Filesystem considerations for embedded devices

ELC2015

03/25/15

Tristan Lelong

Senior embedded software engineer

ABSTRACT

The goal of this presentation is to answer a question asked by several customers: which filesystem should you use within your embedded design's **eMMC/SDCard**?

These storage devices use a **standard block interface**, compatible with traditional filesystems, but constraints are not those of desktop PC environments.

EXT2/3/4, **BTRFS**, **F2FS** are the first of many solutions which come to mind, but how do they all **compare**? Typical queries include performance, longevity, tools availability, support, and power loss robustness.

This presentation will not dive into implementation details but will instead summarize provided answers with the help of various **figures** and meaningful **test results**.

TABLE OF CONTENTS

ABOUT THE AUTHOR

- Tristan Lelong
- Embedded software engineer @ **Adeneo Embedded**
- French, living in the Pacific northwest
- Embedded software, free software, and Linux kernel enthusiast.

Introduction

INTRODUCTION

More and more embedded designs rely on smart memory chips rather than bare NAND or NOR.

This presentation will start by describing:

- Some context to help understand the differences between NAND and MMC
- Some typical requirements found in embedded devices designs
- Potential filesystems to use on MMC devices

INTRODUCTION

Focus will then move to block filesystems. How they are supported, what feature do they advertise.

To help understand how they compare, we will present some **benchmarks and comparisons** regarding:

- Tools
- Reliability
- Performances

Block devices

MMC, EMMC, SD CARD

Vocabulary:

- **MMC**: MultiMediaCard is a memory card unveiled in 1997 by SanDisk and Siemens based on NAND flash memory.
- **eMMC**: embedded MMC is just a regular MMC in a BGA package, that is solded to the platform.
- **SD Card**: SecureDigital Card was introduced in 1999 based on MMC but adding extra features such as security.

MMC

This presentation will use term **MMC** to refer to these 3.

INSIDE MMC

The MMC is composed by 3 elements:

- **MMC interface**: handle communication with host
- **FTL** (Flash translation layer):
- **Storage area**: array of SLC/MLC/TLC NAND chips

FTL

The **FTL** is a small controller running a firmware. Its main purpose is to transform logical sector addressing into NAND addressing. It also handles:

- Wear-leveling
- Bad block management
- Garbage collection.

FTL firmware

FTL firmware is usually a **black box**, and doesn't allow any kind of control or tuning.

JEDEC SPECIFICATIONS

MMC specifications are handled by the JEDEC organisation: http://www.jedec.org

Current JEDEC version is v5.1 (*JESD84-B51.pdf* published in Febuary 2015)

BLOCK VERSUS MEMORY TECHNOLOGY DEVICES

Block and memory technology devices are fundamentaly different.

- **Block devices**: sector addressing. Offers read / write operations
- **Memory technology devices**: sector / subpage / page addressing. Offers read / write / erase operations

Erase operation

On NAND or NOR devices, once a bit is flipped to *0*, only an erase operation can flip it back to *1*.

BLOCK VERSUS MEMORY TECHNOLOGY DEVICES

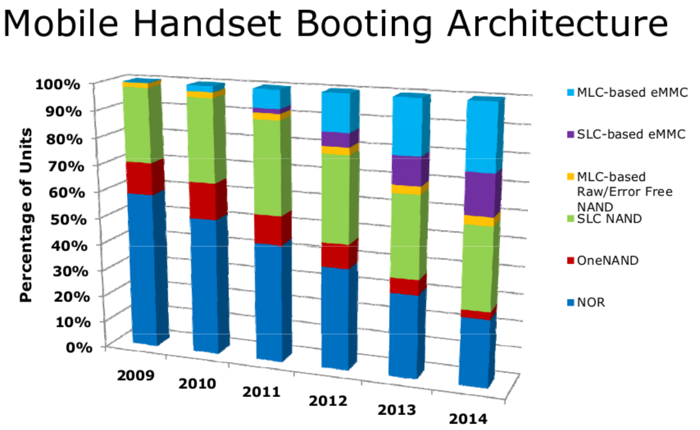
MTD also has some other specificities:

- Short lifetime per cell (due to max number of erase cycle), requires to spread operations on the entire array.
- Bad block table
- ECC
- Spare area (OOB)

Warning

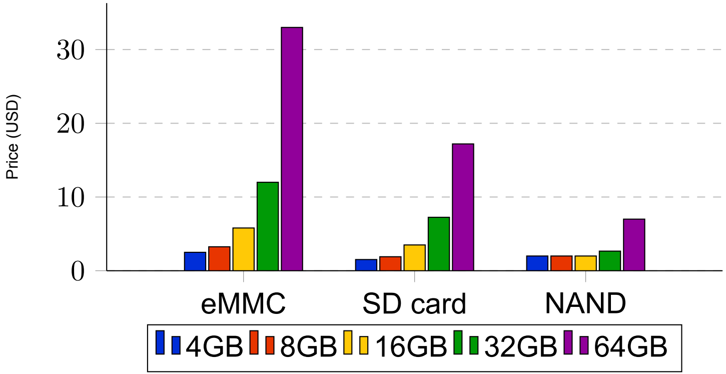
All this is usually handled by the filesystem itself. This requires **new specific filesystems** for MTD.

STATISTICS



Source: Micron marketing via jedec.org

PRICE AND CAPACITY



Source: chinaflashmarket.com

PROS AND CONS

Pros of MMC:

- Standard filesystems are compatible
- No extra operation to do (wear leveling, bad blocks, erase, garbage collection)
- Consistent in bootloader and kernel

Cons of MMC:

- Less control (no tuning possible)
- Need to trust the manufacturer
- Usually more expensive

DESKTOP VERSUS EMBEDDED SYSTEMS

MMC uses the same filesystems as the one found on desktop or servers.

