Accessing File Management Systems: Case study on Apple File Systems

Mahima Gupta (T22055)
School of Computing and Electrical Engineering
Indian Institte of Technology, Mandi
Himachal Pradesh, India
T22055@students.iitmandi.ac.in

***Abstract*— Apple File System (APFS) is the current File Management System for Apple devices. Since, it is relatively new, there is a strong need for documentation to understand the working of APFS. This study is aimed at exploring the performance & working of APFS. It also elucidates the evolution of Apple File systems, its features and what makes it different from the previous file system versions. This study also describes the reasons for interoperability gap between Apple file system and Windows NT file systems. It also discusses about Extensible File Allocation Table (exFAT) formatted volumes which are supported by both Windows and Apple machines.**

The paper also contends Apple File System with NT File Systems, supported by Windows devices. Performance Analysis of both the file systems has been conducted using metric-based analysis. Response time, Read Throughput, Write Throughput and CPU Utilization are the metrics used for measuring performance. Two factors have been considered, namely, file size and type of processors. Results show that Apple File System performs better than NT file System. It also reports that the interaction due to Type of Processor and File Size is 13.2% and 53.75% respectively.

Keywords*— *file system, metrics, files, volumes, storage, performance analysis, macOS

I. INTRODUCTION

Every Operating System has a File Management System. Primary functions of a File Management system are to manage, retrieve, share, store and manipulate files. Since late 2010s, basic expectation from a file system includes file security and file encryption services as well, not just file operations.

In a file system, physical storage is segmented into sections called *partitions* (known as containers in APFS). Each partitions acts an independent storage unit, with its own directory structures and set of files and folders. Partitions provide compatibility, protection, flexibility and help in storing files for various applications in organized manner.

In this paper, we are primarily considering file Management Systems for two Operating Systems, namely Windows 11 and macOS Ventura. Default file system for Windows 11 is NTFS (New Technology File System), although it supports ReFS (Resilient File System), FAT (File Allocation Table) and exFAT (Extended File Allocation Table) as well. Default File System for macOS Ventura is Apple File System (APFS). It also supports HFS+ (Hierarchical File System Plus), FAT and exFAT. Both APFS and NTFS are proprietary systems of Apple and Windows respectively.

In 2017, Apple Inc launched Apple File Systems in the market for mac OS versions Sierra (10.12.4) and higher, iOS 10.3 and later, tvOS (10.2) and later versions, watchOS (3.2) and later versions, and all versions of iPadOS [1]. There were two major reasons for development of Apple File Systems. First, with the advent of advancing technology, more updated version had to be launched to catch up with the latest storage capabilities. Mostly, it was focused on introducing compatibility with Solid State drives. Secondly, it was launched to overcome the problems in HFS+ File system, previous version of file system for macOS.

This paper is organized as follows. Section II discusses the previous versions supported by Apple devices and how their limitations lead to the development of Apple File Systems. It also includes the basic features and functionalities of HFS+ in brevity. Section III is focused on Apple File Systems and its background, features, advantages, and disadvantages. Section IV talks about the interoperability gaps between Apple file Systems and Windows NTFS. It also discusses exFAT and its reasons for compatibility with both Windows and Apple machines briefly.

Section V has detailed report of the performance analysis experiment designed and conducted to contend Apple File System against NTFS. All the observations, inferences and results for the experiment have been mentioned in this section.

II. EVOLUTION OF FILE MANAGEMENT SYSTEMS FOR APPLE DEVICES

In January 1984, Apple Inc launched *Macintosh File System* (MFS) for Apple Macintosh to support up to 400KB floppy disks. It is pretty evident that the major limitation of this file system was storage size. With exponentially increasing data sets, larger storage support was demanded. In September 1985, Apple Inc launched *Hierarchical File System* to support Apple's first Hard disk [2].

HFS is also referred as HFS Standard. HFS provided better performance than MFS. In MFS, to search for a file, a list of set of files stored in that folder had to be maintained. This severely impacted the performance of MFS. The Table structure was replaced by B-tree

structure in HFS. It searched files more quickly regardless of the file count.

One of the major limitations of HFS was inefficient multi-tasking. System did not allow the user to multitask proficiently as a single data structure was maintained for record keeping of files and folders. It affected the system performance significantly. Another limitation of HFS is allocation block limit of 65535. Maximum 65535 files can be stored in a volume irrespective of volume size. In January 1998, Apple Inc launched HFS+ File System which overcame the limitations of HFS file system. From macOS 8.0 onwards, HFS+ became the default file system for Apple devices [2]. It also used B-trees to store volume metadata.

A. Hierarchical File Systems Plus (HFS+)

HFS+ is a Journaling File System, unlike HFS which was a Catalog File System. HFS+ supported longer Unicode file names, up to 255 characters. Bootup time was also reduced with HFS+ as it incorporated a startup file which could be easily location by the operating system [5]. Major components of HFS+ can be given as follows –

i. Catalog File
ii. Allocation File
iii. Volume Header
iv. Attributes B-Tree

Catalog File indexes every file that has been stored on the system. Each record in the catalog file holds metadata about the files stored. Metadata contains following information about the file – Catalog Node ID, location, timestamps, text encoding, permissions, and Finder details. Finder is the graphical interface used in all Macintosh machines as a file manager [5]. The main purpose of allocation file is to track the status of free memory, blocks of hard drive that are available for storing new files.

