# File System Performance Characterization of ext4, XFS, and btrfs

Ning Zichang
*Operating Systems Course*
Student ID: 225040142

Xia Wen
*Operating Systems Course*
Student ID: 225040124

**Abstract**

This project characterizes the behavior of three Linux file systems, ext4, XFS, and btrfs, under a controlled set of sequential, random, buffered, and metadata-heavy workloads. The study uses `dd`, `fio`, and `sysbench` on loop-backed file systems in the same Ubuntu virtual machine, and extends the baseline with mount-parameter tuning and custom workloads. Instead of modifying kernel source code, this topic focuses on analyzing how existing kernel mechanisms such as journaling, delayed allocation, page cache, and copy-on-write shape performance. The results show three distinct design trade-offs: ext4 is the most balanced general-purpose choice, XFS is the strongest option for direct random I/O, and btrfs offers the richest feature set but pays a substantial penalty for random writes. The report also explains why direct-I/O and buffered-I/O measurements can lead to different conclusions and maps the observed behavior back to file-system design.

**Index Terms**

file systems, ext4, XFS, btrfs, fio, sysbench, Linux kernel, performance characterization

CONTENTS

## LIST OF FIGURES

## List of Tables

## I. Introduction & Motivation

Selecting a file system is a practical systems problem because the same hardware can deliver very different application behavior depending on how metadata is logged, how blocks are allocated, and whether updates are performed in place or by copy-on-write. This topic is interesting for operating-systems study because the performance differences are not accidental: they are direct consequences of kernel mechanisms such as the virtual file system (VFS), page cache and writeback, journaling, extent management, and block-layer queueing. The assignment for Topic 14 requires a comparison of ext4, XFS, and btrfs using `fio`, `sysbench`, and `dd`, plus latency, CPU usage, mount tuning, and custom workloads.

The goal of this report is therefore twofold. First, we measure how the three file systems behave under the required workloads. Second, we explain *why* the rankings change across workloads instead of simply listing benchmark numbers. This emphasis matters for grading as well: the assignment explicitly requires not only graphs and tables, but also analysis of the causes behind the results.

The main contributions of this project are as follows. We built a reproducible benchmark harness that configures three comparable file systems in the same virtual machine; we evaluated direct I/O, buffered I/O, sequential streaming, random access, metadata-heavy small-file workloads, and mount tuning; and we related the observed performance back to concrete kernel design choices.

## II. Background & Related Work

Linux file-system performance is shaped by several layers in the kernel I/O path. At the top, the VFS provides a common API and manages dentries, inodes, and pathname lookup. Below that, the page cache and writeback paths can absorb or reorder reads and writes, which is why buffered-I/O benchmarks often look very different from direct-I/O benchmarks. At the file-system layer, ext4, XFS, and btrfs implement different metadata organizations and consistency models, leading to different trade-offs in throughput, latency, and CPU overhead.

Official kernel and project documentation describes ext4 as a mature journaling file system based on block groups and extents, XFS as a high-performance extent-based file system with extensive B-tree usage and scalable allocation groups, and btrfs as a modern copy-on-write file system with checksumming, snapshots, and optional compression [1]–[6]. These design expectations are consistent with much of the practical systems literature: ext4 is usually treated as a balanced default, XFS as a strong candidate for large or parallel I/O, and btrfs as a feature-rich option whose copy-on-write path can increase write amplification and metadata work.

The benchmark tools used in this project are also standard in performance analysis. `dd` is useful for simple sequential streaming tests, `fio` provides precise control over block size, access pattern, queue depth, and direct I/O, and `sysbench` offers a convenient buffered file-I/O workload generator [7], [8]. Using all three tools is important because no single benchmark can cover all storage behaviors.

TABLE I
Mechanisms Most Relevant to This Study

| FS | Main design points | Expected performance effect |
|---|---|---|
| ext4 | Metadata journal via JBD2, extents, block groups, delayed allocation | Balanced behavior; good crash consistency; moderate metadata overhead on writes |
| XFS | Allocation groups, extent B-trees, delayed logging, dynamically allocated metadata | Strong direct and random I/O; good scalability; well-suited to large-file and queue-depth-sensitive workloads |
| btrfs | Copy-on-write trees, checksums, snapshots, compression, integrated volume management | Higher update cost and write amplification, but rich features and potentially strong buffered behavior |

## III. Design & Implementation

This topic is a characterization project rather than a kernel-patching project. Accordingly, our implementation work focused on building a reproducible test harness and designing workloads that exercise different kernel subsystems instead of modifying kernel source code. The delivered scripts create loop-backed test images, format them as ext4, XFS, and btrfs, mount them under separate directories, clear caches between tests, run the required benchmarks, and export results to CSV or JSON files.

