Aaro Mäkinen

Tracing Android applications for file system optimization

Metropolia University of Applied Sciences

Bachelor of Engineering

Information Technology

Bachelor's Thesis

26 May 2021

Abstract

Author:	Aaro Mäkinen
Title:	Tracing Android applications for file system optimization
Number of Pages:	31 pages + 1 appendix
Date:	26 May 2021
Degree:	Bachelor of Engineering
Degree Programme:	Information technology
Professional Major:	Smart systems
Instructors:	Keijo Länsikunnas, Senior Lecturer Szabolcs Szakacsits, CTO, Tuxera Inc.

The goal of this final year project was to find opportunities to optimize of a file system for the client Tuxera Inc. To accomplish this, storage usage of five Android applications were traced using a software Strace. Tracing was done with rooted Sony Xperia X smart phone running stock Android 10. An additional parser was developed for the produced log so data can be summarized and made human-readable.

Along with existing studies, traced data from the applications illustrated file access patterns and provided information about Android storage stack, and storage usage behavior of applications. Heavy usage of SQLite on Android was found to be a big part of storage I/O. Also, database files, executable files and multimedia files seemed to be often accessed in a defined pattern.

From the gathered data, multiple possible optimization proposals were found for file system allocation. It turned out that kernel caching could lead for better file system performance. This thesis also proposes an external software to generate heuristic data for the file system so that a file system could optimize itself using internal algorithms.

Keywords: file system, Android, SQLite, data storage

Tiivistelmä

Tekijä:	Aaro Mäkinen
Otsikko:	Tracing Android applications for file system optimization
Sivumäärä:	31 sivua + 1 liite
Aika:	26.5.2021
Tutkinto:	Insinööri (AMK)
Tutkinto-ohjelma:	Tieto- ja viestintätekniikka
Ammatillinen pääaine:	Älykkäät järjestelmät
Ohjaajat:	Lehtori Keijo Länsikunnas Szabolcs Szakacsits, teknologiajohtaja, Tuxera Inc.

Insinöörityön päämääränä oli löytää mahdollisuuksia optimoida tiedostojärjestelmä. Tämän saavuttamiseen muutaman Android-applikaation tallennustilan käyttöä jäljitettiin Strace-nimisellä ohjelmalla. Jäljitys suoritettiin Sony Xperia X -älypuhelimella, jossa on vakio Android 10 -käyttöjärjestelmänä ja root-oikeudet annettu käyttäjälle. Tuotetulle logille luotiin lisäksi parserointiohjelma, jotta datan saa muutettua ihmiselle luettavaksi ja yhteenveto mahdollistuu.

Olemassa olevien tutkimuksien ja työssä tehdyn jäljitys datan avulla saatiin selville yleiskuva tiedostojen käytöstä ja tietoa Androidin ja sen applikaatioiden datan tallennustavoista. Huomattiin, että SQLiten runsas käyttö Androidissa oli suuri osa tallennuslevyn käytöstä ja että tietokantaa, suoritettavia tiedostoja ja mediatiedostoja käytetään usein tietyllä tapaa.

Usea mahdollinen optimointi ehdotus saatiin löydettyä tuotetun datan avulla tiedostojärjestelmän allokointiin ja kernelin välimuistiin, jotka voi parantaa tiedostojärjestelmän suorituskykyä. Insinöörityössä myös käytiin läpi ehdotus ulkoisesta ohjelmasta, joka loisi heuristista dataa tiedostojärjestelmälle, jotta se voisi optimoida itseään sisäisillä algoritmeilla.

Avainsanat: tiedostojärjestelmä, Android, SQLite, datan tallennus

Contents

List of Abbreviations

ADB	*Android Debug Bridge*. Debugging interfacce for the Android.
AOSP	*Android Open Source Project*. Pure Android from Google.
APFS	*Apple File System.* File system that is used natively in MacOS.
BTRFS	*B-Tree File System.* Modern copy-on-write file system.
ExFAT	*Extensible File Allocation Table.* File system developed by Microsoft that is used currently mainly for compatibility reasons.
Ext3	*Third Extended File system*. Older file system used mainly in Linux.
Ext4	*Fourth Extended File system.* Newest native file system in Linux.
FTL	*Flash Translation Layer.* Data management for flash memory.
I/O	*Input/Output.* Data transaction between storage and system.
IPC	*Inter-process communication.* Interface between processes.
POSIX	*Portable Operating System Interface.* Standards for operating system compability.
RAM	*Random Access Memory.* Volatile memory that is used in computing and storing temporary data.
SLC	*Single-Level Cell.* Type of flash memory cell that only holds one bit per cell.
WAL	*Write-Ahead Log.* Operating mode for SQLite database.
ZFS	*Zettabyte File System.* Modern copy-on-write file system.

1 Introduction

Storage performance is an easily overlooked factor for the system performance in modern days. Studies suggest that, since internet and computing speeds have increased drastically over the years, storage speed is a significant factor in system and application speeds [1; 2]. Studies find storage being also significant factor on CPU and energy consumption with about 36 % of total energy being consumed by storage stack on Android mobile phone [1; 3].

The goal for this thesis is to come up with possible opportunities to optimize file system performance. To accomplish this, Strace software was used to trace system calls related to file system operation on five common Android applications to observe file access patterns and gain ideas from. Additional parser software was also developed to make file access patterns easier to be observed.

The client company for this thesis is Tuxera Inc. that specializes in embedded and storage software solutions; it is also the workplace for the author [4]. The topic for the thesis was inspired by news of rampage storage usage of popular brands like Tesla having to recall over 150 thousand vehicles because of failing flash memory chips [5]. All hardware used in this thesis was provided by the client.

Android was chosen to be the platform for the tracing since its massive growth in recent years and spread to automotive industry. Interest for Android is also due to prior knowledge of Androids extensive use of SQLite database usage and studies showing Android having high storage usage and fragmentation on its files [6; 1].

2 Theory

This chapter introduces the main topics discussed in this thesis. First the file system and its internals are explained with different approaches and designs of

implementing it. Second, the Android, its layered architecture and storage scheme are covered. Lastly, SQLite and databases are introduced, and different operating modes of SQLite are explained.

2.1 File system

In the early days of computing, there was only volatile memory to store any type of data and every time the computer was shut down, all data was lost. Since then, there has been innovations of more permanent data storing solutions in form of tape disks, optical disks, hard disks, and the most recent technology, flash memory. Without any system to keep book of data or any additional management, disk space is just string of data where every user must know where the data begins and when it ends in which offset of the whole storage device. Today there are variety of different file systems and mostly all have the same purpose, to store and retrieve the data on storage on logical groups that are called files.