There are mainly three volume headers. First one, beginning from 1024 bytes of the volume. It stores information about the volume i.e., volume type, count of file and folders, block size and location of the catalog file [5]. The second volume header is a replica of the first volume header, beginning from 1024 bytes of the end of the volume. It is updated only when modifications are made on special files. And the last volume header is for disk repair utilities.

Attributes B-Tree is a data structure that stores the extended attributes of files. It also contains details about extra file permissions like access control lists enabled. This information is often used by Time Machine for backup and recovery. Author information, resource forks (attaching resource to a file, example icon for a file) are some of the extended attributes. Resource Forks was originally developed by Apple and later adopted by Microsoft [4].

B. Limitations of HFS+

Timestamps in HFS+ are not stored in nanoseconds standard. Furthermore, there are no checksums for verifying data integrity. One of the major limitations of HFS+ is that it does not allow concurrent access of the file system. This implies that a shared resource cannot be modified or access by multiple resources concurrently. Another drawback of HFS+ is that it did not support dates beyond February 6, 2040 [3]. Timestamps in HFS+ are stored in epoch format, time elapsed since 1st January 1970 measured in seconds.

HFS+ provided low security, limited capacity and was incompatible with SSDs.

III. APPLE FILE SYSTEMS

A. Overview

Unlike HFS+, Apple File System is not Journaling System. High level APIs are used for handling the file system. There are no catalog files, allocation files and attribute files in APFS. It creates *Apple Containers*, which are further partitioned into volumes to form a logical container. APFS is not an extension of HFS+ [6].

Checkpoints ensure consistency and integrity of the file system. A checkpoint is created whenever file system data is flushed to APFS container. Checkpoints come in handy for system maintenance and data recovery. Each check block has an associated Checkpoint Superblock. It works with both Apple Container as well as Volume metadata. To restore data, we can traverse backwards from the current CSB to previous blocks using checkpoints.

APFS does not have any catalog file, attribute files and volume header. Rather, components of APFS are given as follows –

i. Container Superblock
ii. Checkpoint Superblock Descriptor
iii. Bitmap Structures
iv. Volume Superblock
v. File and Folder B-Tree
vi. Snapshots

APFS is divided in two layers – container layer and file system layer. Container layer entails high level information, like volume Superblock, snapshots, and encryption state. File system layer contains metadata about files and directories and file content.

All information (total number of blocks, previous checkpoints and defined limited for blocks) relating to Apple Container is stored in Container Superblock (CSB). In the hierarchy of file systems, CSB steps on the highest pedestal. In the container layer, multiple copies of the container exist. CSB is the latest copy, often referred as Block Zero. Block Zero is used for mounting process [8]. These help in protection against system crashes. Checkpoint Superblock Descriptor (CSBD) contains information about location of Bitmap structure and

metadata. CSBD contains instances of structures *checkpoint_map_phys_t* and *nx_superblock_t*. In the official Apple File Systems Reference [8], *checkpoint_map_phys_t* has been defined as structure for checkpoint mapping and *nx_superblock_t* as a structure for container superblock. Refer to Appendix A for code blocks of the structures.

Bitmap Structure tracks the used and unused blocks. Volume Superblock contains volume metadata. File and Folder B-tree functions like a Catalog file in HFS+. Information is stored in B-tree as key-value pairs. Each node in a B-tree is primarily sectioned in five areas – node metadata, list of location of keys & values, storage space for keys, storage space for values and complete tree metadata. Each B-Tree is an instance of `btree_node_phys_t` structure, defined as follows [8] –

```
struct btree_node_phys {
obj_phys_t btn_o;
uint16_t btn_flags;
 uint16_t btn_level;
 uint32_t btn_nkeys;
nloc_t btn_table_space;
nloc_t btn_free_space;
nloc_t btn_key_free_list;
nloc_t btn_val_free_list;
uint64_t btn_data[];
 btree_node_phys_t
};  typedef  struct  btree_node_phys
btree_node_phys_t;
```

Default node size has been fixed as 4096 bytes. Checkpoints are system-generated states whereas snapshots are user-generated states. Snapshots allow the user to roll the file system back to an earlier state. Therefore, it implies that if a file is a part of a snapshot, it cannot be deleted completely until that snapshot is removed. Snapshots are created for volumes, not containers. Creation of snapshots is an inexpensive task, whereas deletion of snapshot is a tedious task. Snapshots also store information in key-value pairs.