Embedded devices have different requirements, therefore selection critria are not those of usual PC.

- **Bandwidth**:
  - Important for boot time for instance
  - Not the same figures (x5 to x100)
- **Reliability**: need to be robust to power loss and auto fix error in case of corruptions
- **Efficiency**: the more efficient the usage, the less power it requires
- **Cpu usage**: embedded processors are often less powerful than traditional desktop

Available filesystems

FILESYSTEMS IN LINUX

WHY NOT ZFS

- Provides strong data integrity
- Supports huge filesystems
- **Not intended for embedded systems** (requires RAM)
- **License not compatible with Linux**

WHY NOT REISERFS

- Reiser3 version is **not supported** anymore
- Reiser4 is **not mainline** Linux

WHY NOT RELIANCE NITRO

Datalight provides a custom filesystem for MMC

- **Reliance Nitro**: Filesystem
- **FlashFXe**: optimization layer for accesses on MMC.

Some more info can be found on the product page and datasheet.

- Works on **Linux**, Windows, VXWorks, and several RTOS
- Not free software (evaluation license available)
- VFS layer clear but core is **obfuscated**

HISTORY & SUPPORT EXT4

This file system is used in most of the Linux distribution that can be found.

- **EXT** filesystem was created in April 1992
- **EXT2** replaced it in 1993
- **EXT3** evolution added a journal and was merged in 2001
- **EXT4** arrived as a stable version in the Linux kernel in 2008

PRINCIPLE EXT4

EXT4 is a journalized file system. It adds on top of EXT3:

- Large file support, and better performances on large files
- Journal checksum to improve reliability
- Fast fsck
- Better handling of fragmentation

EXT4 is backward compatible with previous versions, and should provide better performances when used for EXT2 or EXT3 devices.

HISTORY & SUPPORT BTRFS

BTRFS is a new file system compared to EXT originally created by Oracle in 2007.

- Mainlined in 2009
- Considered stable in 2014

It is already the default rootfs for openSUSE.

BTRFS inspires from both Reiserfs and ZFS.

PRINCIPLE BTRFS

BTRFS stands for **B-tree filesystem**.

It brings new features to traditional filesystems:

- **Cloning/snapshots**
- Diffs (send/receive)
- Quotat
- Union
- **Self healing** (with commit periods defaulting to 30s)

HISTORY & SUPPORT F2FS

F2FS is also a new filesystem created by Samsung and stands for **Flash Friendly filesystem**.

F2FS was integrated in the Linux kernel in 2013, and is still considered unstable, even though being used in several consumer products already.

PRINCIPLE F2FS

F2FS aims at creating a NAND flash aware filesystem.

It is a log filesystem, and can be tuned using many parameters to allow best handling on different supports.

F2FS features:

- Atomic operations
- Defragmentation
- TRIM support

HISTORY & SUPPORT FAT

FAT is a really simple yet lightweight and fast filesystem.

FAT exists for than 30 years and used to be the file system used **by default on SD Cards**.

PRINCIPLE FAT

FAT design is **simple** and therefore lacks the feature set of modern filesystems, and **doesn't provide much reliability**.

It relies on the File Allocation Table, a static table allocated at format time. Any corruption of this table might be **fatal to the entire filesystem**.

FAT and flash memory

Since flash memory used to be shipped pre-formatted with a FAT filesystem, several FTL were optimized for it and deliver the **best performances** when used with FAT.

HISTORY & SUPPORT XFS

XFS was developed by SGI in 1993.

- Added to Linux kernel in 2001
- On disk format updated in Linux version 3.10

PRINCIPLE XFS

XFS is a journaling filesystem.

- Supports huge filesystems
- Designed for scalability
- **Does not seem to be handling power loss well**

HISTORY & SUPPORT NILFS2

NILFS stands for New implementation of log filesystem.

- Developed by Nippon Telegraph and Telephone Corporation
- NILFS2 Merged in Linux kernel version 2.6.30

PRINCIPLE NILFS2

As its name shows, NILFS2 is a log filesystem.

- Relies on **B-Tree** for inode and file management
- **CoW** for checkpoints and snapshots.
- Userspace garbage collector

JOURNALIZED

A journalized filesystem keep track of every modification in a **journal** in a dedicated area.

- The journal allow to **restore** a corrupted filesystem
- Modification is first recorded in the journal
- Modification is applied on the disk
- If a corruption occurs: FS will either keep or drop the modification
  - Journal is consistent: we replay the journal at mount time
  - Journal is not consistent: we drop the modification

JOURNALIZED

Well known journalized filesystems:

- EXT3, EXT4
- XFS
- Reiser4

B-TREE/COW

B+ tree is a data structure that generalized binary trees.

Copy on write is a mechanism that will allow an immediate copy of a data, and perform the real copy only when one tries to update.

CoW is used to ensure no corruption occurs at runtime:

- Modification done on a file is done on a copy of the block
- Old version of the block is preserved until modification is fully done: **transaction commited**
- If an interruption occurs while writing the new data, old data can be used.

COW

Well known filesystems using CoW:

- ZFS
- BTRFS
- NILFS2

LOG

A log filesystem will write data and metadata **sequentially** to the storage as a log.

- Recovering from corruption is done by using the **last consistent block** of data in the log for each entry.
- The tail of the log as to be reclaimed as free space in the background: **garbage collection**

Log filesystems take the assumptions that read requests will result in cache hit, since files are scattered on the storage, making it slower.