2.1.1 Hierarchical file system

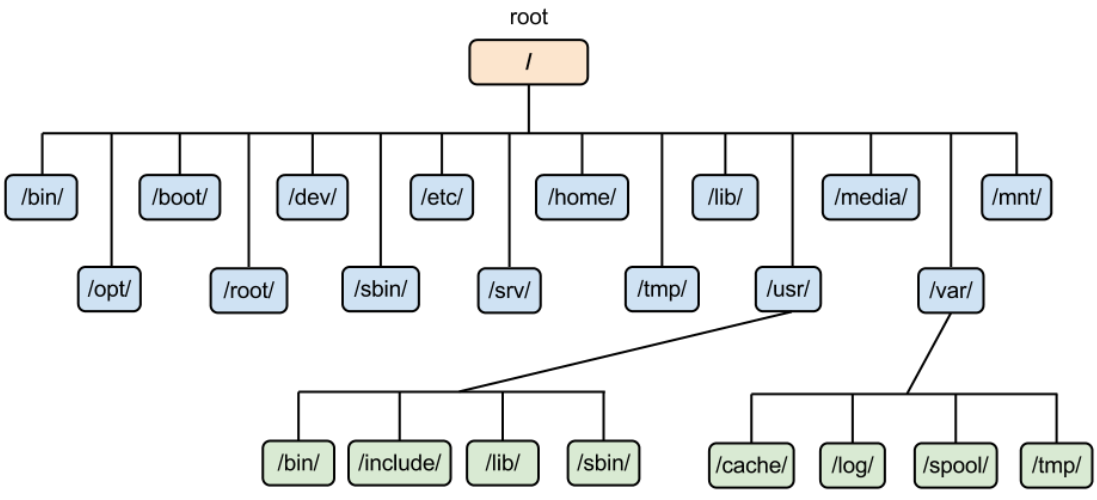

Figure 1. Example of Linux directory tree [7].

Files alone are not enough to make file system easily usable if they are all in the same place. It would be like a library where every book is scattered on the same

shelf and searching the book that is needed can become tedious, therefor they are indexed with categories. Same idea applies in file systems with the use of directories, or folders in windows-based machines. Files are grouped in directories and again in sub directories to produce hierarchy for the data. The most top-one directory is called the root directory, where from all the other files and directories branch out. This hierarchy can be visualized by imaging a tree trunk where from all the branches grow from. Figure 1 illustrates an example of the directory tree on Linux.

2.1.2 Concepts

File systems have many concepts to define its operation and to talk about file systems, some concepts need to be defined.

- Volume: is a file system instance in a partition or storage device.

- Formatting: creation of file system volume by using formatting tool that initializes the file system and creates root directory. Different file systems have different formatting tools that are usually packed with the operating system itself. In Linux, usually Mkfs programs are used from command-line and in Windows, graphical user interface based (GUI) software.

- Partition: region of disk that can be the length of the whole disk or a part of it. Partitioning is not necessary to have file system formatted on disk but is typical to have disk partitioned to at least to contain one partition before formatting a volume.

- Mounting: set up access for operating system for the storage device by either by user-initiated command or by system itself. Mounting involves reading the file system structures from storage and checking the file system state and corruptions as well as possibly repairing them. Unmounting is the reverse of mounting where disk is prepared to be detached from the operating system, so all pending operations are

completed, and new operations are not permitted. This helps the file system to maintain its integrity and keep from corrupting whole or singular files.

- File system block: fourth extended file system (ext4), File allocation table (FAT)-based file systems and b-tree file system (BTRFS) are so called block-based file systems that slice the volume in blocks to store and help retrieval of data. Superblock is a special block that holds the metadata for the whole file system volume and the underlying storage device that is read during mounting from a defined place on a file system. Data is being kept in configurable sized blocks, and their addresses are stored in upper-level blocks that hold only pointers to data blocks or in I-node.

- I-node: Structure defined in kernel to hold all metadata of a file or directory. I-node contain pointers to blocks addresses that hold the data of a file.

- Metadata: Other than actual data that is in file, usually there are information that file system keeps track for each file that is called metadata. Different file systems keep different kind of metadata and in different ways and places in the storage space but usually at least file size, filename, directory containing the file, access and modification time and other attributes.

2.1.3 Different file systems

For an operating system to be able to use a file system, there needs to be a file system-specific driver installed in it. The driver is the file system software that comes with the operating system and every operating system has its own file system. These can be also later installed but some file systems are proprietary and can only be used in the operating system that it is native to or at least restrict the usability to read-only mode, which e.g., Linux has for the Windows developed NTFS.

Usually there are more than one file system supported by the Operating system and these are mainly used for different storage devices or for other different kind of use cases. Windows for example natively uses ExFAT for their portable media and NTFS for non-portable partitions. Linux uses natively open source developed ext4 and MacOS uses its own developed Apple file system (APFS).

File systems differ from each other mainly on how data is stored in storage and how they keep the file system from corrupting. Earlier file systems also suffered from lower file, volume and filename size limitations that differentiated them, but mainly modern file systems have big enough metadata sizes, so this rarely is an issue anymore. Comparison of the file systems are shown in table 1.

Table 1. Comparison of file systems

File system	**Maximum file size**	**Maximum volume size**	**Journaling or COW**
Ext4	16 GB – 16 TiB*	1 EiB	Journaling
NTFS	16 EiB	16 EiB	Journaling
ExFAT	16 GiB	64 ZiB	-
FAT32	4 GiB	512 MiB – 16 TiB*	-
APFS	8 EiB	?	COW
ZFS	16 EiB	2^{128} bytes	COW
BTRFS	16 EiB	16 EiB	COW

*Depending on internal block-size

Maintaining data integrity after a power failure is usually a part of modern file systems responsibilities. There are multiple ways for file systems to get corrupted after underlying storage device gets plugged out, lose power or if the operating system failure occurs. Modern storage devices usually have 512-byte atomic sector write, that means that 512 bytes are guaranteed to either get written as a whole or not at all in case of power failure. Corruption usually happens when

power failure happens between these sectors. One of these corruptions is something called cross-linked file where multiple files have allocations for the same storage data block. Primarily two methods are used to mitigate this fail safety problem, journaling and copy on write.

Journaling

Most popular data integrity method used is journaling as it is used in Linux's ext3 and ext4 and Windows' NTFS, which are the most used platforms with Windows having over 75 % market share on PC and Linux 71 % on mobile through Android [8]. In journaling file systems changes are logged in order so events can be reconstructed in case of failure. Mounting a volume checks the existing file system state or "snapshot" and the journal for additional operations that may have been uncommitted. File system then can redo or undo the changes depending on the mode and journaled data and continue operating normally.

Linux's ext4 and ext3 have three levels of journaling where the user can choose it at mount time, which are explained below [9].

- Journal: mode writes all metadata changes e.g., new filename or new file size, and data that was written, modified or removed to journal before committing anything to main file system. File system recovery from this level is most reliable with this level but there is more write overhead since every single write must be done twice, once for journal and once for the main file system.

- Ordered: mode first commits metadata to journal then the actual data is written to the main file system and lastly the metadata is written from the journal to the main file system. This ordering of the operations ensures the file system consistency, but the file might still be corrupted if there is an interrupt between these three steps. When it comes to performance, this is between the other modes and overall it can be seen as the middle

road for the performance and data recovery, so it is logical it is the default option of the three.

- Writeback: mode behaves similarly to ordered mode where only metadata gets journaled, and data is written straight to the main file system. The difference to the ordered mode is that there is not defined order of operations but instead whatever order kernel decides to get best performance. Writeback approach offers the best performance but lacks behind on data recovery with the file system keeping itself from corrupting. However, files getting more easily corrupted.

Copy-on-write

Unlike journaling, copy-on-write does not record its transactions but rather makes copies of data; it does not modify the original block that holds the data and metadata. When data is written with a copy-on-write method, all blocks that would be affected by the modification from data blocks to the i-node are copied to newly allocated blocks. Modifications are then done to the copied user data blocks and intermediate blocks, containing pointers referring to the lower-level block, are updated from bottom to top. This way if at any time a failure happens during this operation the original data still exists, and original data structures are pointing to the original data. Final transaction is to update the i-node that is also copied and modified before and so the new data is committed to file system. Some file systems that use copy-on-write method are ZFS, BTRFS and APFS. Figure 2 illustrates copy-on-write mechanism of making copies of blocks and hierarchical block structure.