B. Basic Attributes of files in APFS

Inode number, timestamps, permissions, and ownerships are the major attributes that are associated with every file that has been stored on Apple File System. These attributes are commonly shared by the file system with its predecessor – HFS+. Basic information associated with any file can be viewed using the following command in terminal –

```
stat -x filename
```

```
mahimagupta@Mahimas-MacBook-Pro PACN % stat -x LA03.pdf
  File: "LA03.pdf"
  Size: 692919      FileType: Regular File
  Mode: (0644/-rw-r--r--)         Uid: (  501/mahimagupta)  Gid: (   20/   staff)
Device: 1,13   Inode: 13730603    Links: 1
Access: Mon May  1 01:05:44 2023
Modify: Mon May  1 01:02:55 2023
Change: Mon May  1 01:05:44 2023
 Birth: Mon May  1 01:02:54 2023
```

Figure 1: stat command to view file information.

Inode number is data structure that stores some basic information about a file. It also contains a pointer to the location of the data on the disk. Inode number and volume headers, collectively, are used to locate a file. APFS uses 64-bit inode number, implying over 9 quintillion files and directories can be supported on a single volume [7].

Some of the default Inode numbers have been mentioned below [8] –

```
#define INVALID_INO_NUM         0
#define ROOT_DIR_PARENT         1
#define ROOT_DIR_INO_NUM        2
#define PRIV_DIR_INO_NUM        3
#define SNAP_DIR_INO_NUM        6
#define PURGEABLE_DIR_INO_NUM   7
#define MIN_USER_INO_NUM        16
```

Every file has a set of timestamps associated with it, namely, Access, Modify, Change and Birth. Access, Modify, Change and Birth timestamps stores the time at which the file was last accessed, modified, changed (includes file rename, copy and other options) and created respectively [4].

Inode number, timestamps and permissions for file can be viewed in figure 1. It can be observed that the file has following permissions –

```
-rw-r--r--
```

The first bit of the above file permission describes the file type. If the above file was a directory, then first bit would have been set to ‘d’. Next three bits denote owner permissions, following three bits denote group permissions and last three bits denote access permissions for other. There are more bits associated with permissions like sticky bit, special permissions which are beyond the scope of this paper.

C. Additional Features

- *Cloning*

Creating a copy of file such that it does not occupy any additional space on the disk is known as cloning. Cloning was introduced with APFS to reduce the cost of copying. Cloning is a faster operation and power efficient as well.

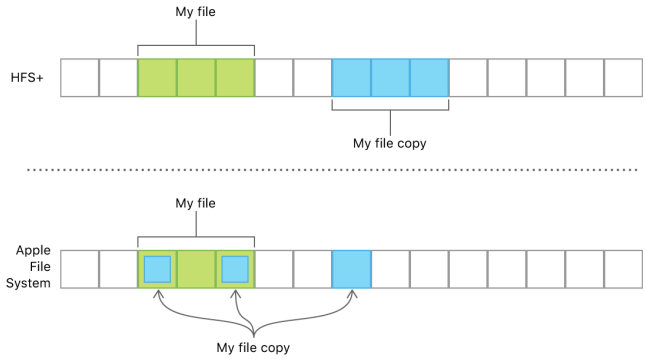

Figure 2: It shows how files are cloned in APFS by sharing few common blocks. Modifications are written separately; unmodified segments are shared by cloned files. Image reference [9].

Method copyItem() is used for cloning files. It accepts two parameters - source URL/ source path and destination URL/ destination path.

- *Data Integrity & Crash Protection*

To protect data, APFS introduced Copy-On-Write feature. It creates a unique copy of the data structure to save the modifications only when write request has been raised while the original data structure remains untouched. Both are merged only when the write operation has been committed. This provides original file in case system crashes during write operation.

The same logic has been applied to file operations as well, it ensures atomicity in file operations. Either the requested operation is completed or not done at all. This is known. As 'Atomic Safe-Save'.

- *Space Sharing*

Space Sharing adds a dynamic dimension to volume partitioning. In HFS+, volumes have a fixed size limit and a single volume could be created per partition. But in APFS, such restriction has been lifted, multiple volumes can be created per partition. Each volume can grow or shrink independently [9]. If free space is available, volume can consume the same to grow. Method `attributesOfFileSystem` gives the free space for the given path (parameter) along with its other attributes.

- *Sparse Files*

In a non-sparse file system, spaces are reserved for storing files which results in memory wastage. In APFS, sparse filing has been introduced to avoid these blank spaces. A sparse file is automatically created when a new file is created. This might lead to storing of files in out of order fashion, but the performance loss caused is insignificant [9].

- *Encryption*

APFS uses hardware or software encryption depending upon the device's capabilities. Hardware encryption (AES-CBC) is used on iOS and macOS devices' internal storage. Software encryption (AES-XTS) is used for external storage. It is also used for devices that do not support hardware encryption.

Keys for encrypted files are not directly available in the disk. They are stored in a wrapped manner. To decrypt contents of a file, B-Tree file must be decrypted using VEK. Encrypted content on volumes can be accessed using Volume Encrypted Key (VEK). But these keys must be unwrapped using Key Encrypted Key (KEK). User password, Institutional recovery key, iCloud recovery key and personal recovery key are the four methods for unwrapping KEK. Following details are stored for all encryption keys [8]–

```
typedef uint32_t cp_key_class_t;
typedef uint32_t cp_key_os_version_t;
typedef uint16_t cp_key_revision_t;
typedef uint32_t crypto_flags_t;
```

`cp_key_class_t` describes the protection class. `cp_key_os_version_t` and `cp_key_revision_t` describes the OS version & build version and version number for that encryption key respectively. `crypto_flags_t` describes flags that are being used by the encryption state [8].

