a sequential write throughput of 107MB/sec. We report these numbers because Intel specifications cite 30,000 IOps: after extensive experience and testing with the device, we have not been able to reproduce those vendor-supplied figures. We dropped all the caches before running any benchmark. To avoid swapping, if any, due to memory pressure, we set the Linux SWAPPINESS parameter to zero and monitored `vmstat`'s output to ensure there was no swapping. Since we a observed slight performance variations across machines, despite identical hardware and software configurations, we do not compare results across machines.

4.2 Transactional Performance in KVDB

Transactions in KVDB can be larger than RAM and are highly parallelizable for partitionable workloads. We focus here on evaluating the overhead introduced and the concurrency provided by transactions running in both snapshot isolation and serializability mode in KVDB. As explained in Section 3.3, transactions in KVDB have a private writable snapshot with its own RAM buffer cache for each of the VT-trees within it. The sizes of these caches are configurable. This allows the transactions to be highly concurrent, cache efficient, and also scalable on multi-cores. KVDB supports range locks with deadlock detection for applications to partition their workload.

4.2.1 Concurrent Transactions

Here, we focus on evaluating the overhead and concurrency provided by transactions running only in snapshot isolation mode. In this mode, transactions have minimal overhead and are highly concurrent as this avoids the conflict resolution; however, it can result in lost updates. Applications running in parallel and having no conflicts between them, or running a partitionable workload, can benefit by running as transactions in snapshot-isolation mode. To use transactions in KVDB, applications need to use `txn-begin` to begin a transaction and `txn-commit` to commit all the changes at the end, as described in Section 3.3.1.

Configuration. Each run of the benchmark randomly inserts a data set of 10GB of 64B pairs into a schema consisting of a single VT-tree. We configured KVDB to have 256MB of cache in total. Each transaction gets a part of this 256MB cache for the RAM buffer within its private snapshot. We ran the benchmark with a single thread on the default *main-line* transaction (ML-1T) requiring no `txn-begin` and `txn-commit`. To compare the overhead of `txn-begin` and `txn-commit`, we began a transaction and ran the same single threaded benchmark (1Txn-1T). In both runs, there was only one active transaction and we configured it to use the entire 256MB cache for the RAM buffer in the VT-tree. For measuring the concurrency provided by the transactions, we compared three runs of the same benchmark with varying numbers of transactions and threads. The first run, ML-10T, has ten threads inserting on the *main-line* transaction. The other two have a separate transaction for each thread to insert to: 10 threads (and 10 transactions) for 10Txn-1T and 4 threads for 4Txn-1T, which was chosen to minimize contention on our 4-core test machine. The cache is divided evenly: 25.6MB for each transaction in 10Txn-1T and 64MB for transactions in 4Txn-1T. Transactions are asynchronous in all runs.

Results. As seen in Figure 4.1, the ML-1T and 1Txn-1T runs show that transactional overhead is minimal. A `txn-begin` comes at no cost as creating a snapshot in KVDB is a light-weight operation. ML-1T completed the benchmark in 1,465s and 1Txn-1T in 1,508s. For 1Txn-1T, `txn-commit` took only 1.2s: this involved flushing the RAM buffer into one of the disk lists, moving the on-disk lists into *main-line*, and journaling the meta data. This is because the in case of asynchronous transactions, KVDB avoids double writes for high-insertion throughput workloads that require partial durability, by flushing the RAM buffer

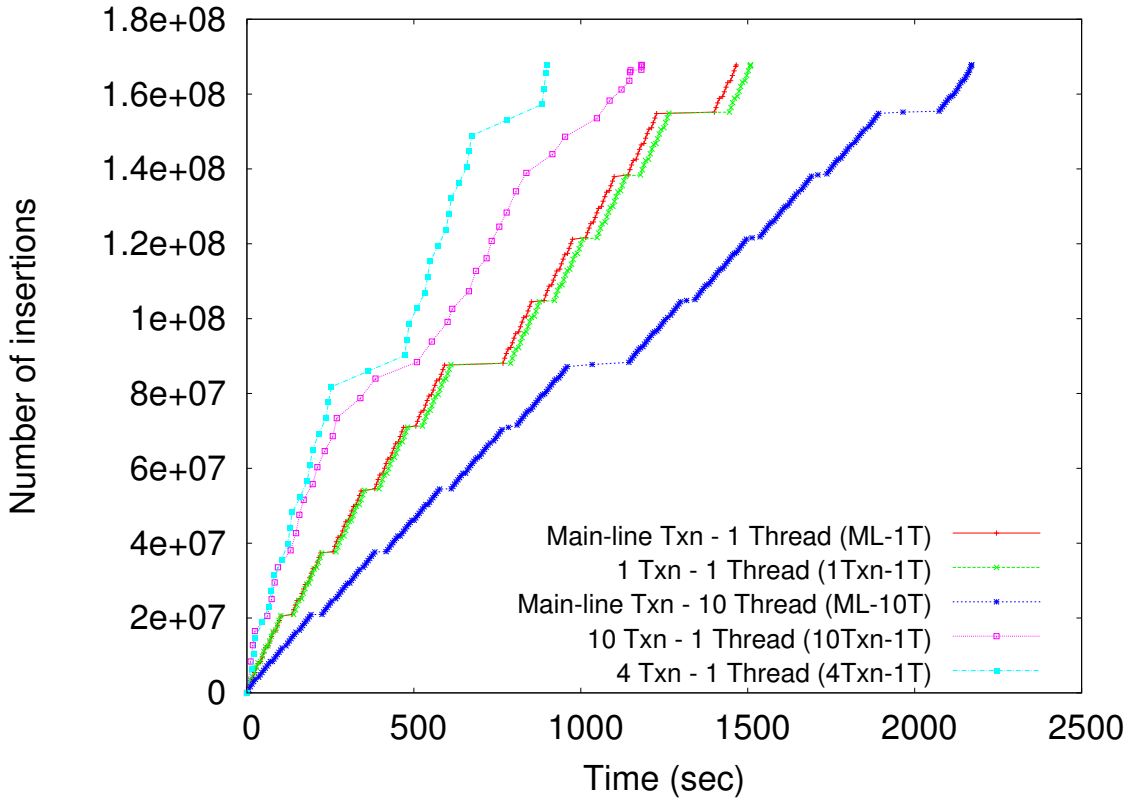


Figure 4.1: Measuring the overhead and the concurrency provided by transactions running in snapshot-isolation mode when the workload is partitionable.

onto disk and journaling only the meta-data. As we see, the 1Txn-1T run had an overhead of only 2–3% which is mainly due to the additional locking required to avoid conflicts.

ML-10T completed in 2,170s and is 33% slower than ML-1T. This is because writes through parallel threads are serialized by the locks within the VT-tree, but can be parallelized across VT-trees. This is what we exploit in the 10Txn-1T and 4Txn-1T runs as each transaction gets its own VT-tree within its private snapshot. 10Txn-1T completed in 1,180s which is 24% faster than 1M-1T, our baseline. Notice that in the 10Txn-1T run, there are ten VT-trees each with a smaller RAM buffer, requiring flushing them to a smaller on-disk list more often. This results in more, but smaller compactions compared to the ML-1T run. This, along with the 3% overhead of each transaction and ten threads contending for four cores, gets us only 24% extra speed. 4Txn-1T completed the task in 899s with 63% better throughput than 1ML-1T. 4Txn-1T is 31% faster than the 10Txn-1T, thanks to reduced contention for cores and also requiring fewer compactions in total. An efficient transactional architecture in KVDB allows us to have only 3% overhead and provides concurrency with 63% better throughput.

4.2.2 Overlapping Transactions

As described in Section 3.3, KVDB supports transactions to be run in the highest level of isolation: serializability. This mode supports repeatable reads and has no lost updates or dirty

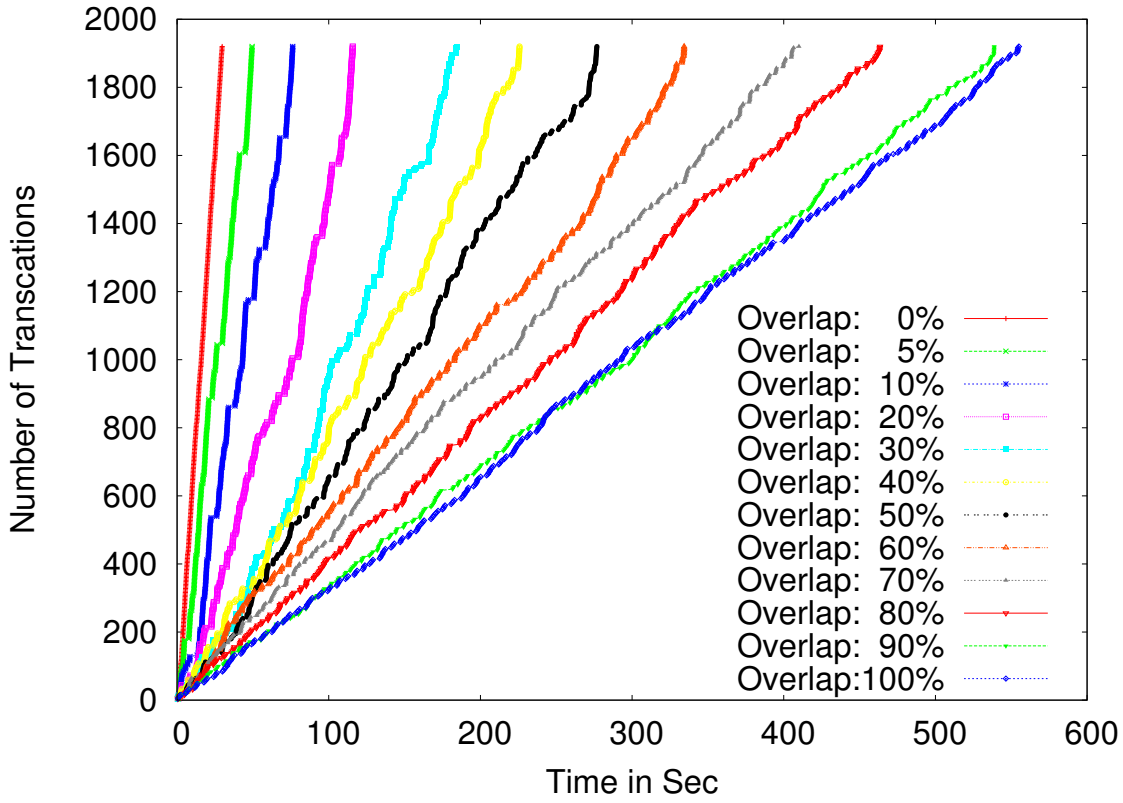


Figure 4.2: Time taken for completing the operations by the transactions running in serializability mode with varying percentage of overlap

reads [14]. KVDB uses strong, strict two-phase locking (SS2PL) to support serializability. We evaluate the transactional overhead when transactions are in serializability mode with a varying percentage of overlap with other transactions.