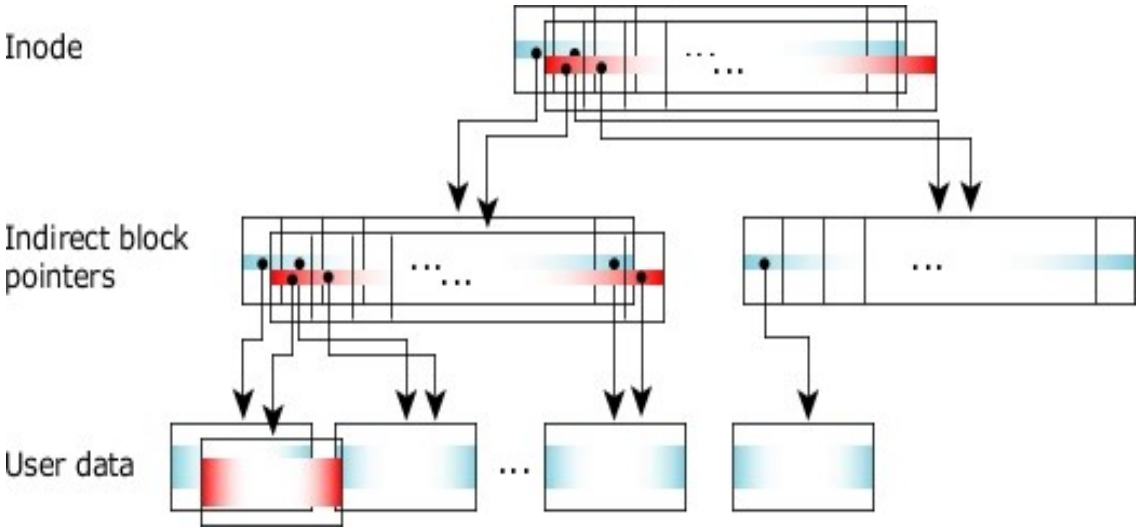

Figure 2. Illustration of block copying of copy-on-write. [10]

2.1.4 Caching

Linux and its native ext-family of file systems use kernel provided caching method called page caching to optimize read and write operations. Page cache is dynamically growing and shrinking buffer consisting pages of random-access memory (RAM) that are read from a disk or pending write to it.

When data is read from a file, first page cache is checked if it contains the requested page and if not, it is only then read from the disk. Reading from RAM is much faster, than reading from the disk, with RAM having around 100 ns read time and approximately 50 µs on flash [12]. Since the RAM is much faster it is beneficial to keep the read page in memory for possible future reads, so it will not have to be fetched again from the disk. Linux uses optimization technique called read-ahead, which as its name implies, reads pages of files and puts them in page cache in advance of process actually reading them if the file is read sequentially. This decreases the time that process must wait for the read operations.

Page cache is also used to hold pages that the file system has marked to be written but are not actually on the disk yet. These pending pages are called dirty pages on Linux. Caching writes enables writes to be more packed so not every write alone is written to disk since latency with 2 I/Os of the same write consumes more time than if the same data were written with only one I/O operation. Linux calls this buffering write back. [11, p. 599-630.]

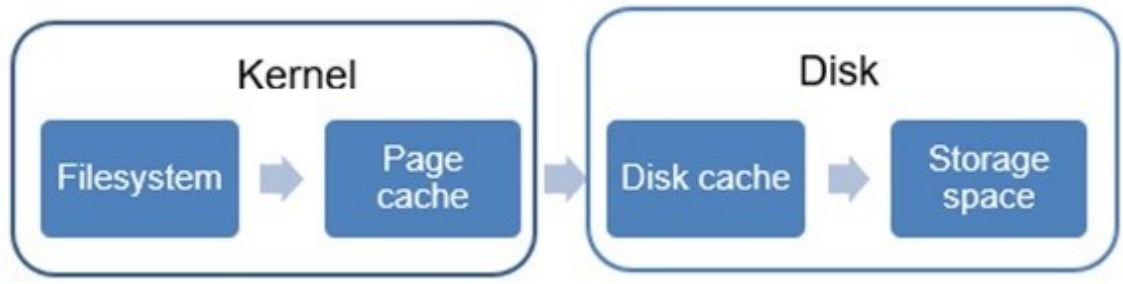

Figure 3. Layers for writing data.

Kernel flushes page cache by writing dirty pages to storage and/or removing read pages if one of three things occur.

- Dirty page gets too old and below the configured threshold.
- Free memory reaches a point below the minimum allowable size.
- User-space process demands the flush of some pages or all of them.

Modern HDD and flash disks contain their own caching mechanism with volatile memory, and it is often called disk cache or disk buffer. Disk caching is used to optimize reads and writes the same way as kernel page cache does it and it also can reorder writes itself. Since the disk tells the kernel that write has occurred although it can still be only inside of the disk cache, some disks use battery to ensure that all remaining writes are pushed from the cache to the actual disk space so no data will be lost. Kernel cannot know if the disk has battery to ensure the integrity of the data so when a flush of page cache is initiated by e.g., file system updating its journal, also the disk cache is flushed by the file system [13].

Disk flushing also enables file systems to guarantee the order of the writes and it is called barriering in ext3 and ext4. Figure 3 illustrates the layers that the write data goes through.

2.2 Android

Android is an operating system designed primarily for mobile devices and is built on top of modified version of Linux kernel in open-source manner, that is led by Google [14]. Android is the most widely used operating system in mobile platforms with over 71 % market share with estimated 1.6 billion users worldwide [8; 15]. In addition of the mobile platform, Android is extending its influence in automotive industry with its automotive extension of Android powering the car infotainment system [16].

2.2.1 Architecture

Android stack consists of layers that are illustrated in figure 4. In addition to hardware drivers, manufacturers usually put their own software on top of the stock Android open-source project (AOSP) in form of a launcher or apps that can replace the core native apps, e.g., calendar or clock.