Every Container has its own keybag called container keybag. Container Keybags store the wrapped VEK and location of volume keybags. Keybags also contain the information that is required to unwrap the keys. Volume keybag stores multiple replicas of the wrapped KEK. Keybags are encrypted using UUID of volume or container. Following enumerations describe the type of information that keybag is withholding [8]–

```
enum {
KB_TAG_UNKNOWN = 0,
KB_TAG_RESERVED_1 = 1,
KB_TAG_VOLUME_KEY = 2,
KB_TAG_VOLUME_UNLOCK_RECORDS = 3,
KB_TAG_VOLUME_PASSPHRASE_HINT = 4,
KB_TAG_WRAPPING_M_KEY = 5,
KB_TAG_VOLUME_M_KEY = 6,
KB_TAG_RESERVED_F8 = 0xF8
};
```

If a keybag has unknown tag, then it must be raised a bug to Apple as this tag has not been reserved by Apple.

D. Limitations of APFS

Since the data is stored in a fragmented manner, more seeks are performed. This degrades performance in HDDs, but not in SDDs. Although APFS performs integrity checks but those are applied on for metadata not user data. APFS does not support backward compatibility. Earlier, APFS did not support fusion drives. Since 2018, macOS Big Sur release incorporated support for fusion drives [10].

IV. INTEROPERABILITY GAPS AND EXFAT

APFS and NTFS are not compatible with each other due to difference in their design and architecture. Architecture and components of APFS have been discussed in detailed in Section III. Attributes & components of NTFS are different than that of APFS.

NTFS volume has four major components – Partition Boot Sector File, Master File Table (MFT), series of meta files and data streams that can be mounted on disk to create partitions. PBS file contains the boot information which is used at the time of system startup. MFT contains the records of files and folders that are stored in the file system. Even though both NTFS and APFS use B-Trees, their difference in architecture contributes to the interoperability gaps.

NTFS works in all Windows devices, but it is read-only format in Macintosh devices. APFS can read NTFS formatted files but cannot write on them. Even though APFS offers more reliability and security, it does not offer cross-platform service. In 2006, Microsoft launched exFAT, which is lightweight file system. It has low power requirements and has been optimized for flash memory. This enables to seamlessly copy files between Windows/Mac device to external storage. It has been

S No	Proc Type	FileSize	Round 1 (second)	Round 2 (second)	Round 3 (second)	Average Response Time	Avg Read Throughput (MBps)	Avg Write Throughput (Mbps)	Avg CPU Utilization (%)
1.	M1 Pro	100 MB	0.532	0.535	0.542	0.542	1294.4	884.58	33.46
2.	M1 Pro	1 GB	5.133	5.132	5.138	5.138	1310.44	913.01	33.46
3.	M1 Pro	5 GB	32.904	31.832	32.738	32.738	1114.80	984.49	30.4
4.	M1 Pro	10 GB	86.944	90.224	87.748	87.748	1062.16	1006.13	27.1
5.	Ryzen 7	100 MB	2.114	2.064	2.114	2.097	302.83	196.68	12.2
6.	Ryzen 7	1 GB	23.299	23.567	23.746	23.537	273.03	181.77	13.33
7.	Ryzen 7	5 GB	127.364	172.243	175.598	158.402	294.23	182.55	7.03
8.	Ryzen 7	10 GB	292.264	326.021	351.248	323.178	292.9	175.27	7.9
9.	i9 - 10	100 MB	2.053	2.01	2.034	2.032	286.47	207.01	12.03
10	i9 - 10	1 GB	19.429	19.436	19.949	19.605	291.17	205.63	13.8
11	i9 - 10	5 GB	102.264	101.426	101.526	101.739	295.2	204.35	6.46
12	i9 - 10	10 GB	210.332	212.981	213.36	212.224	296.3	201.404	7.46

Figure 3: Table displaying experiment results considering all factors and metrics using a Full Factorial Design

designed to support interoperability between Windows and Mac devices.

Files formatted in NTFS format carry a heavy overhead. These files' metadata increases rapidly with increasing file size. NTFS is often called 'lazy write system'. Due to heavy overhead, often files on a removable object get corrupted if device is not properly ejected. Hence, exFAT was introduced as a solution to this problem by Microsoft. Cluster bitmaps were introduced to reduce the overhead. This allowed atomic transactions of file metadata. Maximum cluster size was set to 32MB. [13].