Configuration. We devised a benchmark that inserts 30GB worth of *dmap* pairs into a schema consisting of a single VT-tree using transactions. Each transaction is of 16MB in size with 1,920 transactions in total. This benchmark runs as four threads, each running an equal number of transactions in parallel. So, at any point in time there are four active transactions. We chose four threads to have less CPU contention between each other, based on the results from our previous experiment. The benchmark allows to specify if there is any overlap between the active transactions and the percentage of overlap. The percentage of overlap determines how many of the transactions being run by each thread have a conflict with one of the active transactions run by the other three threads. So, a 0% overlap means none of the transactions conflict; a 100% overlap means that all of the transactions conflict with at least one of the actively running transaction at the time the conflicting transaction was running. We run the same benchmark 12 times with varying overlap percentage between 0% and 100%. We configured the main-line transaction to have 256MB of cache and each of the active transactions to have 20MB. All the operations on the transaction fit in its cache.

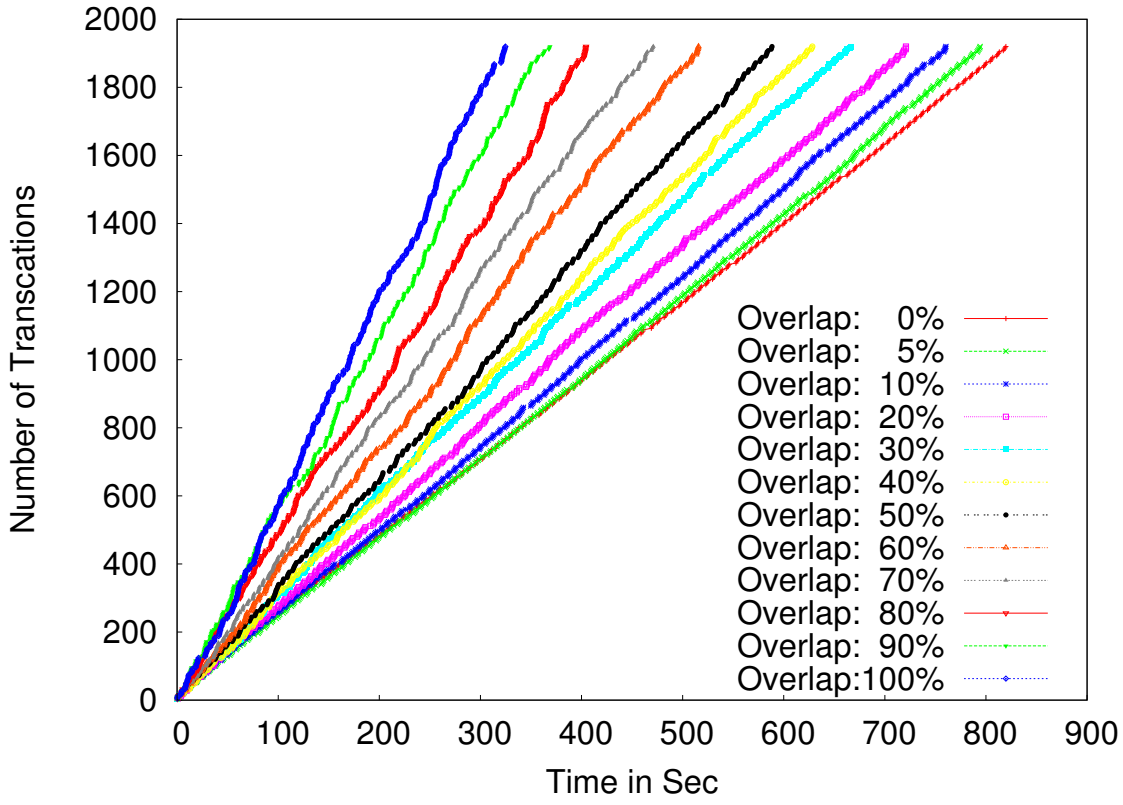


Figure 4.3: Time taken for commiting the transactions running in serializability mode with varying percentage of overlap

Results. Figure 4.2 shows the time taken for inserting 30GB of data (operation time) by 1,920 transactions with a varying percentage of overlap. This does not include the time taken for `txn-begin` and `txn-commit`. Figure 4.3 shows the time taken for committing (commit time) the same 1,920 transactions. The time taken for beginning the transactions was negligible and the same across all runs. The operation time for inserting 30GB of data for the benchmark run with 0% overlap was only 29.7s: see Figure 4.2. This time only considers the total time taken by all the transactions to insert 16MN into its own VT-tree. Since the cache was configured to be 20MB for each of the transaction, this reflects the time taken to insert 16MB of data into its own cache by all the transactions run on four threads. Remember that this is not same as if 1,920 transactions are running in parallel; here, only four transactions at a time could run in parallel. The operation time for 100% overlap is 555.07s. In this run every transaction has a conflicting operation with at least one of the other three actively running transaction. On conflicts, the lower-priority transaction waits for the other conflicting transaction to commit and then resumes. The lower-priority transaction may need to restart its operations in case the conflict causes a deadlock with other transactions. In this run there were two deadlocks resulting in a restart of two transactions. The 100% overlap run is almost the same as if all the transactions were run serially one after the other. All the other runs had their operation time spread linearly between that of the 0% and 100% overlap runs without any anomalies as expected.

The commit time shown in Figure 4.3 is interesting. The total time spent in `txn-commit` by all the transactions in the 0% overlap run is 819.74s, whereas it was 324.69s for the 100% overlap run. *txn-commit* in KVDB requires evicting the pairs in the private cache of the transaction into the corresponding VT-tree in the main-line schema and then journaling the same for durability. Currently, the time to evict the pairs from private cache into the main-line takes at least as much time as it took for inserting those into the private cache of the transaction. This is because we copy all the pairs between the private cache and the main-line instead of moving them. So, the per-transaction `txn-commit` time is actually more than the per-transaction operation time. This is only true if the transaction is small and everything fits in its private cache. Additionally, KVDB does not support group commit yet and all the commits are serialized. So, in the 0% overlap run, every committing transaction has to wait for at least one other transaction to commit except the one that committed first. This forces the commit time of each transactions to include the commit time of at least one other transaction, making a group-commit feature necessary. However, in the case of the 100% overlap, the commit time of some of the transactions are included in the operation time of the other transactions instead. This is because, on a conflict, which happens very frequently in the 100% overlap run, the lower-priority transaction waits for the conflicting higher-priority transaction to commit before resuming. This makes the operation time of the lower-priority transaction to include the commit time of the conflicting higher-priority transaction. This is an expected behavior for the 100% overlap run. Unless a group commit is implemented, and also we move the pairs efficiently between the transaction's private cache and the main-line on commit, the total time taken by all the transactions in the 0% overlap run, the 100% overlap run, or anything in between are be almost the same. This is in fact the case: the 0% overlap experiments completed the entire benchmark in 846.13s and the 100% overlap run completed it in 852.07s. We may not have thousands of transactions running in parallel and all of which have conflicts with others; still, even then this experiment shows the importance of supporting group-commit. Since KVDB's transactional architecture is designed for supporting larger-than-RAM transactions, where the amount of data in cache is small compared to the total data, the commit time is going to be negligible as shown in our previous experiment in Section 4.2.1.