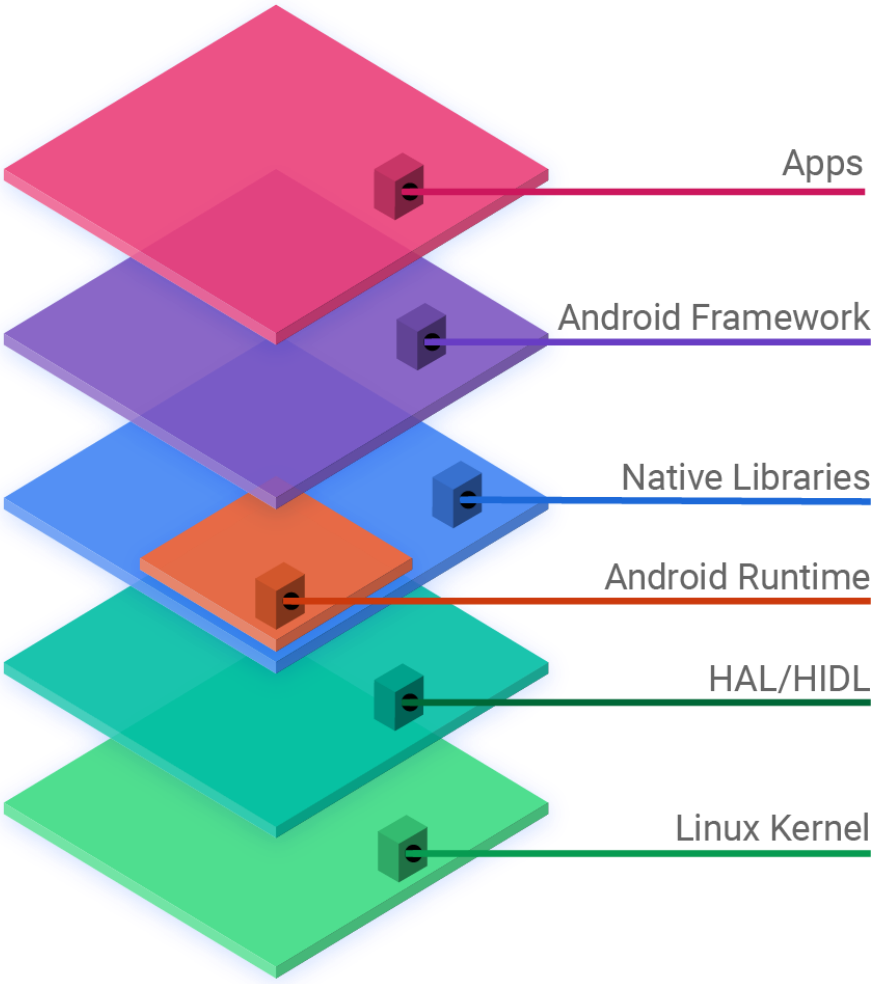

Figure 4. Illustration of Android architecture [14].

Hardware abstraction layer (HAL): bridges the gap for hardware and software by standardized interface between them with HAL interface definition language (HIDL). HAL helps to isolate hardware drivers and their implementations from upper layers. [17.]

Android runtime (ART) replaced the old Dalvik virtual machine implementations in Android 5 and is used to isolate applications by having each run in a separate virtual machine. The use of virtual machines is used to enhance security between the applications and enable access to only defined areas of Android. [17.]

Native C and C++ libraries for Android are provided to developers through Android framework layer and used also by HAL and ART. Framework layer

provides Java language application programming interface (API) for apps to access Android's features. Native and third party apps that are the outer core of Android are written in Java and packaged to Android application package file (APK). [17.]

2.2.2 Android storage

Table 2. Android storage partitions.

Partition	**Function**
boot	Contains Linux kernel and ramdisk. Ramdisk contains initialization files and configurations. Partition is used to boot the device by writing kernel and ramdisk to RAM.
system	Contains Android framework and together with boot partition, make the operating system.
recovery	Backup boot partition.
cache	Used to store temporary and frequently used data from the applications.
misc	Miscellaneous system settings for recovery partition to use.
userdata	All user data from downloads to applications and their data.

Android storage configuration has evolved greatly over time, but the core idea has always been there with partitioning the internal flash that table 2 illustrates. All partitions are formatted with ext4 file system. More optional partitions have been created in recent versions to improve modularity and over-the-air updates from manufacturers e.g., /vendor and /odm which are used for hardware integration and proprietary software. [18.]

Android also support external storage like SD cards or USB storage devices. It is used mainly for application data and multimedia files that are not connected to the Android system itself and can be removed. Since Android 6.0, adoptable

storage feature was introduced to use external storage as if it were internal. Adopting the external storage significantly increases the maximum storage allowed by Android but cannot be considered portable anymore, since it is tied to the system and removal can cause Android to not work anymore. [19.]

2.3 SQLite

SQLite is an open-source relational database management library that is the most used database in the world with deployments including Android, Mac, iOS devices and Web browsers Firefox, Opera and Google Chrome. Estimations say that billions of devices use SQLite and over a trillion databases are actively used since its initial release in year 2000. [20.]

2.3.1 Database

Database is data collection that is easily accessed, managed and updated. Database is usually controlled by database management system (DBMS) software like SQLite to provide interface with user and the database. From the 1960s' databases have evolved with different kind of approaches to modelling the database architecture and three most dominant being:

- Navigational: Earliest approach with linked list or tree like approach for managing data.

- Relational: Data is organized in tables and keys are used to fetch the information in the table cells with additional benefit for having rows and columns to fetch related grouping of data. SQLite falls into this category.

- NoSQL: Uses XML formatted files for its data files with faster approach to managing the data. Unlike relational databases, NoSQL allows data to be nonstructured so it can be used with XML files from different sources and structures.

Relational databases usually use Structured query language (SQL) to store, retrieve and manipulate data for the database. SQL is standardized by the International Organization for Standardization (ISO), but many database systems deviate from it or extend the functionality. This creates compatibility problems between different database systems since same SQL code possibly cannot be ported to different database systems without modifying it.

2.3.2 Features

Usual approach for DBMS is to have it running in a dedicated server process that the process interfaces with interprocess communication methods. Main differences of SQLite comparing to mainly all other databases is that it does not need dedicated server to run and to be configured. Rather very simplistic approach is used to have database written in a single file and possible temporary files that can be transported to completely different environment or platform and it works there.

SQLite database file being portable single file with easy-to-use interface, excel being used as local data storage for isolated applications and devices so embedded or internet of things (IOT) devices are a good fit for SQLite. Other usages include website data analysis, file and data container. SQLite files can also be used as application file format that means that applications can open SQLite database file to view and modify data.

SQLite offers native C/C++ API to interface with the database in code. Other languages like python or java have integrated the interface within their language as a package to be able to use the functionalities od SQLite. SQLite also offers a command-line tool named Sqlite3 that is developed and provided by the developers of the SQLite project to allow users to modify and view SQLite databases. Sqlite3 is available for Mac, Windows and Linux. Tool is used with the SQL language.

2.3.3 Modes

SQLite is serializable transactional, that effectively means that database is not corrupted by a program or system crash or power-loss. Two different modes are implemented for SQLite to achieve this with both using journaling like a file system: rollback and write-ahead-logging, where rollback is the default mode.

Rollback

In rollback mode to modify a database the original unmodified data with journal header is written to a new temporary file which is named with the original database filename and “-journal” appended to the end. Journal file is then flushed to the underlying storage device from the page cache to make sure it exists on disk. After journal is ensured to exist with the original data, the modifications in user-space can be done and written to disk but needs to be also flushed to ensure data integrity for the possible deletion of journal. In case of crash or a power failure, the database integrity is achieved either using the unmodified data still in storage or rolling back the journaled unmodified data, depending on the state of operation when the interruption occurred. If interruption happens before the database file is modified on storage, operation continues using it. If the interruption happens after the changes to the database are partially written, the journal file with original data and a journal header can be played back and original data can be written to the database again.

There are five options to handle the temporary journal file created with rollback journaling after the changes are committed to the main database file:

- Delete: deletes the journal file in cache and disk. (Default).
- Truncate: shrinks the journal file to be zero-length but does not delete the file.
- Persist: header of the journal file is overwritten with zeroes.

- Memory: journaling is done in RAM only and so it does not help preventing database corruption in case of whole system failing or power-loss.

- Off: no journaling is done using this mode, so even if only the program using SQLite crashes, database is not protected from corrupting.

Write-ahead-logging (WAL)

WAL mode was developed most recently to reduce the flushing of the cache and to improve performance of operation. Difference between WAL and rollback journaling is that in WAL mode writes are done to a temporary file which is named by appending “-wal” to main database file. Data is flushed from the page cache to the temporary file to ensure WAL file integrity, but main database file is not modified. This indicates only one write and flush operation per modification while rollback had to write and flush twice per operation.

WAL journal file is then primarily accessed to read from the database and if it does not contain the pages needed, only then the actual database file is read. The journal file can be read concurrently by multiple processes so each of them have an end mark marked into the file to indicate where in file they have left so in case of other process writing to the file, data will seem to be in a single state. Special file living in shared-memory appended with “-shm” is used to index the pages for the WAL file, so readers do not have to traverse the whole time, since it can grow too big to be efficient.