V. Performance Analysis Experiment

Experiment has been conducted to draw a performance comparison between APFS and NTFS. To design the experiment, two factor full factorial design has been implemented. This model has been borrowed from book by Raj Jain [12]. Following metrics are being considered for the experiment:

1. Response Time
2. Read Throughput
3. Write Throughput
4. CPU Utilization

Read Throughput =

Total Bytes Read / Total Read Time

Write Throughput =

Total Bytes Written / Total Write Time

Two Factors are being considered:

1. File Size
2. Type of Processor

There are 4 levels in file size –

1. 100 MB
2. 1 GB
3. 5 GB
4. 10 GB

There are 3 levels in type of processors –

1. Apple M1 Pro
2. Ryzen 7 6800H
3. Intel i9 10th Generation

Machines used :

- Apple Macbook Pro
- ROG Strix G17
- Custom PC i9 10th Gen 24GB RTX 3090

In each iteration, three repetitions have been done for all four metrics. Average values for read throughput, write throughput and CPU Utilization have been displayed in the table for simplicity. Values for response time for all three repetitions has been displayed in the table. In the sixth column, average response time has been calculated.

		File Size						
		100 MB	**1 GB**	**5 GB**	**10 GB**	Row Sum	Row Mean	Row Effect
Processor Type	**M1 Pro**	0.536	5.134	32.491	88.305	379.402	31.6165	-62.61905556
	Ryzen 7	2.097	23.537	158.402	323.178	1521.642	126.8035	32.56794444
	i9 - 10	2.032	19.605	101.739	212.224	1006.8	83.9	-10.33555556
	Column Sum	4.665	48.276	292.632	623.707	969.28		
	Column Mean	1.555	16.092	97.544	207.9023333		94.23555556	
	Column Effect	-92.68055556	-78.14355556	3.308444444	113.6667778			

Figure 4: Table showing calculations of Interactions due to factors (Processor Type, File Size) using ANOVA for two factor full factorial design.

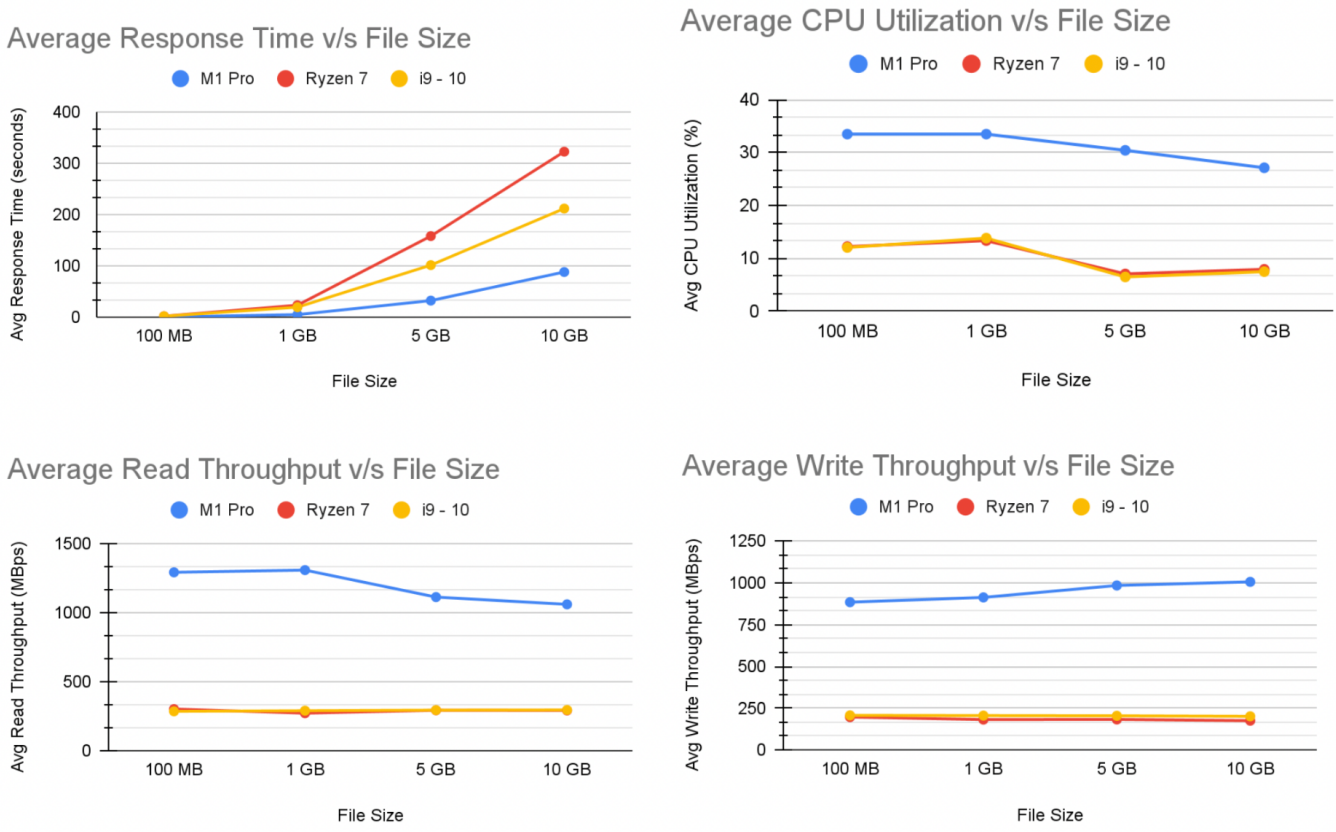

Figure 5: Graph plots showing performance of different types of processors for every response metric.

