

Measure Twice, Cut Once: the Best* Way to Calculate macOS File Hierarchy Size

Artem Levchenko¹ Oleksandr Frankiv¹ Yevhenii Peteliev²
artem.levchenko@ukma.edu.ua o.frankiv@ukma.edu.ua zhenya.peteliev@macpaw.com

¹National University of Kyiv-Mohyla Academy, Kyiv, Ukraine

²MacPaw, Kyiv, Ukraine

Abstract

Efficient file system scanning and sizing of files and directories in the Apple File System (APFS) is essential for storage management and system monitoring. The aim of this study is to determine the most efficient algorithm for measuring the size of APFS file hierarchies, with a focus on improving speed, scalability, and resource efficiency. We propose a solution that integrates three APFS access methods - `NSFileManager`, `URLResourceKey`, and the UNIX utility `du` - with four traversal algorithms (top level, full system bypass, lazy bypass, and stop-words filtering) and three multithreading processing methods (serial processing, Grand Central Dispatch, and Swift concurrency). Experiments on synthetic file hierarchies show that `URLResourceKey` achieves scan times of an order of magnitude shorter than `NSFileManager` and consistently outperforms `du`. Using multithreaded processing with Swift concurrency or GCD leads to a 2-3x reduction in total execution time compared to serial processing. Employing different traversal algorithms reduces scanning time, improves user experience, and enables customization for specific tasks.

1 Introduction

File systems are an integral part of any modern operating system. They play an important role in fast access to data, reliable data storage, and efficient use of space. Apple File System (APFS) was introduced on June 14, 2016 as the next generation file system for Apple devices (Tamura and Giampaolo, 2016). It replaced Hierarchical File System Plus (HFS+), which had been in use since 1998¹. In 2024, macOS was the second most popular desktop operating system, and the total market share of Apple operating systems was about 25% (Kosisochukwu and Abdullahi, 2024).

¹<https://developer.apple.com/library/archive/technotes/tn/tn1150.html>

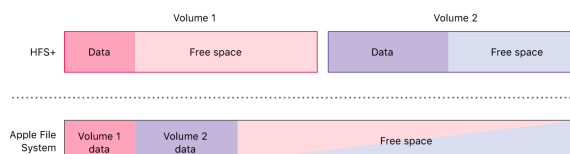


Figure 1: Space allocation between volumes in HFS+ and APFS.

The growing popularity of Apple devices necessitates a deeper understanding of the internal mechanisms of the file system of these devices, as well as optimization of the scanning process, which underlies such tasks as backup, indexing, antivirus scanning, data recovery, and digital forensics.

The Apple File System has brought significant changes that aim to improve data access, ensure data integrity, optimize storage management, and security. Unlike HFS+, where each volume has a fixed size and does not share free space with other partitions, APFS introduced space sharing (Rane and Singh, 2024). This allows multiple volumes in a container to dynamically allocate storage, flexibly resizing without the need to reformat.

APFS uses a copy-on-write mechanism that ensures that data are not overwritten until the changes are completely complete. This helps minimize the risk of data corruption and increases the overall stability of the file system². For file and directory management, APFS uses a highly optimized B-tree structure that speeds up the processes of copying, deleting, and renaming files³.

One of the key features of the APFS file system is the ability to create snapshots that capture the state of the file system at a particular point in time. They help quickly recover data without using additional space for duplicate data (Hansen and Toolan, 2017).

²<https://developer.apple.com/documentation/foundation/about-apple-file-system>

³<https://developer.apple.com/support/downloads/Apple-File-System-Reference.pdf>

All these features of the Apple File System make it a high-performance and reliable file system, but at the same time, they complicate the process of scanning it. That is why APFS scanning optimization is an important topic of modern research, as the speed and accuracy of this process directly affect the overall system performance. In this article, we will focus on analyzing the methods and algorithms that will allow you to scan APFS as efficiently as possible and determine the optimal solution.

2 Measuring and Managing APFS

In this research, we focus on determining the size of the file hierarchy. In the APFS, each element is characterized by two types of sizes: logical and physical. The logical size (actual size) reflects the amount of data actually written to a file or directory, whereas the physical size (allocated size or size on disk) determines the amount of disk space reserved by the file system to store this item. In our research, we will use the physical size for APFS scanning, as it provides a more accurate assessment of the structure and space utilization.

The interaction with APFS can be achieved through several tools, specifically the `NSFileManager` and `URLResourceKey` APIs and through the system's `du` command. `NSFileManager`⁴ is the basic API to interface with the Apple File System. It provides convenient access to files and directories. The `attributesOfItem(atPath:)` method allows to get the file size using the `.size` attribute. However, it requires a recursive traversal to calculate the size of the directory. This makes the scanning process slow and resource-intensive, especially for large file structures. In addition, `NSFileManager` cannot provide information about the physical size of files on the disk.