Checkpointing is a term used in SQLite to indicate writing the temporary WAL file to the actual database file. Checkpointing happens by default when configurable page threshold is reached by some writer process or by manually running it. Checkpointing must take end marks in WAL in consideration to ensure it will not overwrite anything that some reader is currently reading. Checkpoint operation will stop at the end-mark and when the end mark is moved by the reader, checkpointing is continued with following write operation taking place. Pages that

are written to main database file are flushed from the page cache to ensure that disk contains the written pages so WAL file can be written to the beginning of the file and operation can continue. Last process to close connection to the database indicates the deletion of the WAL file, but in case of process not exiting cleanly it will remain in file system to ensure integrity of the database.

3 Application tracing

There are many block-level research papers about the file system activity on Android like 2012 published papers from Kim et.al [1] and Lee et.al [21]. These papers look at the actual disk I/O from the block driver. This thesis concentrates on the actual system calls that the application produces when it is in use so the actual writes, reads and other file manipulations can be seen as they are and not after the file system or the block driver.

3.1 Tracing setup

Table 3. Traced applications and their use cases.

Application	**Usage**
Google Play	1. Browse and search 5 applications. 2. Download Clash of clans
Facebook	1. Browse the feed. 2. Post status. 3. Search people. 4. View pictures and videos.
Firefox	1. Browse several web pages. 2. Download 200 MiB file from “https://www.thinkbroadband.com/download”.
Gallery	1. View thumbnails of photos and videos. 2. View pictures and videos.
Clash of clans	1. Play the installed game.

Device selected for tracing is a smartphone Sony Xperia X running Android version. Mobile phone has three GB of RAM and its internal flash storage is 32 GB. Device is rooted and the Android is self-built stock AOSP using hardware

drivers provided by Sony. Google apps (GAPPS) micro version has been installed externally to it to gain access to play store and ADB (Android debug bridge) command line tool is used to communicate with the phone from pc. Android partitions traced are /storage, /data and /system since they are used by applications and files that applications can only access are kept in traced partitions. Applications were traced for five minutes each and they are listed with their usage in table 3.

3.2 Strace

Software used to trace applications in this thesis is Strace that is included in AOSP build version of Linux. Strace is a diagnostic software used to monitor and record Linux kernel system calls and signals from a single process and with an optional parameter, also its child processes. Strace is attached to a process either by using parameter to give the process ID (PID) that is wanted to be traced or executable command. [22.]

In Android applications are not started by simply executing the binary but special software is needed to run the application. Android developed command-line tool Monkey is used to start the application that sends an event call with Androids inter-process communication (IPC) methods. Ordinary command or program call generates the main process for the Strace to attach which additional child processes are forked. Using Monkey to start the application this does not happen so after executing the Monkey command, applications' PID is grepped and passed to Strace in pipeline to get Strace attached as soon as possible. Trace data generated from Strace then saved to a log file that can be parsed later.

3.3 Parser

In Strace there is a parameter (-e) to use qualifying expression to get only desired system calls to be outputted [22]. This was used to include only system calls that could have effect on file systems and what might be useful for optimization. Complete list of the system calls traced are listed in the appendix 1. The output

for all applications is physically too large for human comprehension with over 500 000 lines. Small parsing program was written to be able to summarize and see any optimization possibilities.

The parser is a C++ program that traverses the generated log file line by line and puts all system calls related to a specific file to a new file named so file access patterns can be seen. Parser program was also needed since significant amount of the system calls are output as unfinished since between the actual system call starting and system call finishing, there is other process producing some call to the log. Strace also produces log line that the call is resumed after the original call was done and from that we only can see e.g., how much actually was read from a file from a read system call. Parser is needed since these, unfinished and resumed can be multiple lines away and only resuming system call with the same PID can be the correct one. Additional scripts were made to summarize gathered data.

3.4 Results

Table 4. Gathered write and read totals of applications.

Application	**Total writes**	**Total reads**	**SQLite writes (%)**	**SQLite reads (%)**
Clash of clans	42 MiB	311 KiB	-	-
Facebook	20 MiB	8 MiB	30	20
Firefox	48 MiB*	77 MiB	36	12
Gallery	339 KiB	29 MiB	-	-
Google Play	15 MiB*	16 MiB*	72	28

*excluding the downloaded file

Applications generated different amount of disk usage that can be seen in table 1. Clash of clans did not use SQLite database at all, but it had a lot of resource files written with almost all its writes contributing to sequential resource file writes.

It is unknown if these files are read ever, but in our testing, these were only appended. Gallery reverses this not surprisingly with only sequentially reading multimedia files that were looked at and played.

Firefox was the heaviest disk user even without including the downloaded 200 MiB file and a lot of it can be contributed to SQLite and its temporary files. Facebook was almost the same but with big difference that there was not flushing of the file data at-all. In Google play writing of the application was mostly done by IPC and so it could not be seen with Strace. Interesting thing was that downloaded application file was sequentially read which generated over 160 MB of file read. Trace data is presented in table 4.

3.5 Limitations

Using monkey will miss some system calls from the beginning since the latency after an application is started and Strace command executing is approximately 0,28 s. In this time there can be and probably is file system related system calls that are not saved to log. Missing system calls do not impact the integrity of the traced data substantially since main interest is not to get definitive analysis of the file system usage of the application, but rather get an idea how applications access files, so possible optimization proposals can be generated.

Strace can trace all system calls that kernel has but additional software was needed to be developed in order to gain useful heuristic data out of it. Mmap system call is used to allocate a new page of memory and occupying it with file data when used for file reading. All other system-calls' data was able to be gathered but, mmap only showed creation of the page but not traced further what was done with the page. Some file manipulations might have been missed because of this limitation of Strace.

Strace does not offer any block-level tracing that could have provided more information about actual I/O between kernel and disk. Kernel had an ability to trace block-level operations with included method of event tracing. This approach

was tried, but simultaneous use of both tracing software resulted unusable environment for testing due to hardware limitations that made the smart phone extremely slow.

Strace is only able to trace process and child processes created by the main process. Data can be relayed to complete other process hierarchies using IPC methods like pipes or shared resource. Android also has its own IPC methods called binders and intents that is heavily used and so it may have prevented some file system manipulation to be seen by Strace [23; 24]. Downloading a game in chapter 3.4 illustrates this problem with writes from the actual game data installed to the disk with Google play process that were unable to be logged.

4 Optimization

Previous studies have shown that there is a room for optimization for Android storage and its file system usage [1, 21, 25]. These studies are not necessarily bound to only Android since the same software, such as file systems and SQLite are used elsewhere also. The same applies for the improvement proposals that are talked about in this chapter.

4.1 Fragmentation

When files over time get scattered to different pieces across the disk it is called fragmentation, which is caused by file system aging, subpar allocation strategies or storage space limitations. Figure 5 illustrates the fragmentation on disk. Additional seeks are required for reading the files which causes performance to decline since every seek takes time. Older hard drive disks (HDD) suffered greatly from fragmentation since there is a physical needle that had to move in seek operation where even 50x slowdown can be experienced due to fragmentation. Flash memory access the data same way as HDD with blocks so the seek is used practically moving between the blocks containing the required data. In Flash memory performance degradation due to fragmentation is much

less significant since flash does not contain any mechanical moving parts and only experiences 2-5x performance deterioration due to fragmentation [25].