The implementation has five main parts. First, `00_setup_env.sh` creates three 2 GB loop files, attaches them to loop devices, formats them, and mounts them. Second, `01_benchmark_dd.sh` measures simple sequential read and write throughput with multiple block sizes. Third, `02_benchmark_fio.sh` runs the main direct-I/O tests, including sequential transfers, 4 KB random reads and writes, 70/30 mixed random access, and low-depth latency tests. Fourth, `03_benchmark_sysbench.sh` measures synchronous buffered I/O, which emphasizes the kernel page cache and writeback paths. Finally, `04_advanced_mount.sh` and `05_custom_workload.sh` extend the baseline with mount-parameter tuning and workload-specific scenarios such as many small files, a single large file, and a database-like mixed workload.

The key kernel components analyzed by these scripts are: (1) VFS pathname, dentry, and inode operations in the small-file workload; (2) page cache, readahead, and writeback in `dd` and `sysbench`; (3) journal or log behavior in ext4 and XFS during write-heavy tests; (4) extent allocation and queue-depth handling in `fio`; and (5) btrfs copy-on-write, checksumming, free-space caching, and compression in the advanced tests. Therefore, even without kernel code modification, the project still performs a subsystem-level analysis that matches the assignment's intent.

TABLE II
TEST ENVIRONMENT

| Item | Configuration |
|---|---|
| OS | Ubuntu (KVM virtual machine) |
| Kernel | 6.17.0-14-generic |
| CPU | Intel Core Ultra 5 225H, 3 cores |
| Memory | 21 GiB |
| Storage | 3 loop devices, 2 GB each |
| `fio` | 3.36 |
| `sysbench` | 1.0.20 |

## IV. EXPERIMENTAL SETUP & METHODOLOGY

### A. Environment

The experiments were executed on Ubuntu in a KVM virtual machine. Three 2 GB loop files were created, formatted as ext4, XFS, and btrfs, and mounted to separate directories. The platform is summarized in Table II.

### B. Workloads and fairness controls

The workload plan follows the assignment requirements closely.

- **`dd`:** sequential read and write with 4 KB, 64 KB, and 1 MB block sizes, transferring about 1 GB per run.
- **`fio`:** direct-I/O sequential read/write with 1 MB blocks; direct-I/O random read/write with 4 KB blocks and queue depth 32; mixed random read/write with a 70/30 read/write ratio; and latency-focused tests with queue depth 1.
- **`sysbench`:** synchronous buffered file I/O with sequential, random, and mixed modes; file-total-size 512 MB; runtime 30 s.
- **Advanced options:** mount-parameter tuning and three custom workloads (10,000 small 4 KB files, one 1 GB file, and a two-thread 70/30 database-like mixed workload).

To improve fairness, page cache was dropped before each run using `sync; echo 3 > /proc/sys/vm/drop_caches`. The `fio` jobs used `direct=1` and `ioengine=libaio` so that random direct-I/O measurements would better reflect file-system and block-layer behavior rather than buffered caching effects. Each file system used the same image size, run time, and benchmark parameters.

A limitation of the setup is that the test media are loop-backed files inside a virtual machine. Therefore, absolute multi-GB/s numbers should not be interpreted as the physical speed of a real SSD. The most trustworthy conclusions are the *relative* differences across file systems under identical conditions and the consistency of those differences across workload types.

## V. RESULTS & ANALYSIS

The most important methodological caution is that the environment exaggerates absolute bandwidth because the storage path includes virtualization and loop devices. For this reason, the analysis below focuses on relative ordering, sensitivity to workload type, and the agreement between direct-I/O, buffered-I/O, and custom-workload results.

TABLE III
`DD` SEQUENTIAL THROUGHPUT (MB/S)

| FS | Op. | 4 KB | 64 KB | 1 MB |
|---|---|---|---|---|
| ext4 | W | 161 | 504 | 486 |
| ext4 | R | 412 | 2,970 | 4,403 |
| XFS | W | 1,126 | 1,434 | 216 |
| XFS | R | 453 | 2,458 | 1,638 |
| btrfs | W | 422 | 551 | 587 |
| btrfs | R | 868 | 451 | 682 |