`URLResourceKey`⁵ is a more modern interface that uses URL objects to securely work with the file system. The `fileSizeKey` and `fileAllocatedSizeKey` keys allow to get the logical and physical size of files, respectively. Calculating the size of directories also requires a recursive approach, but `URLResourceKey` optimizes this process by processing only the necessary attributes and improves scanning performance compared to `NSFileManager`. Moreover, using

⁴<https://developer.apple.com/documentation/foundation/filemanager>

⁵<https://developer.apple.com/documentation/foundation/urlresourcekey>

`URLResourceKey` reduces errors from malformed file paths and offers a more versatile API.

The terminal command `du` (“disk usage”) reports the disk space used by files and directories. It displays sizes in 512-byte blocks (Nordvik, 2022). The `du` command does not require a recursive approach to traverse the file hierarchy. It supports options such as `-a` (display sizes for all files), `-s` (summarize total size without detailing subdirectories) and `-d` (limit output to a specified depth) (Platt, 2021). In Swift, `du` can be integrated via the `Process` class. However, data processing is possible only after the command has been completed, and this can slow down the scanning process. Additionally, `du` may encounter permission errors when accessing certain directories, necessitating extra error handling.

3 Traversal Algorithms

Effective scanning of the Apple File System requires employing algorithmic strategies that traverse the file hierarchy, account for its structural complexity, and minimize redundant operations. The primary traversal algorithms include top-level approach, full system bypass, stop-words filtering, and lazy bypass.

The top-level approach scans only items in the root directory and offers high speed by minimizing file system accesses. Using the `du` command with the `-d 1` option ensures minimal resource consumption. However, `NSFileManager` and `URLResourceKey` require the use of recursive traversal, which reduces efficiency, especially for `NSFileManager`. `URLResourceKey` retains relatively high performance by leveraging optimized APFS queries.

Full system bypass scans the entire hierarchy and provides detailed size information for files and directories. The `du` command with the `-a` option generates a complete list of all items in the file tree, while `NSFileManager` and `URLResourceKey` retrieve size data via the `.size` attribute and the `.fileAllocatedSizeKey` key, respectively. Although this approach yields the highest accuracy, its execution time can be significant, potentially inconvenient for users.

Lazy bypass provides gradual results and improves the user experience. This algorithm is implemented using `AsyncStream`⁶. Files and direc-

⁶<https://developer.apple.com/documentation/swift/asyncstream>

tories are added to the asynchronous stream using the `yield(_:)` and the stream is closed with `finish()`. The lazy bypass allows us to start processing data before the scanning process is complete and makes scanning more flexible and responsive.

Stop-words filtering optimizes APFS scanning by eliminating unnecessary items whose names match specified stop words. The implementation uses the DFS and BFS algorithms. The DFS (Kaur and Garg, 2012) algorithm performs a recursive traversal of directories and skips elements containing stop words, and has a higher speed for deep structures. In contrast, BFS (Holdsworth, 1999) processes elements level by level using a queue, enabling faster analysis of top-level directories. In practice, DFS outperforms BFS in APFS scanning speed, but BFS remains a fairly efficient algorithm.

The choice of scanning algorithm for APFS depends on the goal: top-level approach for rapid overview; full system bypass for detailed analysis; lazy bypass for user convenience; stop-words filtering for resource optimization.

4 Multithreaded Processing of the APFS Hierarchy

Serial processing executes all operations on a single thread. It is useful for small directories or individual files and is easy to debug. However, it performs poorly on complex multilevel file hierarchies. Therefore, to speed up the scanning of the APFS file system, multithreaded processing is used, implemented using Swift concurrency and Grand Central Dispatch (GCD).

Swift concurrency⁷ was introduced in Swift 5.5 and utilizes `async/await`, `TaskGroups`, and actors for structured concurrency. The `withThrowingTaskGroup` function allows us to create and manage groups of asynchronous tasks that run in parallel while handling potential errors. Actors prevent data race problems by automatically controlling access to resources, eliminating the need for manual locks such as `NSLock` and reducing the risk of deadlocks.

Grand Central Dispatch (GCD)⁸ manages parallel tasks using `DispatchQueue` and `DispatchGroup` to synchronize their execution. The `withCheckedThrowingContinuation`

⁷<https://developer.apple.com/documentation/swift/concurrency/>

⁸<https://developer.apple.com/documentation/Dispatch>

function bridges GCD with the modern `async/await` paradigm. Although GCD is an older technology, it remains a robust and efficient solution.