4.3 File System Performance

KVFS uses FUSE to support POSIX operations. Using FUSE requires two additional context switches and buffer copies than running on a native file system. This results in around 2× overhead compared to the native file system performance [28]. However, serial reads on FUSE are comparable to and even some times better than native file systems. This is due to caching and read-ahead performed at both the FUSE kernel component and the lower native file system [28]. The FUSE kernel module caches read pages, but writes are immediately sent to the FUSE server running in user space. We have designed a simple write-back caching but its implementation as of this writing is not complete enough to run bigger workloads. To exclude FUSE overhead, we compare KVFS's performance with FUSE-Ext4 a pass-through FUSE mounted on Ext4. We also evaluated Ext4 to measure FUSE overhead by comparing it against FUSE-Ext4. We use Filebench [10] for evaluating these systems. We first evaluate

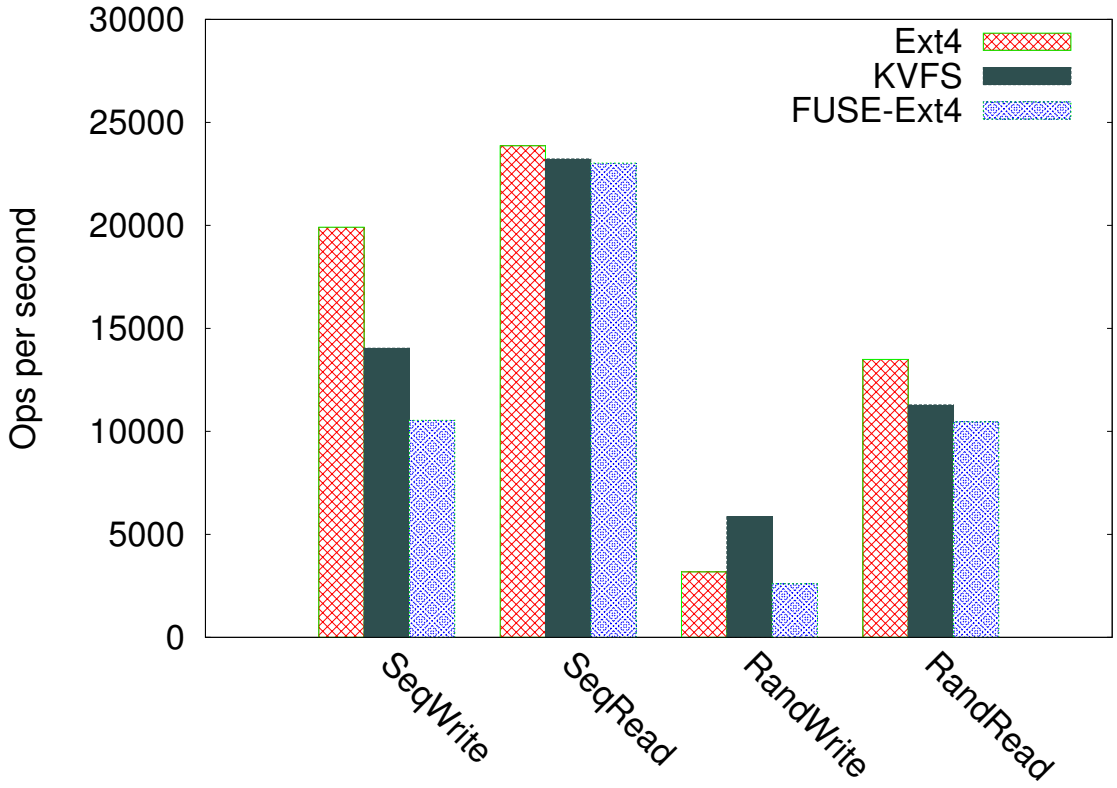


Figure 4.4: Filebench results for micro-benchmark workloads on SSD

these systems by running micro-benchmarks like serial reads, serial writes, random reads, and random writes in Section 4.3.1. We then use some real-world like workloads provided by Filebench such as a fileserver, videoserver, varmail, and webserver to benchmark these systems in Section 4.3.2.

4.3.1 File System Micro-benchmarking

Configuration. KVFS creates an *fs-schema* consisting of three VT-trees in KVDB as described in Section 3.1: *nmap*, *imap*, and *dmap*. We configured *nmap*, *imap*, and *dmap* to have RAM buffers of sizes 6MB, 12MB, and 512MB, respectively. Using larger RAM buffer can improve performance, but at the same time we want enough memory left to accommodate secondary indexes and Quotient Filters to stay in RAM. We set the *stitching-threshold* to 32KB for all the runs. We describe the reason for choosing this stitching threshold in Section 4.5. We use Filebench's randomread, randomwrite, singlestreamread, and singlestreamwrite micro workloads. All the I/Os are done at 4KB size unless otherwise mentioned. Filebench's randomread workload reads a 30GB single file randomly. The randomwrite workload performs random writes on a pre-allocated 30GB file. The singlestreamread workload serially reads a pre-created 30GB file with I/O size of 16KB. The singlestreamwrite workload sequentially starts writing to a file at offset zero.

	SeqWrite	SeqRead	RandWrite	RandRead
Ext4	19,914	23,866	3,179	13,491
KVFS	14,035	23,227	5,857	11,286
FUSE-Ext4	10,537	23,025	2,603	10,462

Table 4.1: Results of Filebench micro-benchmarks in ops/sec

Results. As seen in Figure 4.4 and Table 4.1, both the FUSE variants including KVFS have similar performance for serial reads when compared to Ext4. Since in the serial read workload, the file is first written sequentially, VT-tree does not need to do any sorting; stitching preserves the block order, allowing fast subsequent serial reads. The stitching optimization in VT-tree avoids merging during compaction for serial writes, allowing its performance to be better than FUSE-Ext4 even though these are equipped with extents and delayed allocation optimizations. VT-tree caches the dirty data and writes out the data serially in a sorted order when the RAM buffer becomes full. For this reason, KVFS does not need to support delayed allocations. KVFS also does not need to support extents as frequent compactions in VT-tree reduce fragmentation over the lifetime of the file if the stitching threshold is set appropriately. In KVFS, a random 4KB write becomes a new key-value pair insertion into KVDB, which acts as an update during one of the merges within VT-tree. Since random writes into KVFS become a serial write of a list, KVFS's random write performance is better than other file systems including Ext4. Random writes in KVFS are 84% faster than Ext4 and 125% faster than FUSE-Ext4. Inserting lots of pairs result in frequent merges; and since the data is random, stitching is not of much help, making KVFS's random-write performance not the same as KVFS's serial write performance, but it still significantly outperforms other file systems. With the use of Quotient Filters [4] and secondary indexes, KVDB achieves SSD's random-read throughput for random point queries, making random reads in KVFS comparable to FUSE-Ext4 and Ext4. For sequential writes, FUSE-Ext4 is almost twice as slow and around 22% slower for random writes than Ext4. Providing write-back cache helps here, especially when the workload is CPU bound, by reducing context switches. With the write-back caching at FUSE kernel, KVFS's random write performance can improve even further.

4.3.2 File System Macro-Benchmarking

Configuration. The Fileserver workload performs a sequence of creates, deletes, appends, reads, writes, and stat operations on a directory tree. We configured the mean size of the file to 100KB, mean append size to 16KB, directory width to 20, and number of files to 100,000. Filebench pre-allocates 80% of the files and randomly selects a file for each of the above operations. We ran the benchmark for 10 minutes each with 10 threads and I/O size of 4KB for all file systems on SSD.

The Videoserver workload has two file sets: actively serving videos from one and the other set contains videos that are currently inactive. In this workload one thread writes new videos to replace no longer viewed videos in the passive set. Meanwhile 48 threads are serving videos from the active video set. The Videoserver workload has a write I/O size of 1MB and Read I/O size of 256KB. The active video set contains six videos and the passive

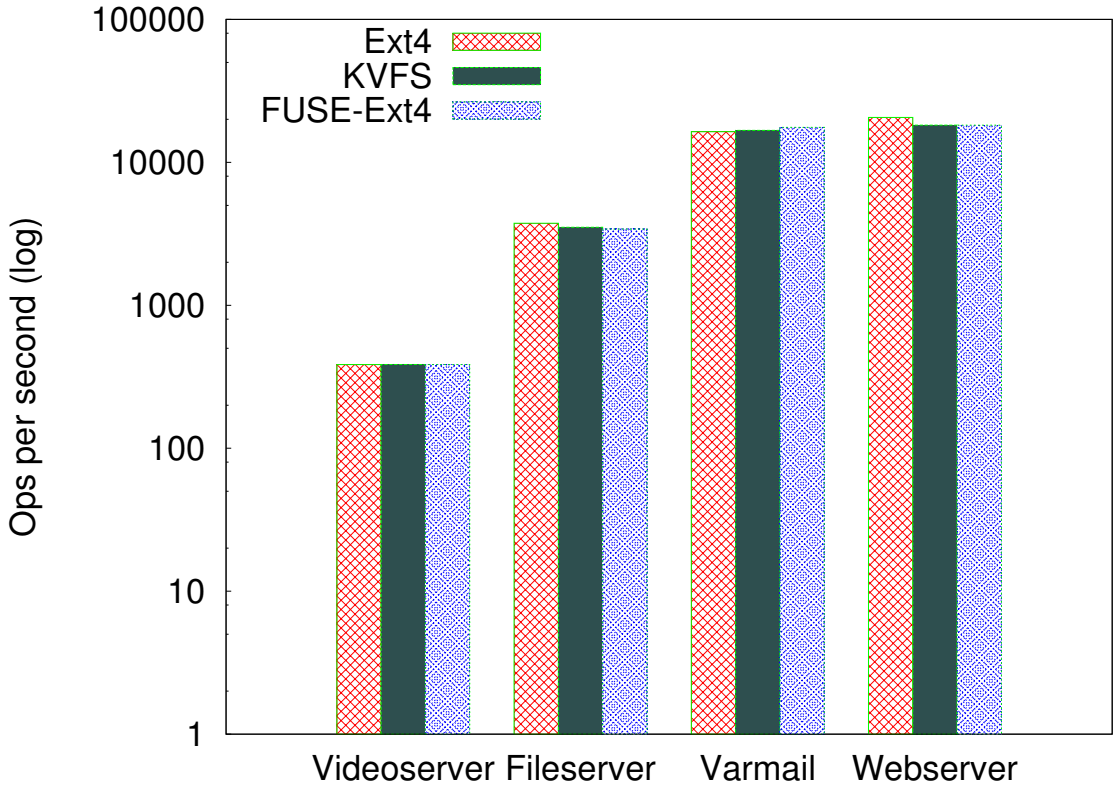


Figure 4.5: Filebench macro-benchmark workloads result on SSD. Workloads include videoserver, fileserver, varmail, and webserver.