Refer to Appendix B for Python Script used to conduct the experiment. To calculate the impact of factors, we will be using ANOVA (Allocation of Variance) technique with two factor full factorial technique based on only response time of the program.

Figure 3 displays the readings obtained from the experiment. Values for all response metrics have been populated in the table.

Inferences :

1. Average Response Time :
   M1 Pro < i9 – 10th Gen < Ryzen 7
2. Average Read Throughput :
   M1 Pro > i9 – 10th Gen > Ryzen 7
3. Average Write Throughput :
   M1 Pro > i9 – 10th Gen > Ryzen 7
4. Average CPU Utilization :
   M1 Pro > i9 – 10th Gen > Ryzen 7

To determine which file system performed better, we need to look at file system which has –

- Lower Response Time
- Higher Read Throughput
- Higher Write Throughput
- Lower CPU Utilization

Keeping above mentioned metric values in mind, we can conclude from this experiment that ***APFS performs better than NTFS.*** In Figure 6, Kiviat Chart also shows ***better shape for APFS rather than NTFS.***

Using ANOVA for calculating impact of two factors, namely, file size and type of processor.

- Interaction due to File Size = 53.75 %
- Interaction due to Processor = 13.20 %
- Interaction due to both factors = 12.80 %
- Interaction due to Error = 20.23 %

It can be observed that File Size has greater impact than the type of processor. We can say that File Size has maximum impact on the performance of a file management system.

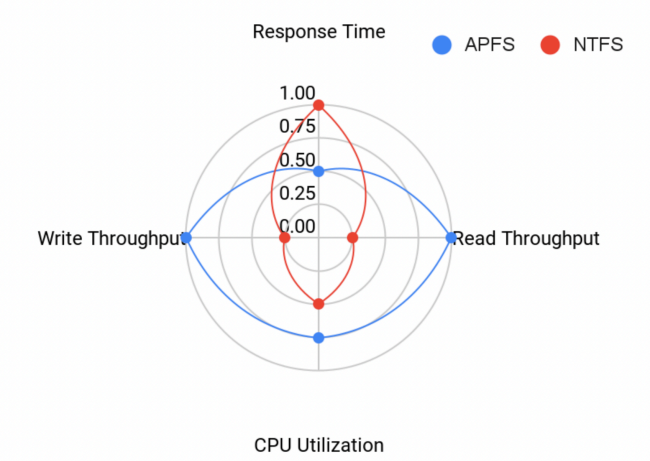

Figure 6: Kiviat Chart for File system. Better shape is denoted by lower response time, higher read throughput, higher write throughput and lower CPU Utilization. APFS has better shape than NTFS.

CONCLUSION

In 2017, Apple Inc launched APFS for Apple devices. In this paper, we discussed about the architecture, components and features of APFS, which make it better than the previously designed file management systems by Apple Inc. APFS makes Containers which partition data into volumes. All files on the system are stored in these volumes. Checkpoints are created to ensure data integrity and to protect file system from crash. Metadata information for files is stored in key-value pairs in a B-Tree. Instead of sequential retrieval, B-Trees are used for fetching data faster.

Cloning, sparse files, encryption, snapshots and space sharing are some of the features introduced in APFS. These features have heavily improved the performance of Apple devices. Encryption is one of the key features that has been implemented at three levels in a hierarchical manner for file security. Every Apple container has a container keybag which stores location of volume keybags and wrapped Volume Encrypted Keys. To decrypt these volume keys, Key Encrypted Keys are used. These keys along with information required for unwrapping them is stored in volume keybags.

Even with such enhanced performance, APS still has few limitations. One of the major limitations of APFS is requirement of Increased number of seeks in HDDs. Another major limitation of APFS is that it performs data integrity checks only for metadata not user data.

Files mounted in APFS format or its predecessor HFS+ can only be read by Windows devices but cannot be written to. In 2006, Microsoft launched exFAT, a lightweight file system. exFAT formatted files are supported by both Mac & Windows devices. Another reason for development of exFAT was to avoid corruption of files formatted in NTFS format (for external storage device). Since these files carry heavy overhead, they have high probability of being corrupted if not ejected properly.

Performance Analysis Experiment shows that *APFS performs better than NTFS.* Response Time, Read Throughput, Write Throughput and CPU Utilization were used as performance metrics**. *Response time for APFS is less than NTFS.*** Two factors were considered, namely, file size and type of processor. Using ANOVA, impact of file size was found 53.75% whereas impact of processor was found 13.20%. Kiviat chart shows that APFS forms a better shape than NTFS.

ACKNOWLEDGMENT

I would like to thank Dr. Sreelakshmi Manjunath for guidance in the performance analysis experiment. I would also like to thank IIT Mandi Catalyst iHub and Sourabh Kundalwal for providing Intel & Ryzen Chip machines to run the experiment.