LOG

Well known log filesystems:

- F2FS
- NILFS2
- JFFS2
- UBIFS

Performances

CLASSES

The concept of classes describe the **minimum speed** (write speed) of an SD Card:

Class name	**Min speed**
Class 2	2 MB/s
Class 4	4 MB/s
Class 6	6 MB/s
Class 10	10 MB/s
UHS1	10 MB/s
UHS3	30 MB/s

HARDWARE USED

The following tests are performed using 3 different SD Cards and 1 eMMC chip:

- **Kingston** class 10
- **Samsung** class 10

The testing is done on a beagleboneblack since it offers on eMMC be default:

- **Micron MTFC4GLDEA 0M WT** (eq class 6)

SOFTWARE TOOLS

The testing are performed using the following software components:

- **Linux kernel** 3.12.10
- **Linux kernel** 3.19
- **buildroot** rootfs
- **fio** 2.1.4
- **e2fsprogs** 1.42.12
- **btrfs-tools** 3.18.2
- **f2fs-tools** git (2015-02-18)
- **xfsprogs** 3.1.11
- **nilfs-tools** 2.2.1

PARAMETERS USED

One document gives hints to tune some filesystems for NAND based flash operation. It is available on eLinux:

EMMC-SSD File System Tuning Methodology

Common options are:

- **noatime**: minimize writes
- **discard**: enable use of TRIM

PARAMETERS USED

EXT4

- Disable journal: faster write (but less reliable)
- `mkfs --stripe` size options. Should be the number of blocks inside an erase block.

BTRFS

- SSD mode (automatic)
- `mkfs --leafsize` option. Should be equal to block size

F2FS

- `mkfs -s` and `-z` options. `s` should be erase size and `z` 1

PARAMETERS USED CONT'D

XFS

- `mkfs -b` Should be equal to block size

NILFS2

- `mkfs -b` Should be equal to block size
- `mkfs -B` number of blocks in 1 segment. Should be the number of blocks inside an erase block.

PARAMETERS USED

Using the geometry tuning is not portable:

- Requires to run some **benchmark** to first detect the MMC geometry
- Check if there is a real gain.

tuning

`flashbench` can help deduce correct geometry by analyzing performance gaps.

BANDWIDTH

Several use cases will be tested using `fio` using only the latest kernel version 3.19

1. Mono threaded random read
   - *ex: boot time*
2. Mono threaded random write
   - *ex: data write into database*
3. Mono threaded sequential read
   - *ex: video streaming*
4. Mono threaded sequential write
   - *ex: video capture/recording*
5. Multi threaded sequential/random read/write
   - *ex: a real system with high I/O load*

FIO

`fio` is an I/O generation tool used for benchmarking

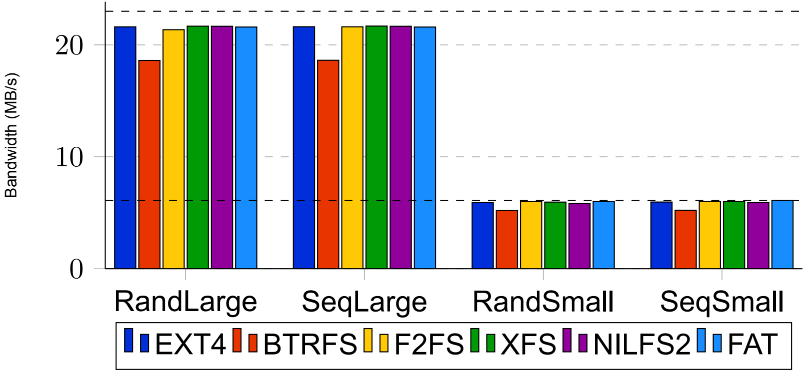
- **Highly configurable**
- Offers a lot of parameters
- Description of jobs
- Exports a lot of **statistics**

BANDWIDTH TEST CONDITIONS

`fio` job description

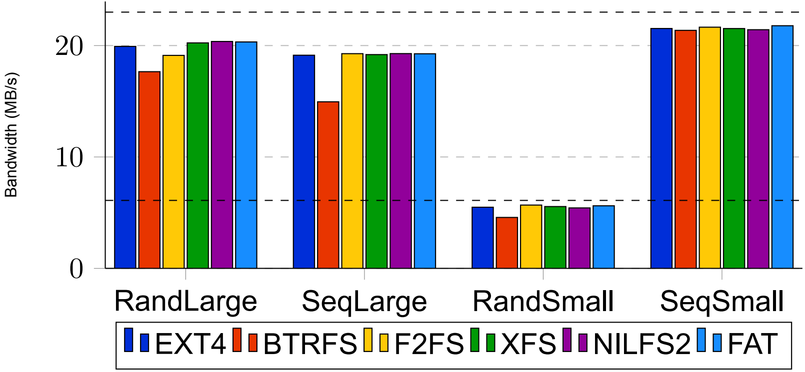
```
name=<test name>
rw=[randread | randwrite | read | write]
size=500MB
blocksize=[4MB | 4kB]
nrfiles=50
direct=[0 | 1]
buffered=[1 | 0]
numjobs=1
ioengine=libaio
```