	FileServer	**VideoServer**	**VarMail**	**WebServer**
Ext4	3,750.2	384.8	16,395.7	20,626.1
KVFS	3,515.6	385.3	16,737.9	18,212.4
FUSE-Ext4	3,416.5	384.8	17,576.0	18,215.8

Table 4.2: Results of Filebench macro-benchmark workloads in ops/sec

set contains 26 videos.

The Varmail workload emulates I/O activity of a simple mail server storing emails in separate files. It consists of 16 threads doing create-append-sync, read-append-sync, read, and delete operations in a single directory. The I/O size used in this workload is 1MB.

The Webserver workload produces a sequence of open-read-close with 100 threads on multiple files in a directory tree, plus a single log-file append thread of mean append size 16KB for every 10 read threads.

Results. As seen in Figure 4.5 and Table 4.2, for all the macro-benchmark workloads, KVFS performs comparably or superior to FUSE-Ext4 and Ext4.

The Fileserver workload has a good mix of metadata operations and large sequential and random read or writes. By looking at ops/sec for each operation, we noticed that KVFS performs comparable or superior to FUSE-Ext4 and Ext4 for metadata operations like open,

create, stat, and delete operations. These metadata operations are similar to random database workloads consisting of small tuples and are ideal for the VT-tree. The file server workload also includes appending 16KB to a random file and randomly reading the whole file. This append workload is similar to random writes. KVFS again performs better than FUSE-Ext4 and Ext4 here. But, for randomly reading the whole file, KVFS has around 5.5% lower throughput than Ext4, but 3% better throughput than FUSE-Ext4. Compactions of on-disk lists in VT-tree brings these randomly appended data together over the lifetime of the file. Frequent insertions of the data in the file server workload ensures that the compactions are triggered often, improving the performance of the subsequent sequential reads of a file. The frequency of compaction is determined by the insertion rate and can also be triggered periodically which to improve KVFS's performance for whole file reads even further.

For the videoserver workload, all the file systems perform equally well as FUSE overhead is negligible for requests with large I/O size [28].

In the varmail workload, FUSE-Ext4 has a better performance than Ext4 for read-append-sync cycle. FUSE file systems are noted to have slightly better performance for sequential reads because of the double readaheads and double caching happening at the FUSE kernel and lower file system layer [28]. For every other operations, all three file systems performs almost the same, as the I/O size is large as in videoserver workload.

FUSE-Ext4 and KVFS are 11.6% slower than Ext4 for the webserver workload. The Webserver workload is mostly read oriented, and files are not large enough to help FUSE's user space file systems with additional readaheads. This makes the read cycle of the workload performing better for Ext4.

In sum, for macro workloads, FUSE shows negligible overhead with the highest of 11.6% for Webserver; sometimes it performs better than the in-kernel file system for workloads like varmail. Additionally, KVFS's performance is superior or comparable to FUSE-Ext4 for almost all workloads.

4.4 Real Workload and KVFS's Transactional Performance

Here, we use Samba and Linux kernel compilation, and other related operations to compare the performance of KVFS with Ext4 and FUSE-Ext4. For KVFS, we ran the same set of operations with and without transactions to measure the overhead caused by transactions. In case of transactions, we ran them in both snapshot isolation (KVFS-TXN-SI) and serializability (KVFS-TXN-SER) mode. As described in Section 3.3.3, applications need to use the APIs `txn-begin`, `txn-commit`, and `txn-abort`—provided by our user library—to invoke transactional operations on KVFS.

In this benchmark we include the following operations in this order:

1. txn-start: This operation begins a transaction and is included only for KVFS when it is running its operations in a transaction.
2. untar: Untar the downloaded tar version of the source.
3. removetar: Remove the downloaded tar version.
4. make: Run `make` inside the source directory.

	Pre-Compile			Post-Compile	
	TarSize (MB)	**UntarSize (MB)**	**Number of Files**	**Size (MB)**	**Number of Files**
Samba-3.3.6	25	80	4,343	189	5,151
Linux-3.0.1	74	494	39,048	739	45,737
Linux-3.1.1	74	498	39,371	746	46,103
Linux-3.2.1	75	505	39,964	753	46,945
Linux-3.3.1	76	511	40,449	763	47,502

Table 4.3: Size and number of files in Samba and Linux source directories before and after compilation.

5. findall: Run `/bin/find` on the source directory. This is basically a meta-data operation that lists the name of all files and directories recursively under the source directory.

6. readall: This scans through each file in the source directory recursively, and reads its content and write to `/dev/null`.

7. txn-commit: This operation commits a transaction and is included only for KVFS when it is running its operations in a transaction.

We excluded the time taken to configure and install the compiled versions as the target file systems were not mounted as root file system.

4.4.1 Single Transaction

Configuration. In this benchmark, the aforementioned operations are first run on Samba version 3.3.6 and then on the Linux kernel version 3.0.1. We measured times separately for each run. Table 4.3 shows the size of the source directories and the number of files in the source directory before and after compilation. The operations findall and readall were performed after compilation, so they have more files to list and read. We used the default configuration for both Samba and Linux. We ran `make` with five concurrent threads.

Results. As seen in Figure 4.6, the total time taken by all the operations for Samba is 53.55s and 69.35s for Ext4 and FUSE-Ext4, respectively. Around 93% of the total time is spent in compilation itself. KVFS with no transaction completes the benchmark in 63.29s, which is around 9.5% faster than FUSE-Ext4, even after having an equal amount of FUSE overhead. KVFS is only 15% slower than the in-kernel Ext4. KVFS performs better than FUSE-Ext4 on all operations except the readall. FUSE-ext4 took only 3.28s for the readall operation whereas KVFS completed it in 4.65s. KVFS stores the data in *dmap* format. *dmap* is 4KB in size to make it aligned with the page size of the underlying file system. In *dmap*, the key takes up 16 bytes and 1 byte is used for the delete flag. This leaves only 4,079 bytes to store the actual data. So, KVFS may need to do more lookups to read the given amount of data when

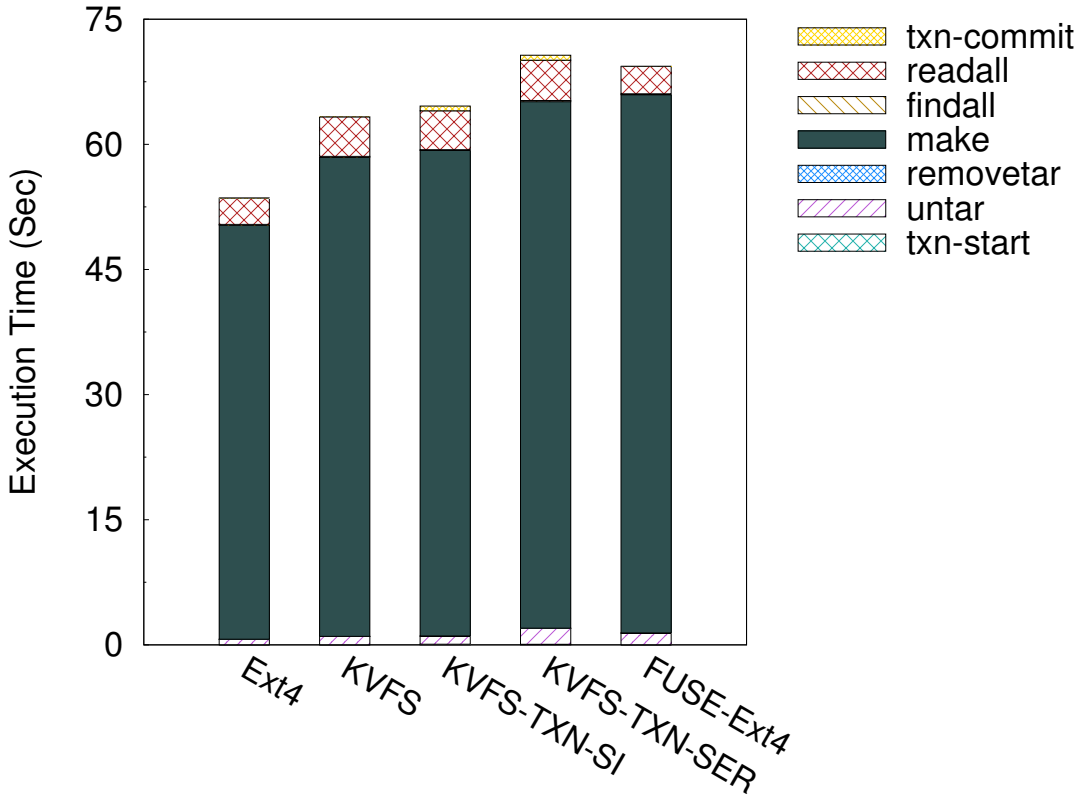


Figure 4.6: Samba compilation results

compared to Ext4 or FUSE-Ext4. KVFS, when running these operations as part of a transaction in both snapshot isolation and serializability mode, completed this benchmark in 64.58s and 70.68s, respectively. The transaction in snapshot-isolation mode has only `txn-start` and `txn-commit` operations in addition to KVFS run without transaction. `txn-start` is negligible and `txn-commit` took only 0.57s to complete. The transaction in serializability mode completed the benchmark in 70.68s with the overhead of 10.45% when compared to KVFS run with no transactions. As serializability needs to maintain strong isolation guarantees, these overheads mainly come from the range-lock tree and deadlock-avoidance checks.