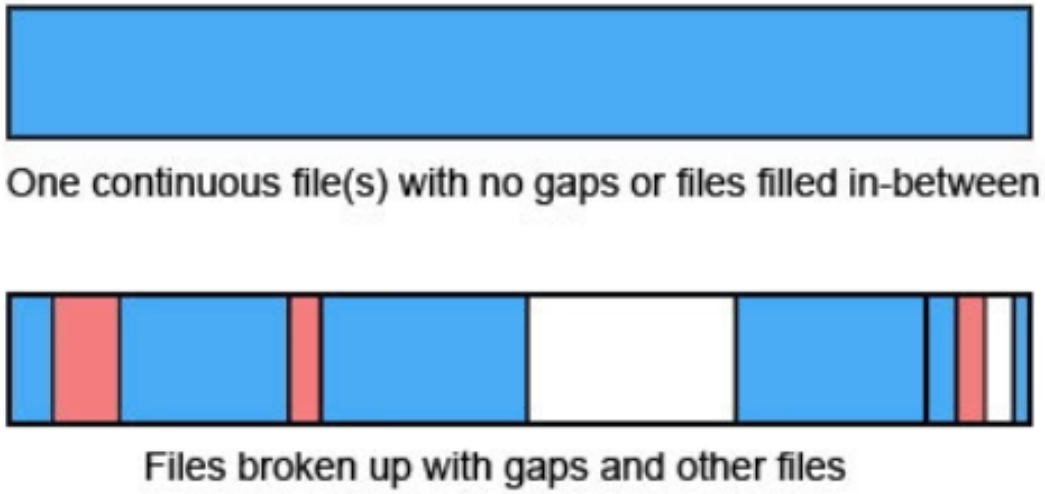

Figure 5. Illustration of file fragmentation [26].

Performance with fragmentation on HDD is magnified by the fact the files are usually physically fragmented in the disk if they are also logically fragmented. Flash memory rarely has its files physically fragmented since flash memory controller (FTL) usually inside of the flash device controls where in the storage the data ultimately will be placed. FTL also reorders the data inside the flash and tries to reduce wear by using all the parts of the storage. On flash, FTL is also responsible for the disk cache that was illustrated in figure 3

Fragmentation performance on flash memory is mainly due to latency from mapping of the physical and logical blocks and latency from the block level drivers caused by increased I/O requests due to fragmentation. [2.]. In this thesis whenever the fragmentation and disk layout are discussed, the logical block mapping is meant.

Defragmenting the file system by reordering the file fragments to be in sequential order is usually performed periodically to HDDs to improve performance, but for flash devices and their limited lifetime, defragmenting can even be considered

harmful [6]. Although fragmentation is not so big performance factor in flash, fragmentation is still widely studied and new defragmentation tools for flash devices are being developed [2].

4.2 Allocation

File systems file allocation is a major factor for the file fragmentation problem. Method of delayed allocation was invented to address this problem, where allocation only happens after the writing is done all the way or pages are flushed to memory. With delayed allocation, allocator knows how much allocation is needed before hand since writes are only written to memory first but there is always the problem of further modifications of the file that the allocator cannot know and limitations of the disk space.

Due to file system aging unallocated gaps are often left like seen in figure 5. These gaps can come from deleting a file or part of it or if there is intentionally left space in order to have sequential space for file appending. If free space is limited in the volume or allocation strategy is not optimal, these gaps are used for new file allocation. This can lead to fragmentation if the original file creating the front of the gap or the file filling the gap is appended so that the next allocation extent would have to go not sequentially to somewhere else in disk.

4.2.1 Sequential and random file allocation

Fragmentation performance drop affects more sequential access of files since then, even though the access pattern would be sequential, the data must be retrieved from different offsets from the storage. Random access pattern must do it whether the file is sequentially allocated or not and it does not get the benefit from read ahead from the page cache. With sequential I/O covering 80 % of all I/O operations, it is very relevant to be considered [21]. With knowledge of the access pattern, storage allocation could be optimized to allocate often randomly accessed or only randomly accessed files to these gaps. Conversely, often sequentially accessed and only sequentially accessed files could be emphasized

to be sequentially allocated. Especially when the storage space is low, it would be beneficial to know if new allocations should be sequential or should the allocator reserve the sequential space when filling the gaps.

4.2.2 Separate regions

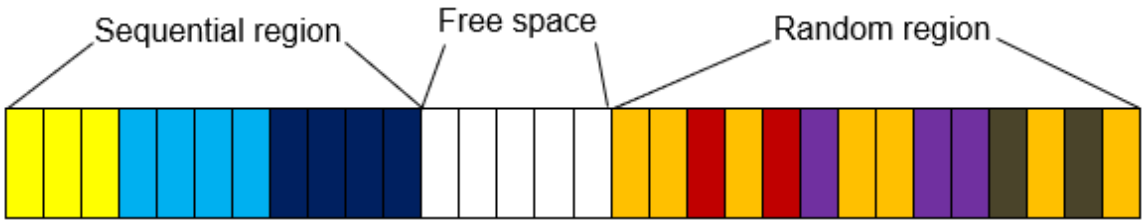

Figure 6. Illustration of allocation on separate logical regions.

Allocation could also use the knowledge of file access patterns to reserve some space in the volume dedicated to randomly accessed files or to have these file allocations on the other end of the volume from sequentially read files. File system could have the offsets for the areas in memory so both areas could grow dynamically to middle but kept from intervening until there is no space left on the disk. Random access area would not need to care that much if the files are fragmented but the size of the area matters as more widely dispersed random writes and reads reduce performance [6]. Separation would help keeping sequentially read files allocated more sequentially since some files are out of the way in tightly packed region across the disk and more sequential space is free. Figure 6 illustrates the idea of separate regions on opposite sides of the volume.

4.3 Page cache

It would be beneficial for page cache to have pages removed from it if files that are only read or written and not accessed again in some time or at all. Page cache would then have more room for more important pages that could be accessed more frequently, increasing the speed of the system. On the contrary, if files are known to be accessed all the time like some SQLite files it would be beneficial to keep these pages in the cache so there is no need to get the pages again from

the storage. File system could remove or keep the pages in the cache with the heuristic information of files that are accessed either way. Concreate example can be found how Firefox reads sequentially little under 7 Mobs from its “goog-phish-proto.vlpset” file and within the same second reads it again from the same offset. If the pages are kept in cache, the second read would be read faster from there.

Utilizing file access flag

For page caching in Linux there is existing portable operating system interface (POSIX) standard that defines a method to predeclare access pattern for the file in defined region. The function is called posix_fadvice and it contains six possible flags where specified data can be said to be accessed: normally, sequentially, randomly, only once, in near future and not at all. In Linux systems, this is implemented from version 2.5.60 onwards with system call fadvise64 to modify the readahead and its subsequent caching behavior [27].

For a file system, allocator fadvise pre-declaration could be used to find out if file is being accessed sequentially or randomly and allocate accordingly. Structure that contains this flag is readily available for file systems to get from kernel so it would be easily accessed method for file systems to optimize their allocation strategy e.g., like described in chapter 4.2 and its subchapters. In this thesis, no file system was found to be utilizing this information and from the traced applications, only Firefox was found to be using fadvise.

4.4 Sync

Manual flushing of the pages from the page cache is called syncing that is abbreviation of synchronization since after the flush operation, the memory and disk are synchronized to not have different data in them. System call sync is used to initiate flushing of all the dirty pages in the page cache, fsync is used to synchronize only file specific data and metadata and fdatasync is only for the data without the metadata if it is not needed for actual data retrieval like modification

or access time [28]. All three sync system call variants have POSIX standard definitions of operation that Linux kernel and its file systems obey in order to be POSIX-compliant. SQLite relies on this in order to keep its database integrity.