READ PERFORMANCES DIRECT



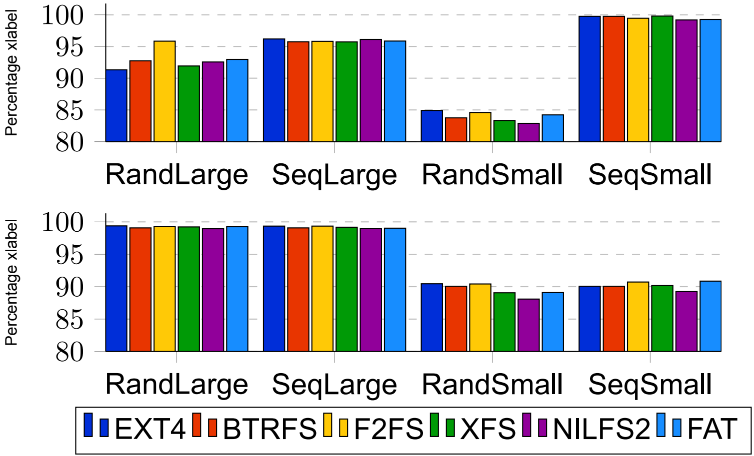
- Filesystems are not the bottleneck when reading
- Large buffers show better performances
- Sequential or Random is not a problem when reading

READ PERFORMANCES BUFFERED



- Small buffers are fast when using non direct I/O and maximize the bandwidth

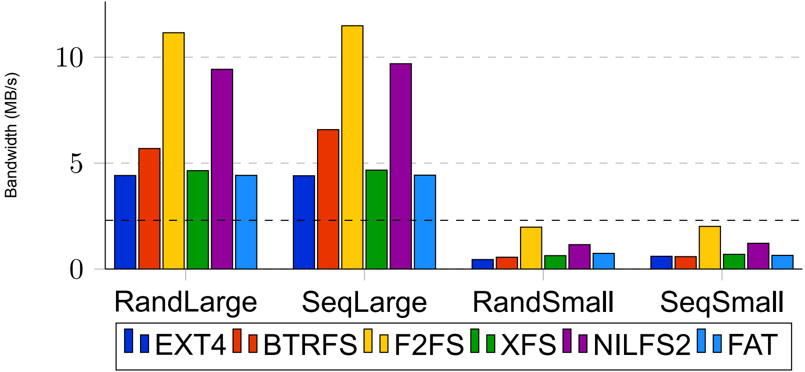
READ BUS USAGE (BUFFERED / DIRECT)



READ BUS USAGE

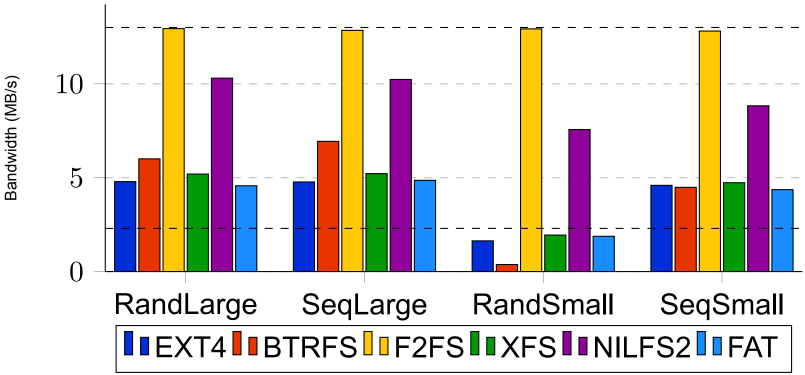
- **Direct mode**: small buffered cannot be merged
- **Buffered mode**: sequential small buffers maximize throughput

WRITE PERFORMANCES DIRECT



- F2FS and NILFS2 are the fastest in all cases

WRITE PERFORMANCES BUFFERED

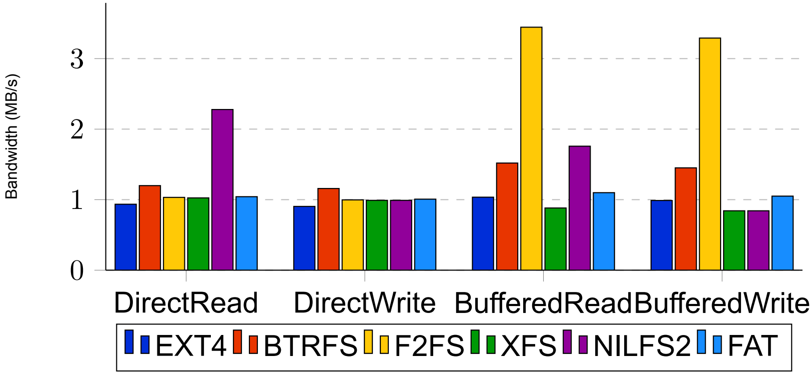


- F2FS shows impressive buffered write performances (log designed)
- Buffering really helps BTRFS again with small sequential buffers

WRITE PERFORMANCES BUS USAGE

- Bus usage is close to **100%** (buffered or direct) when writing
- F2FS clearly shows the best performances by far on this Samsung class 10 SD Card