REFERENCES

Following resources have been cited in this paper -

[1] Apple File Systems – Wikipedia page https://en.wikipedia.org/wiki/Apple_File_System#:~:text=Apple%20File%20System%20(APFS)%20is,and%20deployed%20by%20Apple%20Inc

[2] Hierarchical File Systems – Wikipedia Page https://en.wikipedia.org/wiki/Hierarchical_File_System_(Apple)

[3] Hierarchical File Systems Plus – Wikipedia Page https://en.wikipedia.org/wiki/HFS_Plus

[4] Bradley, J. (2016). *File System. OS X Incident Response, 49–89.* doi:10.1016/b978-0-12-804456-8.00004-2

[5] Su, L. J., Wu, S. X., & Cao, D. (2011). Windows-Based Analysis for HFS+ File System. Advanced Materials Research, 179-180, 538–543. doi:10.4028/www.scientific.net/amr.179-180.538

[6] Kurt H Hansen, Fergus Toolan (2017) – Decoding the Apple File System

[7] Apple File Systems – Wikipedia page

[8] Apple File System Reference Official Document https://developer.apple.com/support/downloads/Apple-File-System-Reference.pdf

[9] Apple Developer Official Documentation https://developer.apple.com/documentation/foundation/file_system/about_apple_file_system

[10] Tamura, Eric; *Giampaolo, Dominic* (2016). *"Introducing Apple File System" (PDF). Retrieved May 28, 2022.*

[11] NTFS Documentation by Richard Russon, Yuval Fiedel (2011) http://dubeyko.com/development/FileSystems/NTFS/ntfsdoc.pdf

[12] Art of Computer Systems Performance Analysis Techniques for Experimental Design Measurements Simulation And Modeling – Raj Jain

[13] exFAT Documentation - https://paradigmsolutions.files.wordpress.com/2009/12/exfat-excerpt-1-4.pdf

Appendix A
Code Snippets

checkpoint_map_phys_t

A Checkpoint mapping block, code snippet from Apple file System Official Reference Documentation [8]

```
struct checkpoint_map_phys {
       obj_phys_t cpm_o;
       uint32_t cpm_flags;
       uint32_t cpm_count;
       checkpoint_mapping_t cpm_map[];
};
```

nx_superblock_t

A Container block, code snippet from Apple file System Official Reference Documentation [8]

```
struct nx_superblock {

       obj_phys_t nx_o;
       uint32_t nx_magic;
       uint32_t nx_block_size;
       uint64_t nx_block_count;
       uint64_t nx_features;
       uint64_t nx_readonly_compatible_features;
       uint64_t nx_incompatible_features;
       uuid_t nx_uuid; oid_t nx_next_oid;
       xid_t nx_next_xid;
       uint32_t nx_xp_desc_blocks;
       uint32_t nx_xp_data_blocks;
       paddr_t nx_xp_desc_base;
       paddr_t nx_xp_data_base;
       uint32_t nx_xp_desc_next;
       uint32_t nx_xp_data_next;
       uint32_t nx_xp_desc_index;
       uint32_t nx_xp_desc_len;
       uint32_t nx_xp_data_index;
       uint32_t nx_xp_data_len;
       oid_t nx_spaceman_oid;
       oid_t nx_omap_oid;
       oid_t nx_reaper_oid;
       uint32_t nx_test_type;
       uint32_t nx_max_file_systems;
       oid_t nx_fs_oid[NX_MAX_FILE_SYSTEMS];
       uint64_t nx_counters[NX_NUM_COUNTERS];
       prange_t nx_blocked_out_prange;
       oid_t nx_evict_mapping_tree_oid;
       uint64_t nx_flags; paddr_t nx_efi_jumpstart;
       uuid_t nx_fusion_uuid; prange_t nx_keylocker;
       uint64_t nx_ephemeral_info[NX_EPH_INFO_COUNT];
       oid_t nx_test_oid;
       oid_t nx_fusion_mt_oid;
       oid_t nx_fusion_wbc_oid;
       prange_t nx_fusion_wbc;
       uint64_t nx_newest_mounted_version;
       prange_t nx_mkb_locker;
};
```