5 Solution Architecture

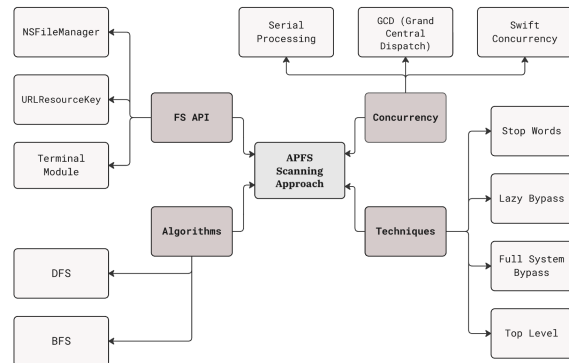


Figure 2: Structure of the developed solution.

The developed APFS file system scanning solution is based on a modular architecture that provides flexibility and scalability. As shown in Figure 2, it consists of the following key components:

- **FS API Protocol**, which interacts with the file system via `NSFileManager`, `URLResourceKey`, or the `du` command;
- **Concurrency Module**, which handles both sequential and concurrent processing using Swift Concurrency or GCD;
- **Techniques Module**, which implements traversal strategies - top-level approach, full system bypass, lazy bypass and stop-words filtering using DFS and BFS.

This modular design enables easy replacement or adaptation of algorithms and APIs depending on specific requirements to ensure efficient scanning of the Apple File System.

6 Evaluation

To evaluate the effectiveness of APFS scanning methods, tests were performed using custom datasets that simulate various file hierarchies to assess performance in real-world scenarios. The data sets were generated with a custom tool that supports four patterns: Breadth-First Structure (BFS) for testing wide hierarchies, Depth-First Structure (DFS) for deep nesting, Balanced Tree Structure for uniform structures, and Unbalanced Tree Structure for realistic irregular scenarios. The test was

performed on a computer running macOS Sequoia 15.4.1 with an SSD using APFS. The scanning algorithms were evaluated across all patterns with varying file sizes (from empty to random). During testing, we evaluated the execution time and peak memory usage.

Figure 3 compares the execution time (ms, log scale) of scanning methods (du, NSFileManager, and URLResourceKey) across four hierarchy types: BFS, DFS, balanced and unbalanced. URLResourceKey consistently delivers the best results. NSFileManager shows the worst performance, outperforming du only on the balanced structure.

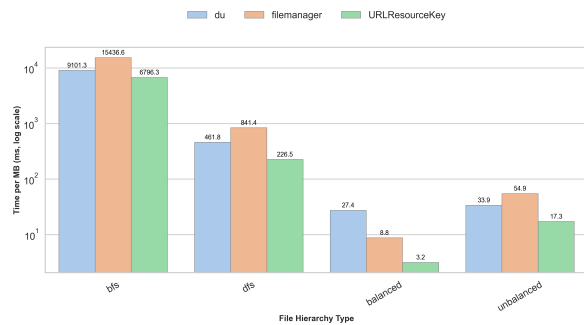


Figure 3: Comparison of execution time per MB for du, NSFileManager, and URLResourceKey on bfs, dfs, balanced, and unbalanced hierarchies.

Figure 4 compares the mean execution times (ms) of two traversal algorithms: full system bypass and top-level approach using NSFileManager, URLResourceKey, and du. The top-level approach does not yield improvements with NSFileManager, as it still requires recursive traversal of the entire tree, even in the top-level approach. However, when using URLResourceKey and du, the top-level approach offers an average time improvement of 2–3 times compared to full system bypass.

Figure 5 shows the mean execution times for three processing modes single-threaded (or serial), GCD, and Swift concurrency in four directory structures with NSFileManager on the left and URLResourceKey on the right. GCD and Swift Concurrency deliver comparable performance, and both multithreaded approaches reduce scanning time by a factor of 2–3 compared to single-threaded execution.

Table 1 presents the average RAM consumption across different file system APIs and concurrency modes. NSFileManager consistently consumes the most memory, while du is the most efficient, showing the lowest usage. Additionally, the concurrency

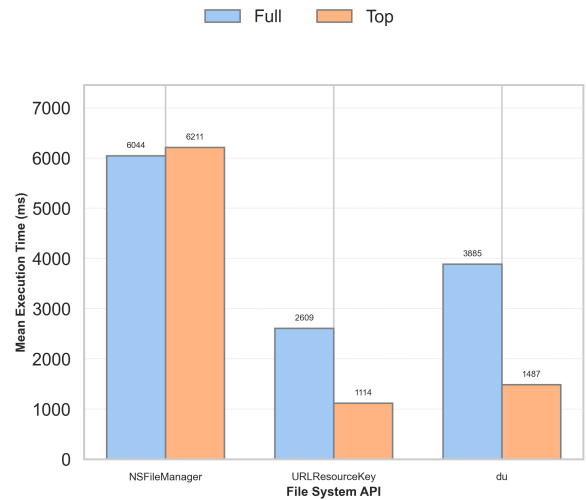


Figure 4: Mean execution time (ms) for full system bypass and top-level approach using NSFileManager, URLResourceKey, and du.

strategy has a significant impact on memory usage. Swift concurrency has the highest memory overhead, GCD has moderate overhead, and serial processing has the lowest overhead.