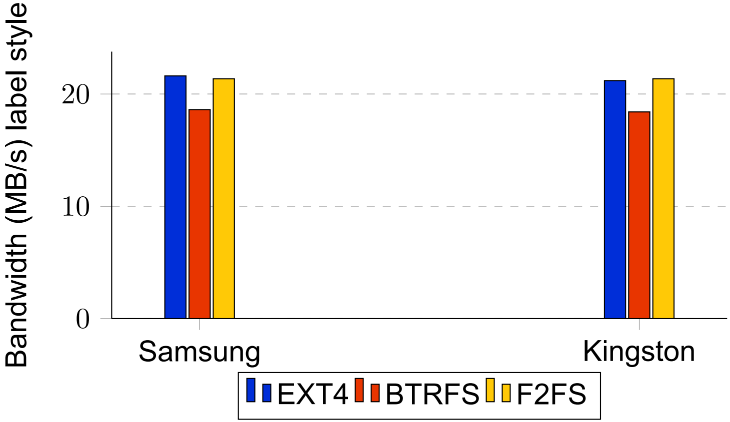
MIXED PERFORMANCES



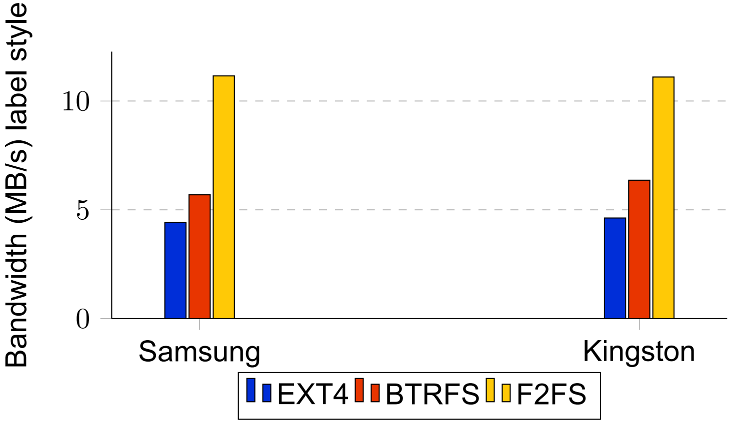
MIXED PERFORMANCES

- F2FS **scales better** on buffered I/O
- EXT4 is for once way below both BTRFS and F2FS
- XFS doesn't scale that well on MMC
- NILFS2 results might be wrong and need to be checked

READ PERFORMANCES SUPPORTS



WRITE PERFORMANCES SUPPORTS



WRITE PERFORMANCES SUPPORTS

Test done on direct I/O, large sequential blocks.

- Both SD Cards show approximately the same performances
- **No specific tuning in F2FS** for Samsung SD Cards

BOOT TIME

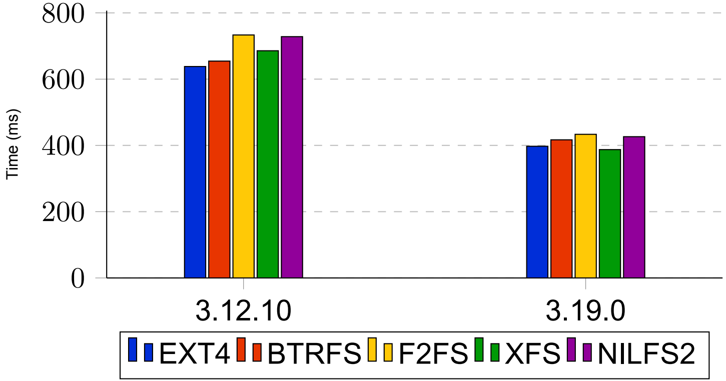
Description:

- Load the MMC with the buildroot rootfs (about 15MB)
- Measure time using `grabserial` between the mounting of the rootfs and the console prompt

Note

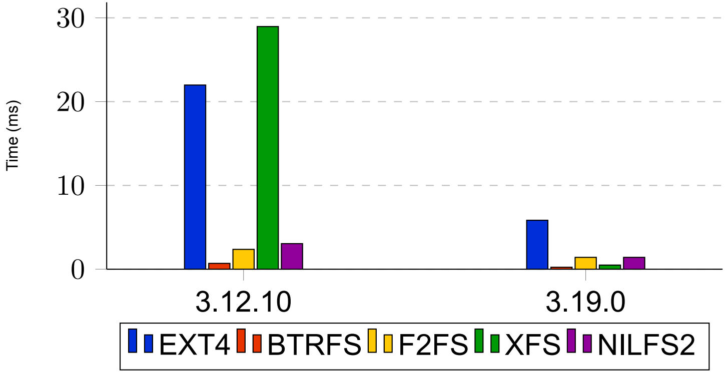
The kernel `rootfstype` will be set to the fs type in order to avoid the lookup of the filesystem.

BOOT TIME DEPENDING ON KERNEL VERSION



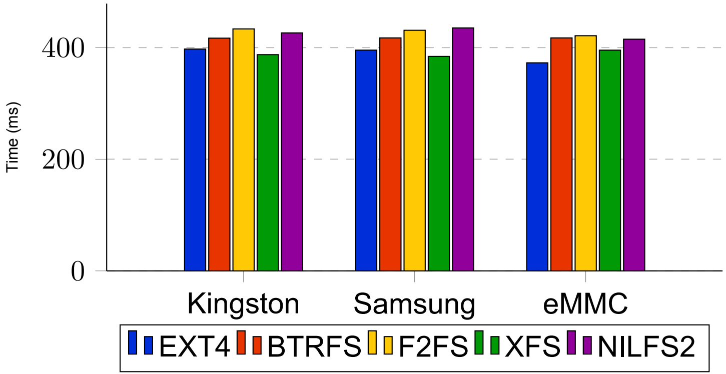
- Great performance gain for last 18 month
- Gap is closing between EXT4 and challengers

BOOT TIME VARIATIONS DEPENDING ON KERNEL VERSION



- EXT4 and XFS variations makes them less deterministic
- Linux 3.19 shows 1% max variation for EXT4 and less 0.3% for the others

BOOT TIME DEPENDING ON SUPPORT



- All 3 shows same kind of figures

MOUNT TIME

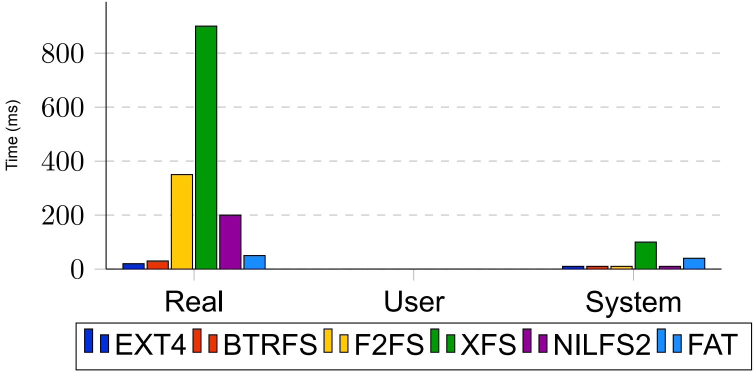
Description:

- Load the MMC with a large rootfs (1GB) 60% filled
- Measure time using `time` for the mount command to run

Note

The filesystem type needs to be specified using the `-t` option in order to avoid the lookup of the filesystem.