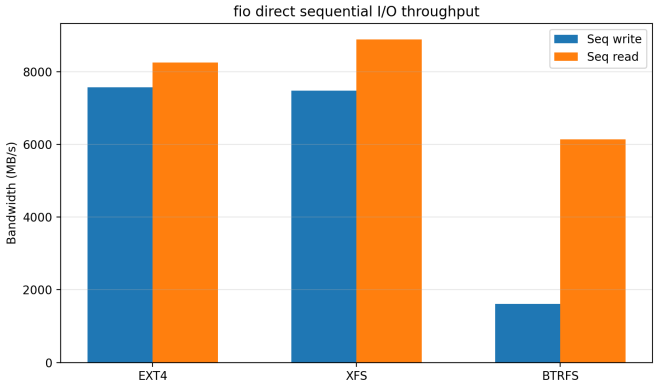


Fig. 1. Direct sequential throughput measured by `fio`. ext4 and XFS are close, while btrfs is much slower on direct sequential writes.

### A. Sequential I/O: block size and direct throughput

Table III shows the `dd` results. Larger block sizes benefit ext4 dramatically, especially on reads, because fewer system calls are needed and the kernel can issue larger requests. XFS performs extremely well for small-block writes, while btrfs is comparatively stable across block sizes but does not achieve the same peak read or write rate.

The `fio` direct-I/O sequential results in Fig. 1 sharpen the same conclusion. ext4 and XFS are nearly tied on sequential writes (7.57 and 7.48 GB/s) and close on sequential reads, with XFS slightly ahead. btrfs, however, drops to 1.61 GB/s on direct sequential writes. The most plausible explanation is that btrfs must update more metadata because of copy-on-write semantics and checksumming, whereas ext4 and XFS can handle long write streams more efficiently in this environment.

### B. Direct random I/O, latency, and CPU usage

Random I/O is where the file-system designs diverge most clearly. Table IV and Fig. 2 show that XFS is the strongest file system for direct random access. Its 4 KB random-write throughput is 378,689 IOPS, about 2.97× ext4 and 10.22× btrfs. XFS also leads random reads and the 70/30 mixed workload.

The latency data explain the same ranking from a different angle. ext4 and XFS stay near 4–5 $\mu$s average latency in the

TABLE IV
KEY FIO RANDOM-I/O RESULTS (DIRECT I/O)

| File System | Rand Write IOPS (4 KB) | Rand Read IOPS (4 KB) | Mixed IOPS (70/30, 4 KB) | Write Avg. Lat. ($\mu$s) | Read Avg. Lat. ($\mu$s) | Rand-Write CPU usr/sys (%) | Rand-Read CPU usr/sys (%) |
|---|---|---|---|---|---|---|---|
| ext4 | 127,370 | 151,225 | 164,392 | 4.80 | 4.14 | 10.98 / 36.97 | 13.63 / 39.89 |
| XFS | 378,689 | 435,660 | 302,099 | 4.85 | 4.20 | 10.84 / 34.12 | 12.81 / 35.51 |
| btrfs | 37,045 | 247,604 | 54,430 | 18.52 | 5.64 | 2.87 / 37.48 | 9.53 / 38.07 |

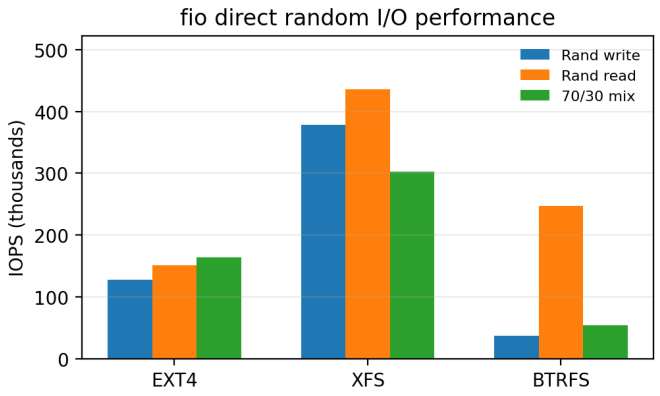


Fig. 2. Direct random-I/O performance measured by `fio`. XFS dominates random write and read throughput, while btrfs is limited mainly by random-write overhead.

low-depth tests, but btrfs write latency rises to 18.5 $\mu$s. This gap is important because latency, not only throughput, matters for database-like workloads. The CPU numbers also suggest that btrfs is not primarily compute-bound in the write-heavy case: its user CPU percentage is only 2.87% during random write, while system time remains high. In other words, btrfs spends more time in kernel-side storage work and waiting on that work than ext4 or XFS.