Figure 4.7 shows the result for all file systems for operations on the Linux kernel source. The results are similar to that of Samba run except that each operation took more time as the code base of Linux kernel is larger. Ext4 completed the benchmark in 220.05s, FUSE-ext4 in 250s, and KVFS without transaction in 240.18s. In this run KVFS was 4% better than FUSE-Ext4 and only 8.5% slower than the Ext4. Here, compilation took around 52% of the total time and readall took 40% of the time. After compilation, there were 45,737 files with a total size of 739MB. Transactions in snapshot isolation have almost zero overhead with `txn-commit` taking only 0.28s. Transactions in serializability mode show a similar overhead of 10% in this benchmark as well when compared to KVFS run without any transaction.

In these runs, KVFS is 4–9.5% faster than FUSE-Ext4 and only 9.5–15% slower than the in-kernel Ext4 file system, showing that its practicality to be used in desktops and servers with the additional benefits of transactions and other features. Transactions in KVFS with

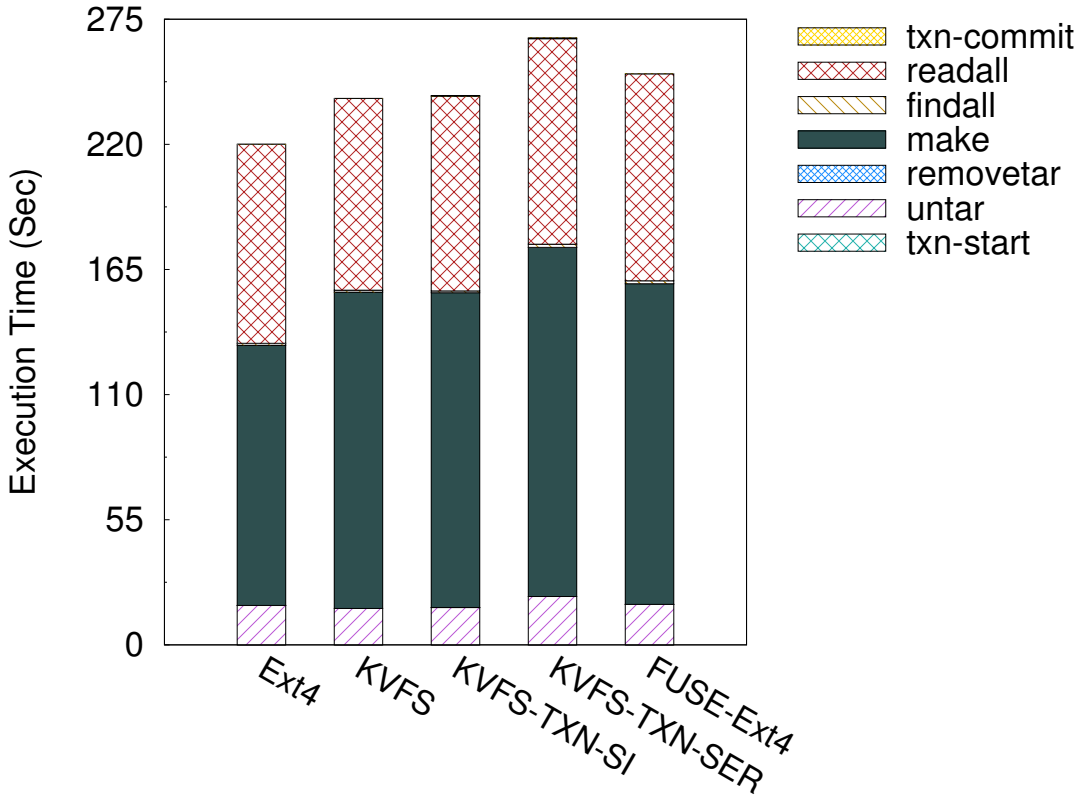


Figure 4.7: Linux kernel compilation results

weaker isolation guarantees come almost for free, and offer stronger serializability isolation for overlapping transactions with only 10% overhead.

4.4.2 Parallel Transactions

Configuration. In this experiment, we ran the same benchmark with all the operations mentioned in Section 4.4.1 for the four Linux kernel versions 3.0.1, 3.1.1, 3.2.1, and 3.3.1—only now all of them were run in parallel. In each of the benchmarks with a particular Linux kernel version, we ran `make` internally with five threads as before. For KVFS, we configured the mainline cache to be 256MB in size and any used transaction had 64MB configured for their private cache.

Results. Figure 4.8 shows the results for this experiment. Ext4 took 220.5s to run these operations on a single Linux kernel source; it completed this parallel benchmark on four Linux kernel versions in 453.76s. FUSE-Ext4 completed the same in 571.10s and KVFS took only 514.45s. So, KVFS performed 11% better than FUSE-Ext4 and 13.4% slower than the Ext4. For running transactions in KVFS, the snapshot-isolation mode is best suited here as these operations do not include any conflicting requests. Remember that these file systems are not mounted as root file systems. We ran each of these four benchmarks in separate transactions in snapshot-isolation mode. All transactions work in their own snapshot

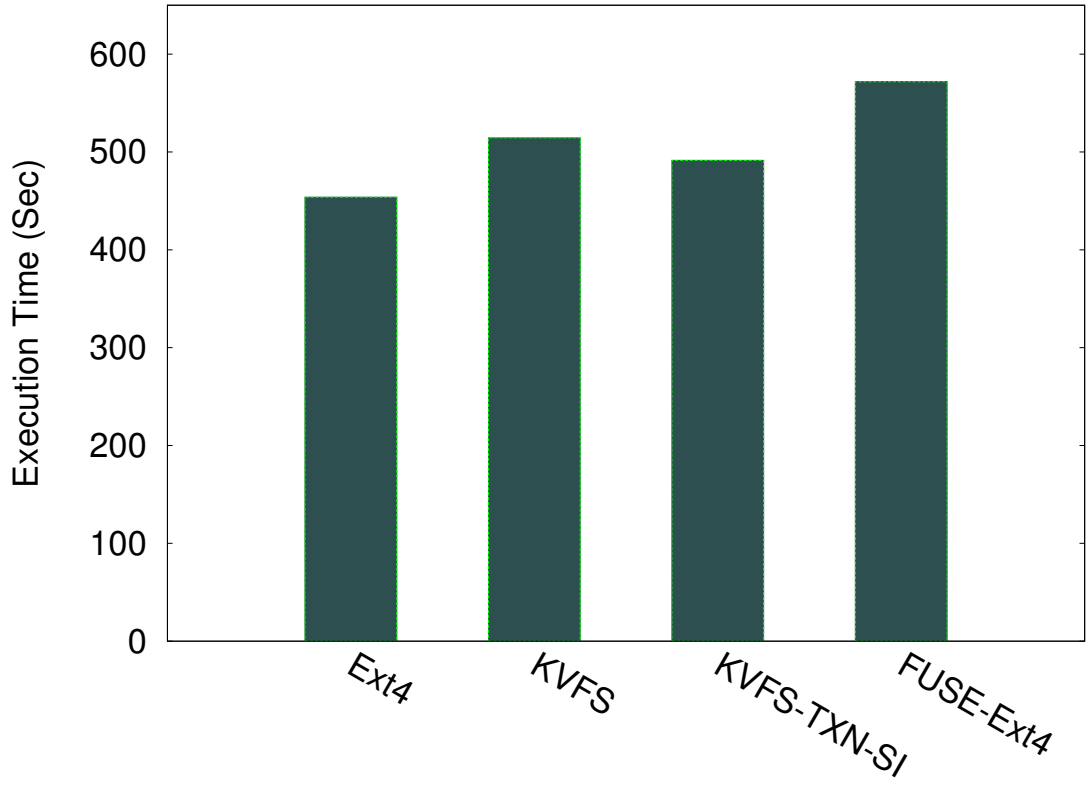


Figure 4.8: Parallel Linux kernel compilation results

directory without stepping over each other and they commit their changes at the end. These snapshots have their own VT-tree with a smaller private cache, and is highly concurrent. This run, with the transactions, completed in 491.10s, 4.5% faster than without transactions. A smaller private cache size means that a larger number of lists gets generated and merges become more frequent. Increasing the cache size improves the performance even further as long as it does not swap out other entities like secondary indexes and Quotient Filters from RAM.

4.5 Defragmentation in KVFS

Log structured systems face the issue of fragmentation; KVFS, which uses VT-trees, is no exception. We have devised a defragmentation algorithm described in Section 3.2.1. We first measure the effectiveness of our defragmentation algorithm using a synthetically generated workload. We then analyze the characteristics of fragmentation in KVFS and its relation with the stitching threshold supported by the VT-tree.

Evaluating the Defragmentation Algorithm

Configuration. We devised a benchmark that inserts around 38GB of *dmap* pairs representing the file data blocks into KVDB. Given a stitching threshold, the benchmark repeatedly

generates a sequence of data of length chosen randomly between 4KB and the *stitching-threshold* part of different a file each time, until the specified amount of data (38GB) is inserted. We ran this benchmark with a stitching threshold of 64KB. We configured KVDB with 512MB cache. Inserting 40GB into VT-tree creates 11 lists.