MOUNT TIME



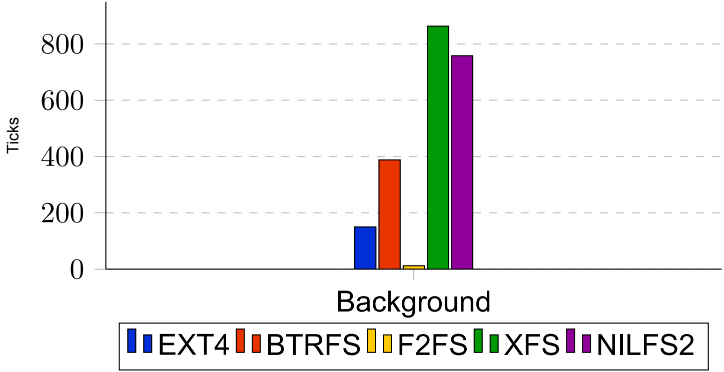
- F2FS & NILFS2 show bigger delay for mounting even a clean partition
- XFS shows the biggest delay for mounting even a clean partition

TEST DESCRIPTION

Description:

- Mount the filesystem
- Perform a fixed amount of I/O operations on the mountpoint: 38GB
- Measure time using `/proc/[pid]/stat` for every kernel thread

TEST RESULTS



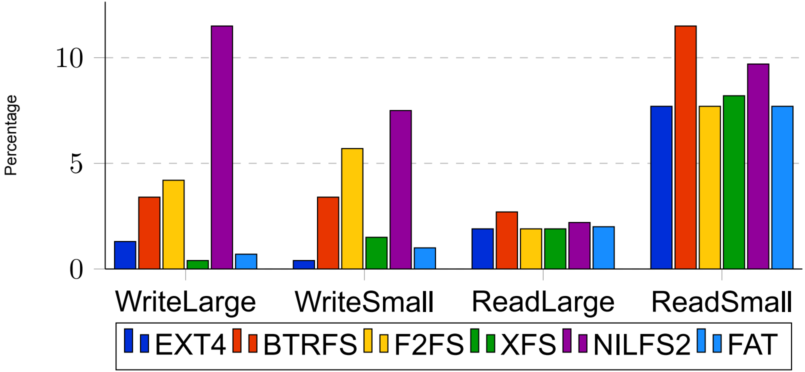
- Even though CPU usage can vary by **1 order of magnitude**, Background tasks are negligible.

CPU USAGE

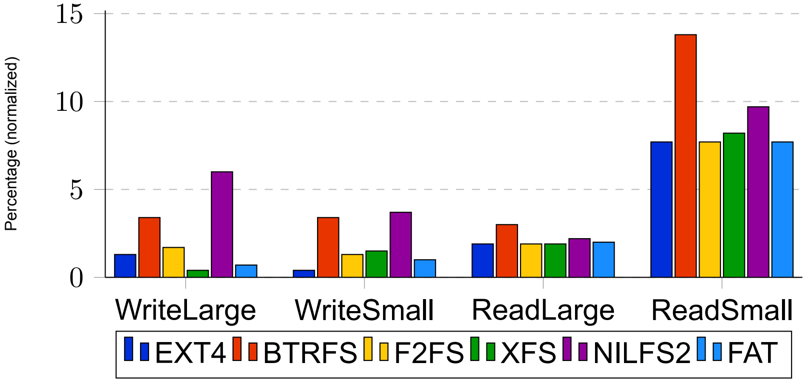
Description:

- Mount the filesystem
- Perform a fixed amount of I/O operations on the mountpoint
- Extract time using `fio` output

EFFICIENCY


Percentage
0
5
10
WriteLarge
WriteSmall
ReadLarge
ReadSmall
EXT4
BTRFS
F2FS
XFS
NILFS2
FAT

EFFICIENCY CONT'D



EFFICIENCY

The tests show the average CPU usage for the duration of the complete test.

- Needs to **compare with I/O real duration**
- Write operation takes longer than CPU to copy: Uses less relative CPU time
- BTRFS is not CPU efficient
- F2FS and NILFS2 uses more CPU for writing but I/O duration is shorter
- F2FS is clearly more efficient than NILFS2

Tools

MKFS TOOL

This is the most basic task done by `mkfs`:

- `mkfs.ext4 [-d <offline folder>]` only with patches
- `mkfs.btrfs [--rootdir <offline folder>]`
- `mkfs.f2fs`
- `mkfs.xfs`
- `mkfs.f2fs`

MKFS STATS

Statistics on filesystem after formatting:

FS	Total	Empty MB used
EXT4	976 MB	1.3 MB
BTRFS	1024 MB	0.25 MB
F2FS	1023 MB	141 MB
XFS	981 MB	32 MB
NILFS	936 MB	16 MB

Once mounted all filesystems will create kernel threads.