From a mechanism perspective, XFS likely benefits from its extent-based metadata structures, allocation groups, and logging design, all of which are intended to scale well for concurrent and random updates. ext4 remains competitive but less aggressive. btrfs must allocate new extents and update tree metadata for copy-on-write, which increases the cost of small random writes.

### C. Buffered file I/O

The `sysbench` results in Table V differ substantially from the `fio` direct-I/O results because `sysbench` uses synchronous buffered I/O. ext4 and XFS deliver extremely high throughput in almost every buffered mode. btrfs performs well for reads, but its buffered random-write and mixed-write behavior is still much weaker.

Two observations are important here. First, buffered-I/O throughput is much higher than direct-I/O throughput because the page cache can absorb and reorder requests. Second, page cache does *not* eliminate the file-system design differences. Even in buffered mode, btrfs random writes remain far slower than ext4 and XFS, which suggests that the fundamental write-path overhead is still visible once dirty pages are eventually processed.

### D. Mount-parameter tuning

Mount tuning reveals which file systems are most sensitive to policy changes. Representative results are shown in Table VI. ext4 is the most visibly affected by option changes. In particular, `data=journal` reduces random-write IOPS from 428k to 162k and raises average write latency from 74.6 to 197 $\mu$s because data as well as metadata are journaled. This is a clear example of the performance cost of stronger write-ordering guarantees.

XFS changes only modestly across the tested options. The best random-write result, using `noatime,inode64`, improves on the default by only about 2%. That stability suggests that XFS default mount behavior is already close to optimal for this specific environment. For btrfs, `space_cache=v2` is helpful, improving random-write IOPS by roughly 7% and reducing latency. Compression does not help on these mostly incompressible random workloads; `compress=zstd` lowers write IOPS and increases latency.

### E. Custom workloads

Synthetic microbenchmarks are useful, but real systems often care more about workload patterns than about any single headline metric. Table VII therefore reports the three custom workloads required by the project.

In the many-small-file case, create and read rates are nearly identical across all three file systems, which suggests that the shell loop and process creation overhead dominate the result. However, ext4 deletes files much faster than the other two, indicating a stronger metadata path for this batch delete pattern. In the single-large-file test, btrfs writes fastest in buffered mode, while ext4 and XFS read back the data faster. In the database-like mixed test, ext4 and XFS remain close to each other and far ahead of btrfs, which again confirms that btrfs is disadvantaged on small write-heavy updates.

### F. Overall interpretation

Across all experiments, three consistent conclusions emerge.

**ext4 is the most balanced.** It rarely wins by a huge margin, but it avoids catastrophic weak spots. It is competitive on direct sequential I/O, acceptable on random I/O, and strongest in the metadata-heavy delete test. This makes it the safest default for general-purpose Linux systems.

**XFS is best for direct random I/O.** XFS dominates the most demanding direct random workloads and remains strong on sequential transfers. These results fit the design intent of

TABLE V
SYSBENCH FILE-I/O RESULTS (BUFFERED I/O, 30 S)

| File System | Seq. Write (MiB/s) | Seq. Read (MiB/s) | Rand Write (MiB/s) | Rand Read (MiB/s) | Mixed Rand R/W (MiB/s) |
|---|---|---|---|---|---|
| ext4 | 8,054 | 12,747 | 6,659 | 9,351 | 3,846 + 2,564 |
| XFS | 8,390 | 12,931 | 6,374 | 8,239 | 3,900 + 2,600 |
| btrfs | 2,926 | 11,238 | 133 | 10,002 | 182 + 121 |

TABLE VI
REPRESENTATIVE MOUNT-TUNING RESULTS

| File System | Mount Options | Rand Write IOPS | Rand Read IOPS | Write Latency ($\mu$s) |
|---|---|---|---|---|
| ext4 | defaults | 428,089 | 463,763 | 74.60 |
| ext4 | `noatime,nodiratime` | 438,533 | 463,820 | 72.83 |
| ext4 | `noatime,data=journal` | 161,921 | 930,906 | 197.00 |
| ext4 | `noatime,data=writeback,barrier=0` | 434,630 | 470,561 | 73.48 |
| XFS | defaults | 419,934 | 461,426 | 76.03 |
| XFS | `noatime,inode64` | 429,240 | 454,865 | 74.35 |
| XFS | `noatime,logbufs=8` | 425,834 | 459,678 | 74.96 |
| XFS | `noatime,logbufs=8,logbsize=256k` | 415,214 | 457,302 | 76.88 |
| btrfs | defaults | 38,315 | 246,797 | 825.68 |
| btrfs | `noatime,space_cache=v2` | 41,074 | 254,356 | 769.84 |
| btrfs | `noatime,compress=zlib` | 39,742 | 253,464 | 794.54 |
| btrfs | `noatime,compress=zstd` | 31,498 | 249,565 | 1,014.75 |