Stitching threshold	Range query before defrag (sec)	Defragmentation (sec)	Range query after defrag (sec)	Used zones	Reclaimable zones
64KB	377.71	284.14	274.82	11,264	6,270

Table 4.4: Results before and after defragmentation

Results. After inserting all the data, the total number of used zones were 11,264 as shown in Table 4.4. The total number of zones required for holding 38GB worth of *dmap* pairs is only 4,864. After scanning through all the lists, our algorithm determined that it can reclaim 6,270 zones, which is 55% of the total used zones. The rest of the 130 zones are either used by the lists at the top or by the disk-backed secondary indexes and Quotient Filters. Before running our defragmentation algorithm, we ran a range query in our KVDB, which reads all the *dmap* pairs in KVDB in sorted order. As the data is sorted only within each list and not across the lists, this operation needs to do a lot more work than sequentially reading all the lists. This operation took around 377.7s reading at 103MB/s. We measured this to compare the performance of the same operation after defragmentation. As our defragmentation considers the sequentiality of existing data before moving it around, it should ideally improve the performance of the range operation. Defragmentation required moving around 15GB of data from the candidate zones to the rest of the zones. Our defragmentation algorithm reclaimed all the 6,270 zones in only 284.14s with a rate of 54MB/s. Our algorithm reads the data from the zone sequentially, but the write to the zone, even though it is sequential, may not be contiguous, resulting in an acceptable throughput of 54MB/s considering that the data needs to read from and written to different locations. The range query operation after the defragmentation took only 274.8s with an improvement of 27.3%.

Analysis of Fragmentation in KVFS

We used a Filebench workload consisting of appends to a random file picked from a set of 10,000 files with mean append size of 16KB and a random write to a pre-allocated 10GB file in each cycle. The benchmark runs for 30mins and inserts around 34GB of data. To analyze the fragmentation in KVFS, we ran the same benchmark with different stitching thresholds each time.

Figure 4.9 shows the percentage of under-utilized zones after each run of this benchmark for stitching thresholds of 32KB, 64KB, 128KB, 256KB, and 512KB. As we see, the fragmentation decreases as the stitching threshold increases. This is because larger stitching thresholds result in fewer number of stitched regions and most of the data is compacted and thereby gets copied to the zones in the new list.

Figure 4.10 shows the distribution of zones and its percentage of usage for different stitching thresholds. As we see, the fragmented zones are concentrated around 40% or 70% usage

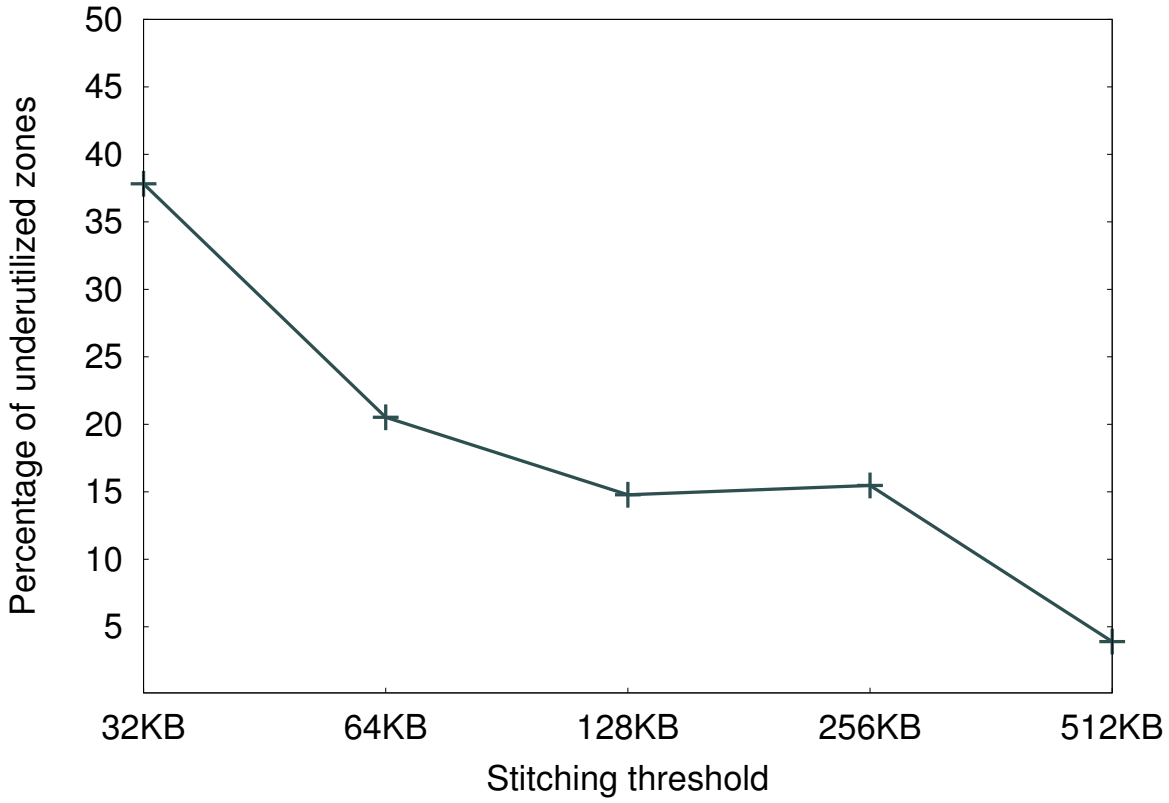


Figure 4.9: Percentage of under-utilized zones ($< 75\%$) with different stitching thresholds after inserting 34GB of data in KVFS

for smaller stitching thresholds. The 32KB stitching threshold also shows a large number of highly fragmented zones with around 10% utilization. Runs with larger stitching thresholds (256KB and 512KB) are in the extreme: either they have a large number of fewer than 10% utilized zones, or a large number of less fragmented zones. This is because if the zone contains any stitched region, their number is less or they are part of the compacted lists with no fragmentation for larger stitching thresholds.

The average file size in a general-purpose file system is around 32–64KB [1]. Having a large stitching threshold defeats the purpose of stitching in order to save the copies for an already sorted data of files smaller than the stitching threshold. At the same time, if the stitching threshold is small, sequential reads on large files like video and audio files might have to read from many stitched regions across many zones. This can only happen if the file was not written once and instead appended to at different point in time, which is often not true for video and audio files. So, having a relatively small stitching threshold of, say, 32KB with periodic defragmentation appears the best strategy for KVFS.

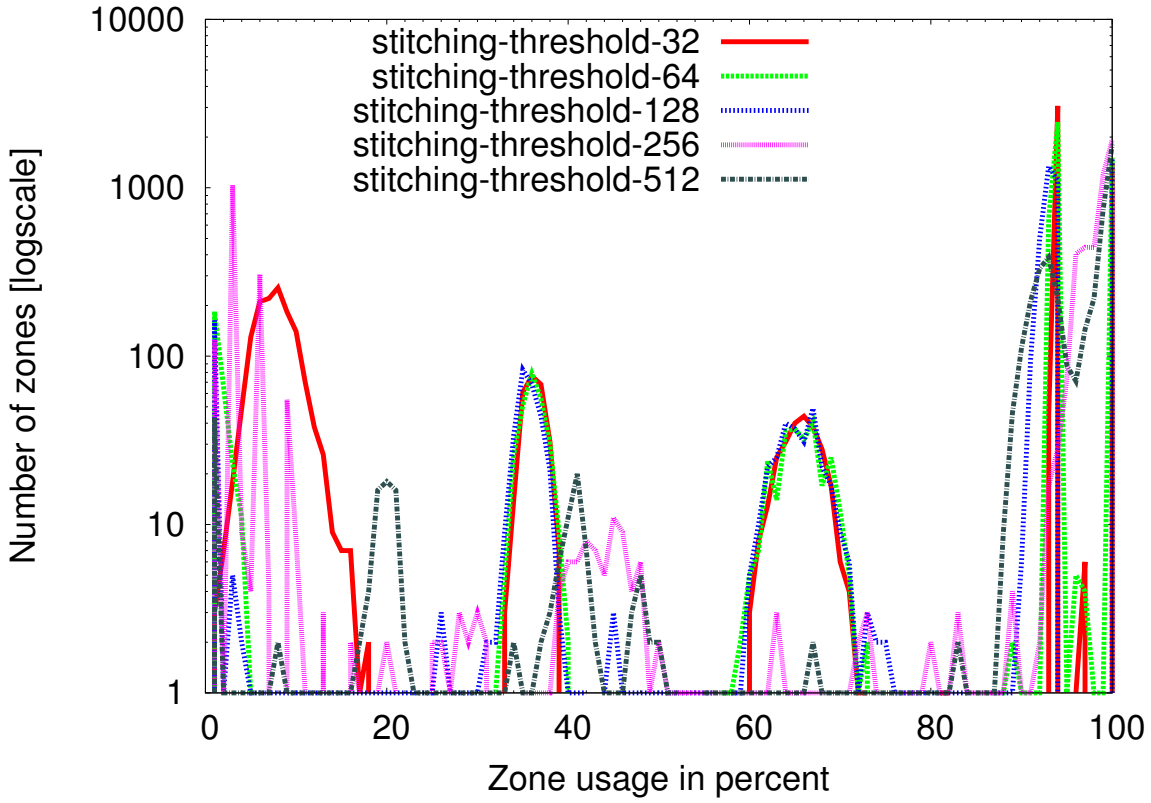


Figure 4.10: Zone usage distribution for runs with different stitching thresholds

4.6 Evaluation Summary

An efficient transactional architecture in KVDB allows us to have only 3% overhead and provides concurrency with 63% better throughput by using snapshots for partitionable workload. For small and parallel running transactions requiring strong isolation guarantees, our experiment shows that supporting a group-commit feature in KVDB is necessary to improve performance further. KVFS shows comparable performance with FUSE-Ext4 and Ext4 for all micro-benchmarks except random writes where KVFS is 84% faster than Ext4 and 125% faster than FUSE-Ext4. For KVFS, the overhead of using FUSE is mainly visible in sequential writes with an overhead of around 30% when compared to Ext4. Our ongoing write-back cache implementation should help here and further improve KVFS's random write performance. As we saw in the macro-benchmarks like fileserver, videoserver, varmail, and webserver workloads, KVFS is capable of handling small file operations well, and also handles large writes and sequential reads of whole files comparably to FUSE-Ext4 and Ext4. In our Samba and Linux kernel compilation experiments, KVFS consistently performs better than FUSE-Ext4. Using transactions in KVFS for running many Linux kernel compilations in parallel actually improves the performance as opposed to adding an overhead. These parallel Linux kernel compilations are an example of applications that can run in parallel with no conflicts and can benefit from KVFS's transactional architecture. We also show that fragmentation in KVFS is higher when used with a smaller stitching threshold. However, it can

be addressed efficiently with the defragmentation algorithm we have devised specifically for the VT-trees used within KVFS.