File System API	GCD (MB)	Serial (MB)	Swift (MB)
NSFileManager	222.0	226.5	241.1
URLResourceKey	96.4	70.2	113.9
du	59.9	43.5	65.7

Table 1: Average memory usage by file system API and concurrency mode

7 Conclusions and Future Work

This work demonstrates significant potential for improving the efficiency of APFS file hierarchy sizing by adapting algorithms to its unique architectural features. The solution that unifies multiple interfaces and algorithms has achieved substantial reductions in scanning time. Evaluation on synthetic file hierarchies confirmed consistent performance gains without compromising accuracy.

Parallel processing with GCD and Swift Concurrency accelerates scanning by 2–3 times, while the combination of Swift Concurrency and URLResourceKey proved the most effective for typical user file hierarchies. Furthermore, the implemented traversal algorithms demonstrated high efficiency in diverse tasks, enabling flexible optimization tailored to specific scanning needs.

Future work will apply these techniques to real-world datasets and explore adaptive optimization strategies to further enhance scalability and resource utilization.

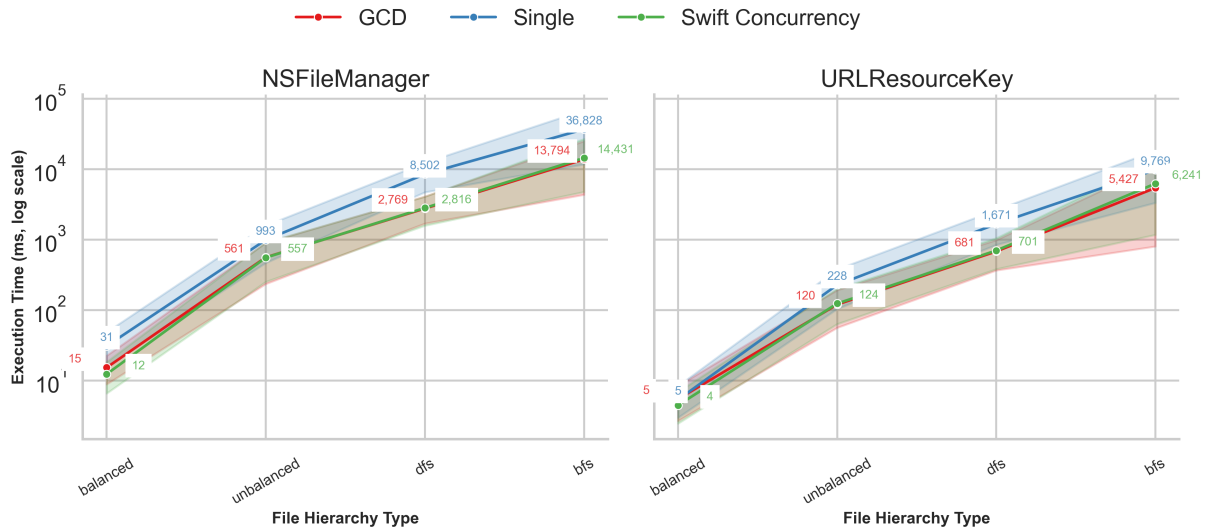


Figure 5: Mean execution time for single-threaded, GCD, and Swift concurrency processing of four APFS directory structures.

References

- Kurt H Hansen and Fergus Toolan. 2017. Decoding the apfs file system. *Digital Investigation*, 22:107–132.
- Jason Holdsworth. 1999. The nature of breadth-first search. *James Cook University School of Computer Science, Mathematics, and Physics Technical Report*, pages 99–1.
- Navneet Kaur and Deepak Garg. 2012. Analysis of the depth first search algorithms. Technical report, Computer Science and Engineering Department, Thapar University, Patiala.
- H. U. Kosisochukwu and M. I. Abdullahi. 2024. Exploring operating system diversity: A comparative analysis of windows, mac os, android and ios. *Systematic and Modern Science Research (JSMSR)*, 5(9):23–40.
- Rune Nordvik. 2022. Apfs. In *Mobile Forensics—The File Format Handbook: Common File Formats and File Systems Used in Mobile Devices*, pages 3–39. Springer.
- Daniel Platt. 2021. *Tweak Your Mac Terminal: Command Line MacOS*. Springer.
- Raksha Rane and Asmit Singh. 2024. Demystifying file systems: A comprehensive exploration of data organization.
- E. Tamura and D. Giampaolo. 2016. Introducing apple file system. Technical report, Apple Inc.