- **EXT4**: 2 kthreads
- **BTRFS**: 23 kthreads
- **F2FS**: 1 kthread
- **XFS**: 5 kthreads
- **NILFS**: 1 kthread

FSCK

Only 4 filesystems offer file system check

- `fsck.ext4`
- `btrfs check`
- `fsck.f2fs`
- `fsck.xfs` or `xfs_repair`
- NILFS will always mount the latest consistent checkpoint

FSCK

Statistics on clean filesystem check tool:

FS	Real time	Sys time + User time
EXT4	60 ms	0 ms + 10 ms
BTRFS	130 ms	20 ms + 40 ms
F2FS	2090 ms	960 ms + 740 ms
XFS	1320 ms	300 ms + 0 ms
NILFS	NA	NA

EXT4 EXTRA

The different packages that brings utilities for every filesystem usually contains the basic formatting and check tools.

- `debugfs` **Filesystem debugger** (advanced)
- `dumpe2fs` Dumps filesystem info
- `e2image` **Backup metadatas**
- `e2label` Changes the label of a filesystem
- `e4defrag` **Online defragmenter**

EXT4 EXTRA CONT'D

- `e2fsck` Filesystem check
- `fsck.ext4` link to `e2fsck`
- `mke2fs` Creates a filesystem
- `mkfs.ext4` link to `mke2fs`
- `resize2fs` Offline resize partition
- `tune2fs` Changes options on an existing filesystem

BTRFS EXTRA

BTRFS offers a lot of extra features. Most of them are available as subcommands of `btrfs` master command.

- `btrfs` **Master command** for accessing most of the BTRFS features.
  - `subvolume` **Manages subvolumes**
  - `filesystem` Manages options
  - `balance` `device` `replace` **Manages devices**
  - `scrub` Erase a filesystem
  - `check` Filesystem check
  - `rescue` Filesystem rescue

BTRFS EXTRA CONT'D

- `btrfs-convert` Converts EXT filesystem to BTRFS
- `btrfs-debug-tree` Dumps filesystem info
- `btrfstune` Changes options on an existing filesystem
- `fsck.btrfs` Does nothing (compatibility)
- `mkfs.btrfs` Creates a filesystem

BTRFS tools

Due to its structure, BTRFS cannot reliably show disk space usage using traditional tools and one must rely on `btrfs` command for this.

F2FS EXTRA

F2FS is still new and doesn't really offer any extra feature:

- `mkfs.f2fs` Creates a filesystem
- `fsck.f2fs` Filesystem check

XFS EXTRA

- `xfs_repair`
- `xfs_fsr`: **Online reorganize filesystem**
- `xfs_growfs`: Offline resize partition
- `xfs_freeze`: **Suspend/Resume all access to filsystem**
- `xfs_admin`: Changes options on an existing filesystem
- XFS realtime sections: Made for low latency files

NILFS2 EXTRA

- `nilfs_cleanerd`/`nilfs-clean`: Garbage collector
- `nilfs-tune`: Changes options on an existing filesystem
- `nilfs-resize`: Offline resize partition
- `chcp`: **Convert checkpoints into snapshots**
- `lscp`: **List checkpoints and snapshots**
- `mkcp`: **Create checkpoints or snapshots**
- `rmcp`: **Remove checkpoints or snapshots**

Reliability

TESTING FS RELIABLILITY

Testing the **filesystem reliability** can be done using several use cases:

- **Power loss** while writing files
- **Corrupted** writes
- Blocks going bad

TESTING FS RELIABLILITY

To simulate these:

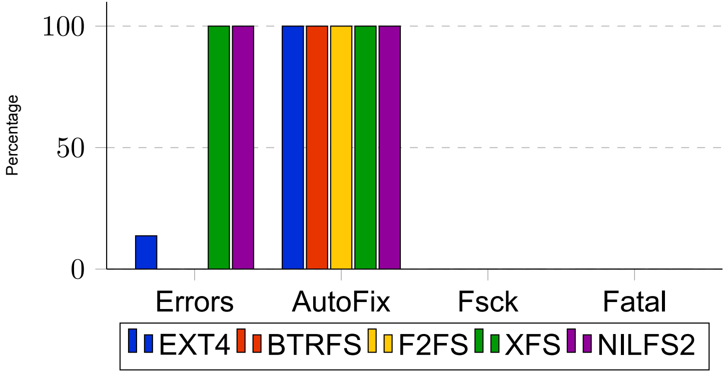
- **Watchdog** to trigger hard reboot on a system to simulate how likely the fs will fail
- Device mapper **dm-flakey** module to simulate how the fs recovers from errors
  - Ignore all writes after a certain period using `drop_writes`
  - Corrupt writes after a certain period using `corrupt_bio_byte`
  - Corrupt reads after a certain period using `corrupt_bio_byte`

CORRUPTION OF THE FILESYSTEM

Description:

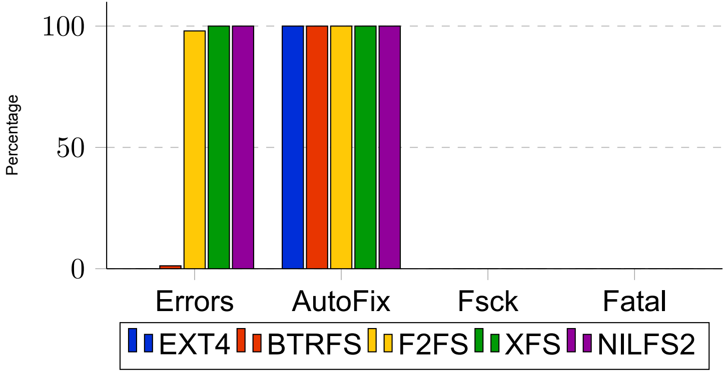
- Test **auto starts** with the board
- Mounts with sync and async options
- Write files and rely on the watchdog to cut power
- Check for mount return code, mount errors/warnings, fsck result
- Test ran for 226 iterations for each use case