Chapter 5

Related Work

In this chapter we discuss work related to KVDB and KVFS. We first discuss other kinds of write-optimized systems and databases that use LSM-trees and its variants. We then discuss additional features supported by KVFS when compared to some of the existing log-structured file systems. Finally we discuss some of the existing transactional file systems and versioning file systems.

Write-optimized databases. GTSSL [40] uses an LSM-tree variant [23], but with a multi-tier extension. GTSSL also supports ACID transactions. However, GTSSL does not support snapshots and transactions larger than RAM, which are critical for a transactional file system [39]. It also lacks the sequential optimization that the VT-tree has. VT-trees can be easily extended to support multi-tier storage and is part of our future work. Anvil [19] provides a modular framework for applications to have different data layouts. Anvil does not provide ACID transactions; Anvil's future work includes transactions, support for concurrent LSM-trees and larger-than-RAM transactions. BDB Java Edition (BDBJE) [9] is a log-based, no-overwrite transactional storage engine. In BDBJE, each key-value pair is stored in a separate leaf node of a B-Tree. It uses a log to store the dirty key-value pair and behaves like other log-structured systems. Our evaluation shows that BDBJE performs well for out-of-RAM sequential insertions. However, its performance drops considerably for random insertions of small key-value pairs, and requires 5$\times$ more memory for its Java heap to keep up with its competitors. BDBJE's authors are aware of this [25].

B^+-trees keep all records in the leaf nodes and they chain nodes at the leaf level or at each level using *sibling* pointers. These are used to improve range-scan performance. Using sibling pointers causes write operation to ripple forward, backward, and back among the leaf nodes. Goetz's write-optimized B-trees [13] do not maintain pointers to siblings, which are required for chaining leaf nodes. Instead of these sibling pointers, write-optimized B-trees use fence keys, thereby avoiding the expense of updating the siblings during a write operation. This affects the scan performance. However, write-optimized B-trees also support in-place insertions, which result in random writes during updates. Write-optimized B-trees are not scalable as they require all the index nodes to be in cache. Our VT-trees do not have any of these limitations.

Rose [34] uses a more traditional LSM with a fixed number of levels. Rose is designed only for database workloads and it allows fast replication by compressing the column. Rose

is orthogonal to our work.

Log-structured file systems. Log-structured file systems (LFS) [30] write the changes to a circular log and do not overwrite the file data on disk. LFS's *threading* concept is reminiscent of VT-tree's sequential optimization, called *stitching*. LFSs require that all keys in their data structure reside in RAM to avoid disk I/O boundedness [45]. This is not an issue for file systems that deal with large 4KB pages, but is an issue for smaller tuples, which KVFS was designed to transparently support as well. KVFS uses a similar segment-cleaning method as used in LFS, but our algorithm, used in KVFS differs in how it chooses the candidate segments for cleaning. LFS uses the age of the data in the segment, whereas KVFS uses the sequentiality of the existing data to determine the benefit of cleaning a particular segment.

Snapshots are an implicit feature of LFS. Other log-structured file systems like WAFL [15] and ZFS [42] use copy-on-write to support snapshots and to preserve data atomicity. KVFS also supports snapshots. It uses snapshots to support highly concurrent application-level transactions. Unlike WAFL and ZFS, we allow the snapshots to be modified by the transaction that owns it, and we keep the changes isolated from other transactions. We merge the changes to the active file system when the transaction commits. We avoid conflicts using pessimistic locking in KVFS. VT-tree, used internally within KVFS, behaves like LFS for file-system workloads and like an LSM for database workloads, preforming equally well on both types of workloads.

Transactional file systems. Amino [48] uses `ptrace` to interpose file system operations in order to provide transactions. Using ptrace introduced high overheads, and the ptrace monitor has to run in privileged mode. Valor [39] adds an in-kernel logging subsystem for write-ordering and an in-kernel locking to provide a transactional file interface. Valor supports long lived transactions and transactions larger than RAM. QuickSilver [33] is one of the early systems to incorporate transactions into the OS, but each component had to support two-phase commit and rollback for a complete transactional support. TxOS [27] detects conflicts and data versioning at the virtual file system layer to behave like a transactional file system. TxOS does not support transactions larger than RAM. TxF [46] uses the exiting transactional manager in NTFS to support transactions. It requires each component to implement its commit and rollback methods similar to QuickSilver. TxF performs writes twice and must gather undo images using reads when data is overwritten, as it does not operate on a log-structured file system and instead it uses a traditional transactional architecture. These transactional file system interfaces either add some high overhead, require complex kernel changes, or require a complete redesign. KVFS supports long lived and larger-than-RAM transactions, requires no kernel modifications, introduces negligible overhead, and is simple.

File system meta-data indexing. FFS's optimizations for directory lookups and insertions are simple but do not scale as well as B-Trees. XFS [43], ReiserFS [29], Ext3, and Btrfs [24] use better B+Trees to index path names. Perspective [31] offers a user-friendly distributed file system based on MySQL. BeFS [11] and HFaD [35] focus on multiple indexes for the same file. Both use traditional B+Trees. Previous projects showed how inefficient randomly written B+Trees can be, even for SSD devices [37, 40]. Spyglass [18] partitions names-

pace metadata into smaller KD-trees [5] that fit in RAM; it writes updated KD trees in new locations and links them to previous versions to increase update throughput. The authors admittedly avoid tree compaction [18], which is almost always required [8]; they also admittedly do not measure its asymptotic performance or compare against other write-optimized indexes. VT-tree, however, scales well when indexes exceed RAM sizes and it works well for any type of hard-to-partition indexed data (e.g., dedup, global search).

Versioning file systems. Versioning file systems provide file-system level as well as per-file versioning capabilities. Ext3cow [26] provides time-shifting versioning at the file level, allowing users to access these versions based on time. Versionfs [21] implements versioning at the stackable file system layer and adds versioning functionality transparently on top of any other file system. The Elephant file system [32] allows user-specified retention policies to manage versioned files. Cedar [12] provides file versioning by maintaining different copies of data with no sharing between versions. WAFL and ZFS uses snapshots to facilitate backup and recovery. KVFS also aims to provide file-system–level snapshots for the same purpose but does not support per-file versioning. It currently does not provide a user interface for accessing the snapshots other than via transactions. In KVFS, snapshots are private to a transaction, but it is easy to support shared user snapshots.

Chapter 6

Conclusions

Our main goal was to build a simple transactional storage system that supports both file system and database workloads efficiently. Our co-developed VT-tree data structure performs efficiently for any mix of highly sequential workloads (e.g., file-based data access), as well as highly random workloads (e.g., structured data access and databases). Our defragmentation algorithm efficiently solves the issue of fragmentation in VT-trees caused by stitching, which is an important feature to enable efficient sequential and file-system workloads. We designed an efficient transactional architecture with only 2–3% overhead on single asynchronous transactions, but improves highly concurrent performance by as much as 63%. Transactions requiring strong isolation guarantees incurs only 10% overhead. Our KVFS performance is comparable to Ext4 and superior to FUSE-Ext4 for most of the workloads, showing its practicality to be used in desktops and servers. Additionally, KVFS provides snapshots, a low-overhead concurrent transactional API, and keeps FUSE kernel caches enabled and consistent. We have also designed a simple write-back caching scheme for FUSE that would improve the write performance of KVFS further.

Future Work. (1) Stitching can cause file system fragmentation over time; we have developed an offline defragmentation tool. We want to extend it to run online and also explore other defragmentation techniques. (2) We want to complete the implementation of the write-back cache for FUSE's kernel module and measure the performance improvement for large workloads consisting of writes. (3) We also want to extend KVFS to do in-line deduplication. VT-tree and LSM-trees in general are suitable for deduplication as they eliminate any duplicates during compactions. In KVFS, a simple way to implement in-line deduplication is to change the format of the *dmap* entries to include a checksum or hash of the data block as the key, and a separate dictionary format to map file data-blocks to their respective hashes. (4) We are currently using one VT-tree with *dmap* format to store data blocks belonging to all the files in the file system. We want to explore using VT-tree with no RAM-buffers to store individual file data separately. This ensures that the data belonging to a file is always sequential, improving the performance of sequential reads of any file in the file system. So, there will be as many VT-trees as there are files in the file system. This may require supporting zone sizes of 4KB to accommodate small files in the VT-tree. However, large zones are also necessary to maintain the sequentialty on the disk for large lists in a VT-tree backing a large file, requiring an efficient zone allocation and management functionality.