SQLite database maintains its integrity by flushing its journal and database writes after all commits like described in chapter 2.3.3. This access pattern exists always on these SQLite files with journaling mode having two separate syncs and Wal mode one. Writeback of these pages can result multiple additional syncs if kernel initiates flushing of the pages before they are needed from SQLite's perspective. Kernel initiated sync flushes the list of written pages that SQLite is still appending to and the remaining pages are synced separately. This is important since extensive syncing has been found to be one of the main contributors to storage performance alongside with file system journaling [21].

Sync prioritizing

File system can remove or add pages to the page cache but is not able to control which pages gets flushed or block any pages to be flushed by to kernel initiated flush of page cache. File systems could be granted the possibility to mark pages not to be synced with kernel initiated flush or have some control about the importance of the syncing order like flags to increase or decrease the priority of the page getting flushed. Prioritizing the pages would reduce or eliminate possible redundant syncs taking place for the SQLite and other files that behave similarly. Implementing this kind of interface would probably require significant modifications for the kernel page cache and its operation so there could be a multiple of other performance reasons for not implementing this kind of optimization interface.

4.5 SQLite

It was found that SQLite related files are severely fragmented on Android due to nature of the SQLite file access pattern of appending and removing [6]. High usage of SQLite in Android and 70 % of I/O being generated from SQLite

database and its temporary files constitutes Android experiencing as high as 52 % delay of app launch times [21; 2].

4.5.1 Separate flash

There has been a shift in flash market to transfer from more reliable and better performing single-level cell (SLC) to cheaper multi-level cell (MLC) flash. This is obvious performance factor in storage, especially with high I/O intensive nature of the workload with SQLite. Moving the database files to RAM has been tested to significantly boost application launch times so improving the storage for the database would also most likely yield significantly better performance for the system [21].

Separate flash device could be used to contain all the systems SQLite database files and temporary files. Device could use the better performing lower-level cells like SLC to handle the high I/O of the SQLite and be relatively small since most SQLite database files are under 100 KB [6]. This would improve the speeds for the SQLite file operations that was most of all IO on Android and reduce I/O traffic on the other more fragile and slower disk.

This idea has been implemented with external journaling in ext4, where the journal is placed to a separate faster disk and used as a cache to improve performance [29]. This approach would be more realistic with closed hardware systems like a smart phone where there is no modularity about the internal storage. Drawback for this idea is that there might be security concerns relating to non-separation of block devices and their volumes, which for example Android relies on a lot in its partitioning scheme.

4.5.2 Mode-based heuristic

SQLite modes defined in chapter 2.3 vary in files that they produce; both temporary files and database files are accessed differently depending on the mode used. Currently all file systems are not aware which mode the database

uses, but if this information could be passed to the underlying file system it could be used to optimize allocation strategy and predict access patterns of file created by the database. Direct interface with user-space software and kernel level file system is not possible and it should not be since it raises plenty of security questions. One possibility could be to file system to assume the mode from the file extension and possibly have some other metadata in the beginning of the SQLite file.

In WAL mode the default for the checkpointing threshold is 1000 pages long [30]. File system could assume that to be the case to estimate the temporary file size to be little over 1000 pages long. File system should emphasise allocating and reserving sequential space for the temporary file to be little over 1000 pages to reduce fragmentation, so efficiency for reading would be optimal, as the file is only accessed in sequential manner. Another reason for some interface between the file system and the SQLite would be to let file system know if the default checkpointing threshold is changed.

4.6 File extension heuristic

File types differ greatly how they are accessed due to their intended purpose in the system. Study by Lee et.al. [21] shows that 70 % of all read operations are from executable files (.so, .apk) and that they are accessed in read only mode except in case of installation or updating. Firefox shows also the same kind of behaviour for font files (.ttf, .ttc) and they contribute most of the Firefox read operations.

File system knows the file extension since it manages the filenames and could use this knowledge with internal heuristic to treat these files differently since it is known how they are usually accessed. In case of an executable or font file, allocation for optimal performance for these files should be emphasized to be sequential since they only accessed for reading and not modified further. Internal heuristic inside the file system driver could always check the file extension for file when it is created and have some commonly only sequentially or randomly

accessed file extensions saved to compare the newly created file with and mark it in memory for the allocations for the file.

Multimedia files also act the same way with sequential writing when file is created and sequential reading only from there on. Some editing software might change the original file, but mainly multimedia files are only sequentially written and read and not appended anymore so allocation for these files should be emphasized to be in one chunk and there would be no need for reserving space for any appending.

4.7 External heuristic

So far suggested optimizations have been about internal heuristic of files for file systems that would need to be coded directly to the file system driver itself. This kind of approach is probably not optimal solution since file access behaviour or file extensions can change over time. In order to change heuristic for the file system it would require it always to change file system code.

More optimal suggestion is to have an external heuristic that would be provided for a file system for example in form of a file using mount option to give path to it. Formatting tool also could have an optional parameter for heuristic file and the volume would be created having the heuristic inside of the volume. External heuristic file could provide hints for file system of access patterns for writing and reading, sync usage and file sizes, all the things that was previously mentioned and since it is just some data on a file it would be more convenient to modify to accommodate different systems and environments.

File system would need to provide a well-defined syntax for the heuristic file and how it would use this information but from a user standpoint, it can be abstracted away. Aforementioned suggestions in chapter 4 could be generalized and used according to heuristic file information. Example for this is file system having different levels for emphasizing allocating new blocks sequentially or randomly

according to this files' access pattern depending on the percentage of sequential I/O or accommodation of reads and writes overall knowing the files type.

More specific file heuristic could be done with Linux process specific files and accesses, and not just assume optimization strategy from its file type. The best utilization for process specific heuristic probably would be found in closed systems or native operating system processes where there is certainty of them existing and updates to them are usually rarer than in third party apps. In Android, data suggests that Android applications are updated as often as every ten days, which could hinder the usability of the heuristic approach if the update would change the behavior of its file system usage [31]. With knowledge of the most downloaded apps, some of them could be optimized this way since they are probably also the most used ones.

This work demonstrated a bare bones method of getting human readable heuristic data with the tracing system implemented. If file systems were to adopt the idea for external heuristic to optimize its performance, same kind of software would be needed to be developed. Existing block-level and system tracing software could be developed further to satisfy the needs for this kind of heuristic.

5 Conclusion

The objective for this thesis was to study the file system usage of common Android applications and come up with possible optimizations based on the gathered data. This thesis went over tracing methods and results that at least partially confirmed and reinforced knowledge from the studies that SQLite and Android use storage quite heavily and the file access patterns are not beneficial for the flash storage.

In this thesis, we covered several propositions for file system optimization. From file access pattern and file extension, allocation and usage of page cache was found to be optimizable. Communication between SQLite and file system was theorized, so information about the database files could be related to the file

system so SQLite file operation could be optimized. Lastly, external heuristic software was proposed to gather heuristic data about file system usage in advance, which could be used to optimize file system.

Tracing software Strace that was used in this thesis was found not to be optimal for the objective. For further study in this subject, different software should be used. Existing software e.g., Androstep could probably have been a better tool for this kind of study with added block-level tracing and more intended usage for this kind of study [32].

Scope for this thesis was to only theoretically come up with optimizations for the file system. Next steps for the study would be to prototype suggested optimizations to actual software and test the performance benefits, find about possible problems that arise and evaluate concreate work needed for implementing suggested optimizations.