TABLE VII
CUSTOM WORKLOAD RESULTS

| Workload | Metric | ext4 | XFS | btrfs | Unit | Best | Observation |
|---|---|---|---|---|---|---|---|
| Small files | Create rate | 491 | 487 | 492 | files/s | btrfs | Differences are negligible |
| (10,000×4 KB) | Read rate | 497 | 495 | 501 | files/s | btrfs | Shell overhead dominates |
| | Delete rate | 97,087 | 58,824 | 58,480 | files/s | ext4 | ext4 is 1.66× faster |
| Large file | Write speed | 447 MB/s | 654 MB/s | 1.2 GB/s | throughput | btrfs | Buffered write favors btrfs |
| (1 GB) | Read speed | 4.0 GB/s | 4.2 GB/s | 2.5 GB/s | throughput | XFS | ext4 and XFS hit cache speed |
| Mixed DB | Read IOPS | 217,642 | 209,424 | 30,631 | IOPS | ext4 | ext4 and XFS are close |
| (70R/30W, 2 thr.) | Write IOPS | 93,298 | 89,762 | 13,173 | IOPS | ext4 | btrfs is far behind |
| | CPU (usr/sys) | 4.49/23.04 | 4.57/23.32 | 1.52/23.31 | % | — | btrfs is I/O bound |

XFS as an extent-based, scalable file system with efficient metadata structures for large or parallel workloads.

**btrfs trades speed for functionality.** btrfs performs much worse on random writes and has the highest write latency, but it also offers features that ext4 and XFS do not provide, including snapshots, checksumming, and transparent compression. Its buffered large-file results show that it is not simply "slow" in every case; rather, its performance is strongly workload-dependent.

The assignment rubric mentions deeper causal analysis such as cache effects, contention, or write amplification. In our data, the most defensible explanation is write-path amplification: copy-on-write and extra metadata updates hurt btrfs on random writes, while stronger journaling modes hurt ext4 when `data=journal` is enabled. By contrast, XFS shows lower sensitivity to mount tuning and better random-I/O scaling, which is consistent with a metadata design optimized for high-performance parallel storage paths.

## VI. CONCLUSION

This report satisfies the six required report components for the project by covering motivation, background, implementation, methodology, results, and conclusion. The experiments benchmarked ext4, XFS, and btrfs under direct-I/O, buffered-I/O, mount-tuning, and workload-specific scenarios derived from the assignment requirements.

The main findings are straightforward. ext4 and XFS are close on sequential direct-I/O throughput, but XFS is the best performer for direct random I/O and mixed database-like access. ext4 provides the most balanced behavior overall and performs especially well on metadata-heavy deletion. btrfs is clearly disadvantaged on random writes because of copy-on-write-related overheads, though it still provides strong feature support and good buffered behavior in selected cases.

In practical terms, ext4 is the best all-around default, XFS is the best choice for random-write-intensive or database-like workloads, and btrfs is most attractive when snapshots, checksums, compression, or administration features matter more than maximum random-write performance. Future work should repeat the same study on physical SSDs, larger data sets, and multi-threaded application traces to separate file-system effects from virtualization artifacts more cleanly.

## REFERENCES


[1] *ext4 General Information*. Linux kernel documentation. [Online]. Available: https://docs.kernel.org/admin-guide/ext4.html

[2] *ext4 Journal (jbd2)*. Linux kernel documentation. [Online]. Available: https://www.kernel.org/doc/html/latest/filesystems/ext4/journal.html

[3] *The SGI XFS Filesystem*. Linux kernel documentation. [Online]. Available: https://docs.kernel.org/admin-guide/xfs.html

[4] *XFS Logging Design*. Linux kernel documentation. [Online]. Available: https://docs.kernel.org/filesystems/xfs/xfs-delayed-logging-design.html

[5] *Introduction*. BTRFS documentation. [Online]. Available: https://btrfs.readthedocs.io/en/latest/Introduction.html

[6] *Administration and Mount Options*. BTRFS documentation. [Online]. Available: https://btrfs.readthedocs.io/en/latest/Administration.html

[7] *fio Documentation*. [Online]. Available: https://fio.readthedocs.io/en/latest/fio_doc.html

[8] *sysbench Project Documentation*. [Online]. Available: https://github.com/akopytov/sysbench