Bibliography

[1] N. Agrawal, W. J. Bolosky, J. R. Douceur, and J. R. Lorch. A five-year study of file-system metadata. In *FAST '07: Proceedings of the 5th USENIX conference on File and Storage Technologies*, pages 3–3, Berkeley, CA, USA, 2007. USENIX Association.

[2] The Apache Foundation. Hadoop, January 2010. `http://hadoop.apache.org`.

[3] M. A. Bender, M. Farach-Colton, J. T. Fineman, Y. R. Fogel, B. C. Kuszmaul, and J.Nelson. Cache-oblivious streaming b-trees. In *SPAA '07: Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, pages 81–92, New York, NY, USA, 2007. ACM.

[4] M. A. Bender, M. Farach-Colton, R. Johnson, B. C. Kuszmaul, D. Medjedovic, P. Montes, P. Shetty, R. P. Spillane, and E. Zadok. Don't thrash: How to cache your hash on flash. In *HotStorage '11: Proceedings of the 3rd USENIX Workshop on Hot Topics in Storage*, June 2011.

[5] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, 1975.

[6] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

[7] Silas Boyd-Wickizer, Haibo Chen, Rong Chen, Yandong Mao, M. Frans Kaashoek, Robert Morris, Aleksey Pesterev, Lex Stein, Ming Wu, Yuehua Dai, Yang Zhang, and Zheng Zhang. Corey: An operating system for many cores. In *OSDI*, pages 43–57, 2008.

[8] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: a distributed storage system for structured data. In *OSDI '06: Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, pages 15–15, Berkeley, CA, USA, 2006. USENIX Association.

[9] BDB Java Edition. Bdb java edition white paper. `www.oracle.com/technetwork/database/berkeleydb/learnmore/bdb-je-architecture-whitepaper-366830.pdfi`, September 2006.

[10] Filebench, July 2011. `http://filebench.sourceforge.net`.

[11] D. Giampaolo. *Practical File System Design with the Be File System*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1998.

[12] D. K. Gifford, R. M. Needham, and M. D. Schroeder. The Cedar File System. *Communications of the ACM*, 31(3):288–298, 1988.

[13] G. Graefe. Write-optimized b-trees. In *VLDB '04: Proceedings of the Thirtieth international conference on Very large data bases*, pages 672–683. VLDB Endowment, 2004.

[14] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, San Mateo, CA, 1993.

[15] D. Hitz, J. Lau, and M. Malcolm. File System Design for an NFS File Server Appliance. In *Proceedings of the USENIX Winter Technical Conference*, pages 235–245, San Francisco, CA, January 1994. USENIX Association.

[16] IEEE/ANSI. Information Technology–Portable Operating System Interface (POSIX)–Part 1: System Application: Program Interface (API) [C Language]. Technical Report STD-1003.1, ISO/IEC, 1996.

[17] Avinash Lakshman and Prashant Malik. Cassandra: structured storage system on a p2p network. In *Proceedings of the 28th ACM symposium on Principles of distributed computing*, PODC '09, pages 5–5, New York, NY, USA, 2009. ACM.

[18] A. W. Leung, M. Shawo, T. Bisson, S. Pasupathy, and E. L. Miller. Spyglass: Fast, scalable metadata search for large-scale storage systems. In *FAST '09: Proceedings of the 7th USENIX conference on File and Storage Technologies*, Berkeley, CA, USA, 2009. USENIX Association.

[19] M. Mammarella, S. Hovsepian, and E. Kohler. Modular data storage with anvil. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 147–160, New York, NY, USA, 2009. ACM.

[20] Microsoft Corporation. Microsoft MSDN WinFS Documentation. `http://msdn.microsoft.com/data/winfs/`, October 2004.

[21] K. Muniswamy-Reddy, C. P. Wright, A. Himmer, and E. Zadok. A Versatile and User-Oriented Versioning File System. In *Proceedings of the Third USENIX Conference on File and Storage Technologies (FAST 2004)*, pages 115–128, San Francisco, CA, March/April 2004. USENIX Association.

[22] N. Murphy, M. Tonkelowitz, and M. Vernal. The Design and Implementation of the Database File System. `www.eecs.harvard.edu/~vernal/learn/cs261r/index.shtml`, January 2002.

[23] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree (LSM-tree). *Acta Inf.*, 33(4):351–385, 1996.

[24] Oracle. Btrfs, 2008. `http://oss.oracle.com/projects/btrfs/`.

[25] Oracle. Bdb java edition faq. `www.oracle.com/technetwork/database/berkeleydb/je-faq-096044.html#38`, September 2011.

[26] Z. N. J. Peterson and R. C. Burns. Ext3cow: The design, Implementation, and Analysis of Metadat for a Time-Shifting File System. Technical Report HSSL-2003-03, Computer Science Department, The Johns Hopkins University, 2003. `http://hssl.cs.jhu.edu/papers/peterson-ext3cow03.pdf`.

[27] Donald E. Porter, Owen S. Hofmann, Christopher J. Rossbach, Alexander Benn, and Emmett Witchel. Operating system transactions. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 161–176, New York, NY, USA, 2009. ACM.

[28] Aditya Rajgarhia and Ashish Gehani. Performance and extension of user space file systems. In *25th Symposium On Applied Computing*. ACM, March 2010.

[29] H. Reiser. ReiserFS v.3 Whitepaper. `http://web.archive.org/web/20031015041320/http://namesys.com/`.

[30] M. Rosenblum and J. Ousterhout. The Design and Implementation of a Log-structured File System. In *ACM Transactions on Computer Systems (TOCS)*, pages 26–52. ACM, 1992.

[31] B. Salmon, S. W. Schlosser, L. F. Cranor, and G. R. Ganger. Perspective: Semantic data management for the home. In *FAST '09: Proceedings of the 7th USENIX conference on File and Storage Technologies*, Berkeley, CA, USA, 2009. USENIX Association.

[32] D. J. Santry, M. J. Feeley, N. C. Hutchinson, and A. C. Veitch. Elephant: The file system that never forgets. In *Proceedings of the IEEE Workshop on Hot Topics in Operating Systems (HOTOS)*, pages 2–7, Rio Rica, AZ, March 1999.

[33] F. Schmuck and J. Wylie. Experience with transactions in QuickSilver. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP '91)*, pages 239–253, Pacific Grove, CA, October 1991. ACM Press.

[34] R. Sears, M. Callaghan, and E. Brewer. Rose: Compressed, log-structured replication. In *Proceedings of the VLDB Endowment*, volume 1, Auckland, New Zealand, 2008.

[35] M. Seltzer and N. Murphy. Hierarchical file systems are dead. In *Proceedings of the 12th Workshop on Hot Topics in Operating Systems*, 2009.

[36] M. I. Seltzer. Transaction Support in a Log-Structured File System. In *Proceedings of the Ninth International Conference on Data Engineering*, pages 503–510, Vienna, Austria, April 1993.

[37] R. Spillane, S. Dixit, S. Archak, S. Bhanage, and E. Zadok. Exporting kernel page caching for efficient user-level I/O. In *Proceedings of the 26th International IEEE Symposium on Mass Storage Systems and Technologies*, Incline Village, Nevada, May 2010. IEEE.

[38] R. P. Spillane. *Efficient, Scalable, and Versatile Application and System Transaction Management for Direct Storage Layers*. PhD thesis, Computer Science Department, Stony Brook University, Jan 2012. Technical Report FSL-12-02.

[39] R. P. Spillane, S. Gaikwad, E. Zadok, C. P. Wright, and M. Chinni. Enabling transactional file access via lightweight kernel extensions. In *Proceedings of the Seventh USENIX Conference on File and Storage Technologies (FAST '09)*, pages 29–42, San Francisco, CA, February 2009. USENIX Association.

[40] R. P. Spillane, P. J. Shetty, E. Zadok, S. Archak, and S. Dixit. An efficient multi-tier tablet server storage architecture. In *2nd ACM Symposium on Cloud Computing*. ACM, Oct 2011.

[41] M. Stonebraker. One Size Fits All: An Idea Whose Time has Come and Gone. In *In Proceedings of the International Conference on Data Engineering (ICDE)*, pages 2–11, 2005.

[42] Sun Microsystems, Inc. Solaris ZFS file storage solution. *Solaris 10 Data Sheets*, 2004. `www.sun.com/software/solaris/ds/zfs.jsp`.

[43] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck. Scalability in the XFS file system. In *Proceedings of the Annual USENIX Technical Conference*, pages 1–14, San Diego, CA, January 1996.

[44] M. Szeredi. Filesystem in Userspace. `http://fuse.sourceforge.net`, February 2005.

[45] Andy Twigg, Andrew Byde, Grzegorz Milos, Tim Moreton, John Wilkes, and Tom Wilkie. Stratified b-trees and versioned dictionaries. In *Proceedings of the 3rd USENIX conference on Hot topics in storage and file systems*, HotStorage'11, pages 10–10, Berkeley, CA, USA, 2011. USENIX Association.

[46] S. Verma. Transactional NTFS (TxF). `http://msdn2.microsoft.com/en-us/library/aa365456.aspx`, 2006.

[47] L. Walsh, V. Akhmechet, and M. Glukhovsky. Rethinkdb - rethinking database storage. `www.rethinkdb.com/papers/whitepaper.pdf`, 2009.

[48] C. P. Wright, R. Spillane, G. Sivathanu, and E. Zadok. Extending ACID Semantics to the File System. *ACM Transactions on Storage (TOS)*, 3(2):1–42, June 2007.