References

1 Hyojun Kim, Nitin Agrawal, Cristian Ungureanu. Revisiting Storage for Smartphones. [Online]. <https://www.usenix.org/system/files/conference/fast12/kim.pdf>. Accessed 2.4.2021.

2 Sangwook Shane Hahn, Sungjin Lee, Cheng Ji, Li-Pin Chang, Inhyuk Yee, Liang Shi, Chun Jason Xue, Jihong Kim. Improving File System Performance of Mobile Storage Systems Using a Decoupled Defragmenter. [Online]. <https://www.usenix.org/system/files/conference/atc17/atc17-hahn.pdf>. Accessed 2.4.2021.

3 Jayashree Mohan, Dhathri Purohith, Mathew Halpern, Vijay Chidambaram, Vijay Janapa Reddi. Storage on Your Smartphone Uses More Energy Than You Think. [Online]. <https://www.usenix.org/system/files/conference/hotstorage17/hotstorage17-paper-mohan.pdf>. Accessed 2.4.2021.

4 Tuxera Inc. 2021. [Online]. <https://www.tuxera.com/company/about-us/>. Accessed 21.4.2021.

5 Stephanie Mlot. January 14, 2021. Tesla Asked to Recall 158,000 Vehicles Over Flash Memory Failure. [Online]. <https://www.pcmag.com/news/tesla-asked-to-recall-158000-vehicles-over-flash-memory-failure>. Accessed 21.4.2021.

6 Cheng Ji, Li-Pin Chang, Liang Shi, Chao Wu, Qiao Li, and Chun Jason Xue. An Empirical Study of File-System Fragmentation in Mobile Storage Systems. [Online]. <https://www.usenix.org/system/files/conference/hotstorage16/hotstorage16_ji.pdf>. Accessed 2.4.2021.

7 D_SarMac. June 29, 2016. Linux File System Hierarchy. [Online]. <https://nepalisupport.wordpress.com/2016/06/29/linux-file-system-hierarchy/>. Accessed 20.4.2021.

8 Statcounter GlobalStats. 2021. Operating System Market Share Worldwide. [Online]. <https://gs.statcounter.com/os-market-share>. Accessed 19.4.2021.

9 Linux. 2021. Linux documentation. [Online]. <https://www.kernel.org/doc/Documentation/filesystems/ext4.txt>. Accessed 12.4.2021.

10 QNX. 2021. QNX documentation. [Online]. <http://www.qnx.com/developers/docs/qnxcar2/index.jsp?topic=%2Fcom.qnx.doc.neutrino.sys_arch%2Ftopic%2Ffsys_COW_filesystem.html>. Accessed 20.4.2021.

11 Daniel P. Bovet, Marco Cesati. 2006. Understanding the Linux Kernel, Third Edition, p.599-630. California: O'Reilly Media, Inc.

12 Wooseong Cheong, Chanho Yoon, Seonghoon Woo, Kyuwook Han, Daehyun Kim, Chulseung Lee, Youra Choi, Shine Kim, Dongku Kang, Geunyeong Yu, Jaehong Kim, Jaechun Park, Ki-Whan Song, Ki-Tae Park, Sangyeun Cho, Hwaseok Oh, Daniel D.G. Lee, Jin-Hyeok Choi, Jaeheon Jeong. 12.3.2018. A flash memory controller for 15μs ultra-low-latency SSD using high-speed 3D NAND flash with 3μs read time. [Online]. <https://ieeexplore.ieee.org/document/8310322>. Accessed 26.4.2021.

13 Red Hat. 2021. Storage administration guide. [Online]. <https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/storage_administration_guide/writebarrieronoff> Accessed 28.4.2021.

14 Google. 2021. About the Android Open Source Project. [Online]. <https://source.android.com/>. Accessed 19.4.2021.

15 S. O'Dea. Apr 16, 2021. Android - Statistics & Facts. [Online]. <https://www.statista.com/topics/876/android/> Accessed 19.4.2021.

16 Google. Last updated 2021-04-27. Android documentation. [Online]. <https://source.android.com/devices/automotive/start/what_automotive> Accessed 19.4.2021.

17 Google. Last updated 2021-03-11. Android documentation. [Online]. <https://developer.android.com/guide/platform>. Accessed 19.4.2021.

18 Google. Last updated 2021-04-27. Android documentation. [Online]. <https://source.android.com/devices/bootloader/partitions>. Accessed 19.4.2021.

19 Google. 2020-09-01. Android documentation. [Online]. <https://source.android.com/devices/storage/adoptable>. Accessed 19.4.2021.

20 SQLite. 2021. SQLite documentation. [Online]. <https://sqlite.org/mostdeployed.html>. Accessed 21.4.2021.

21 Kisung Lee, Youjip Won. 2012. Smart Layers and Dumb Result: IO Characterization of an Android-Based Smartphone. [Online]. <http://www.esos.hanyang.ac.kr/files/publication/conferences/international/Smart_Layers_and_Dumb_Result.pdf>. Accessed 2.4.2021.

22 Linux manual. 2020-11-29. [Online]. <https://man7.org/linux/man-pages/man1/strace.1.html>. Accessed 15.4.2021.

23 Google. Last updated 2021-01-27. Android documentation. [Online]. <https://developer.android.com/guide/components/intents-filters>. Accessed 28.4.2021.

24 Google. 2021-04-27. Android documentation. [Online]. <https://source.android.com/devices/architecture/hidl/binder-ipc>. Accessed 28.4.2021.

25 Alex Conway, Ainesh Bakshi, Yizheng Jiao, Yang Zhan, Michael A. Bender, William Jannen, Rob Johnson, Bradley C. Kuszmaul, Donald E. Porter, Jun Yuan, Martin Farach-Colton. File Systems Fated for Senescence? Nonsense, Says Science!. [Online]. <https://www.usenix.org/system/files/conference/fast17/fast17-conway.pdf>. Accessed 2.4.2021.

26 Computer Hope. Updated 11/16/2019. File fragmentation. [Online]. <https://www.computerhope.com/jargon/f/filefrag.htm>. Accessed 26.4.2021.

27 Linux manual. 2021-03-22. [Online]. <https://man7.org/linux/man-pages/man2/posix_fadvise.2.html>. Accessed 18.4.2021.

28 Linux manual. 2021-03-22. [Online]. <https://man7.org/linux/man-pages/man2/fdatasync.2.html>. Accessed 18.4.2021.

29 Linux manual. February 2021. [Online]. <https://man7.org/linux/man-pages/man5/ext4.5.html>. Accessed 3.5.2021.

30 SQLite documentation. 2021. [Online]. <https://sqlite.org/wal.html>. Accessed 22.4.2021.

31 Udayan Kumar. June 8, 2015. Understanding Android’s Application Update Cycles. [Online]. <https://www.nowsecure.com/blog/2015/06/08/understanding-android-s-application-update-cycles/>. Accessed 2.5.2021.

32 Sooman Jeong, Kisung Lee, Jungwoo Hwang, Seongjin Lee, Youjip Won. AndroStep: Android Storage Performance Analysis Tool. [Online]. <http://www.esos.hanyang.ac.kr/files/publication/conferences/international/AndroStep.pdf>. Accessed 15.4.2021.

List of traced system calls

close, lseek, open, openat, pread64, preadv, read, readv, pwrite64, pwritev, write, writev, fdatasync, fsync, sync, syncfs, ftruncate, truncate, fallocate, fchmod, fchmodat, mkdir, mkdirat, rmdir, unlink, unlinkat, rename, renameat, renameat2, mmap, munmap