Appendix B

Python Script used for conducting Performance Analysis Experiment

```
"""
Author : Mahima Gupta (T22055)
Date of Start : 01-May-2023
Last Modified Date : 05-May-2023

Script to measure the performance of a file system
-----------------------------------------------------
Operations performed :
1. Creating 50 files
2. Writing to a file
3. Appending a file
4. Reading a file
5. Reading from a random index in a file
6. Renaming 50 files
7. Duplicating 5 files
8. Deleting 50 files

Full Factorial Design -
    File Size = [100MB, 1GB, 5GB]
    Processor = [Silicon Chip, Intel, Ryzen]

Performance Metrics -
    Response time
    Read Throughoput
    Write Throughput
    CPU Utilization

"""

import os
import shutil
import timeit
import psutil
#---------------------------------
# Function to create directory
#---------------------------------

def make_directory():
    path = os.path.join(os.getcwd(), 'Experiment')
    os.makedirs(path)
    #print("Directory created")
    os.chdir('Experiment')

#---------------------------------------
# Function to create multiple files.
#---------------------------------------

def create_file():
    for i in range(0,50):
        filename = "sample" + str(i) + ".txt"
        with open(filename, 'w') as fr:
            pass
        fr.close()

#----------------------------------
# Function to write in a file.
#----------------------------------

lines = ['Hello! This is a demo file. We are writing here to get more content. This file will be used as basic
file for read and write file operations.', 'This will increase the text file size. We will create file size in
such a manner that minimum iterations are required to increase file size.\n']
def write_file(iterations):
    write_start_time = timeit.default_timer()
    with open("sample0.txt", 'w') as fr:
            for i in range(0, iterations):
                fr.writelines(lines)
    write_end_time = timeit.default_timer()
    fr.close()
    fileSize = os.stat("sample0.txt").st_size
    return(write_end_time - write_start_time, fileSize)

#----------------------------------
# Function to append a file.
#----------------------------------

lines = ['These lines are different than the  previous one and have been written to further increase the size of
the file.', 'These lines will be used only for appending text to a file.\n']
def append_file(iterations):
    write_start_time = timeit.default_timer()
```

```
    fileSize = os.stat("sample0.txt").st_size
    with open("sample0.txt", 'a') as fr:
            for i in range(0, iterations):
                fr.writelines(lines)
    write_end_time = timeit.default_timer()
    fr.close()
    fileSize = os.stat("sample0.txt").st_size - fileSize
    return(write_end_time - write_start_time, fileSize)

#---------------------------------
# Function to read a file.
#---------------------------------

def read_file():
    read_start_time = timeit.default_timer()
    with open('sample0.txt', 'r') as fr:
          body = fr.readlines()
    read_end_time = timeit.default_timer()
    fr.close()
    fileSize = os.stat("sample0.txt").st_size
    return(read_end_time - read_start_time, fileSize)

#-------------------------------------------------
# Function to read from a random index in a file.
#-------------------------------------------------

def random_read():
    read_start_time = timeit.default_timer()
    with open('sample0.txt', 'r') as fr:
          fr.seek(5000)
          body = fr.readlines()
    read_end_time = timeit.default_timer()
    fr.close()
    fileSize = os.stat("sample0.txt").st_size - 5000
    return((read_end_time - read_start_time), fileSize)

#---------------------------------
# Function to rename 50 files.
#---------------------------------

def rename_files():
    for i in range(0, 50):
          filename = "sample" + str(i) + ".txt"
          newFilename = "demo" + str(i) + ".txt"
          os.rename(filename, newFilename)
    #print("Files renamed successfully")

#---------------------------------
# Function to duplicate 5 files.
#---------------------------------

def duplicate_files():
     for i in range(1, 6):
          filename = "sample0.txt"
          newFileName = "copied" + str(i) + ".txt"
          shutil.copy(filename, newFileName)

#---------------------------------
# Function to delete 50 files.
#---------------------------------

def delete_files():
    for i in range(0, 50):
          filename = "demo" + str(i) + ".txt"
          os.remove(filename)
    for i in range(1,6):
         filename = "copied" + str(i) + ".txt"
         os.remove(filename)
    #print("Files deleted successfully")

def calculate_throughput(total_read_time, total_MB_read, total_MB_written, total_write_time):
    total_read_time = round(total_read_time, 3)
    total_write_time = round(total_write_time, 3)
    total_MB_read = round(total_MB_read/(10 ** 6), 3)
    total_MB_written = round(total_MB_written/(10 ** 6), 3)
    read_th = round(total_MB_read/ total_read_time, 3)
    write_th = round(total_MB_written/ total_write_time, 3)
    return(read_th, write_th)

#-----------------
# Main Function
#-----------------

if __name__ == "__main__":
```

```
    total_read_time, total_MB_read, total_MB_written, total_write_time, read_throughput, write_throughput = 0,
0, 0, 0, 0, 0
    #fileSizeFactor = [300000, 3000000, 15000000]
    fileSizeFactor = [300000, 3000000, 15000000, 30000000]
    #files = ["100MB", "1GB", "5GB"]
    files = ["100MB", "1GB", "5GB", "10GB"]
    make_directory()
    for  x in range(0, 4):
        print(f"For File Size : {files[x]}")
        for repetitions in range (0, 3):
            start_time = timeit.default_timer()
            create_file()

            write_time, size = write_file(fileSizeFactor[x])
            total_write_time = total_write_time + write_time
            total_MB_written = total_MB_written + size
            write_time, size = append_file(fileSizeFactor[x])
            total_MB_written = total_MB_written + size

            read_time, size = read_file()
            total_read_time = total_read_time + read_time
            total_MB_read = total_MB_read + size
            read_time, size = random_read()
            total_read_time = total_read_time + read_time
            total_MB_read = total_MB_read + size

            duplicate_files()
            rename_files()
            delete_files()
            end_time = timeit.default_timer()
            #cpu = psutil.getloadavg()[1]
            cpu = psutil.cpu_percent()
            execution_time = round((end_time - start_time) * 10 ** 3, 3)
            read_throughput, write_throughput = calculate_throughput(total_read_time, total_MB_read,
total_MB_written, total_write_time)

            print(f"[{execution_time} ms], [{read_throughput} MBps], [{write_throughput} MBps], [{cpu}%]")
```