CORRUPTION OF THE FILESYSTEM MOUNTED ASYNC



- EXT4 async filesystem sometimes require **journal recovery**
- All filesystem never got corrupted enough to require `fsck`

CORRUPTION OF THE FILESYSTEM MOUNTED SYNC



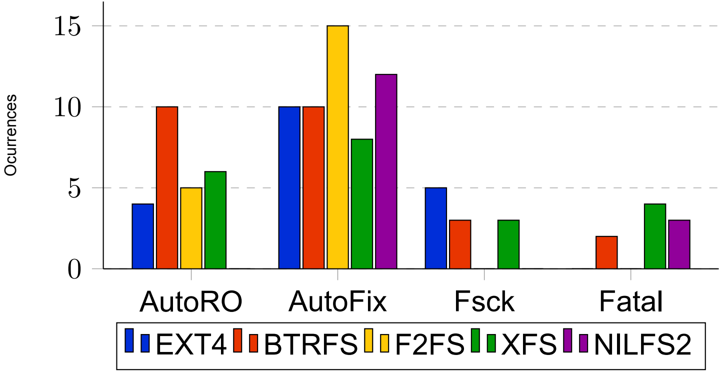
- F2FS sync filesystem almost always requires fixing
- BTRFS showed errors only 3 errors times
- No filesystem ever got corrupted enough to require fsck

DETECTION/RECOVERY OF CORRUPTED FILES

Description:

- Prepare **corruption** model by mount all filesystems using **dm-flakey** and corrupt the first byte of each block: write 00
  - Corrupt all writes after 10 seconds
  - Corrupt all writes for 1 seconds then allow writes for 1 second (**trickiest**)
- Perform the write
  - Write a 30MB random file and sync the device
  - Write multiple 1MB file and sync the disk
- Unmount and remount the partition normally then **inspect** its content

DETECTION/RECOVERY OF CORRUPTED FILES



Test done on 15 iterations

DETECTION/RECOVERY OF CORRUPTED FILES

- **EXT4**: filesystem does not mount properly
  - Sometime turns filesystem `RO`
  - `fsck` required
  - Output file is present but **zeroed** or **emptied**

DETECTION/RECOVERY OF CORRUPTED FILES

- **BTRFS**: filesystem mounts **immediately**
  - Sometime turns filesystem `RO`
  - Loses the corrupted file or present files with I/O error
  - Filesystem keeps running as expected
  - Can be **unfixable** if internal structure checksums are corrupted (backup sb?)
  - Best detection of corruption

DETECTION/RECOVERY OF CORRUPTED FILES

- **F2FS**: filesystem takes up to several minutes to mount
  - Most **robust** to this kind of corruption
  - Sometime turns filesystem `RO`
  - Auto recovery recovers most of the data (file is there with corrupted bytes)
  - File is sometimes corrupted with **no warning** but `dmesg`

DETECTION/RECOVERY OF CORRUPTED FILES

- **XFS**: Recovery can clean some old files when used with most agressive options
  -
  - Sometime feezes the filesystem
  - Auto recovery recovers from small corruptions
  - Recovered file can be truncated.

DETECTION/RECOVERY OF CORRUPTED FILES

- **NILFS2**: Filesystem can sometimenot mount at all
  - No way to recover anything since no `fsck`
  - Auto recovery recovers most of the data (file is there with corrupted bytes)
  - File is sometimes corrupted with **no warning** but `dmesg`
  - Umounting a corrupted NILFS2 system can hang undefinitely

DETECTION/RECOVERY OF CORRUPTED FILES

Fatal corruptions

Fatal corruptions only occured when writing corrupted bytes (0x00 at offset 0).

Conclusion

PERFORMANCES

When it come to **performances**:

- **EXT4** used to be the best match for embedded systems using eMMC for a long time
- New reliable and powerful alternatives are growing quickly
- **F2FS** and **NILFS2** show impressive write performances
- Performances are still device dependent and requires measurements

Feature wise:

- **BTRFS** is a next generation filesystem
- **NILFS2** provides simpler but similar features

SCALABILITY

Scalability:

- Embedded systems can have several cores
- Embedded systems can do extensive IO operations
- EXT4 clearly doesn't scale as well as **BTRFS** and **F2FS**
- XFS scalability works better on spinning disk or high bandwidth supports

Productization:

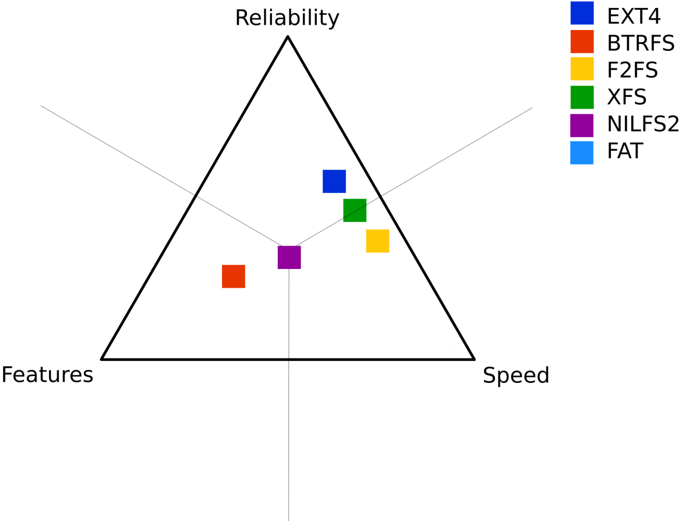
- **EXT4** is the most mature
- Google uses **F2FS** in its phones
  - Moto X, G, E family
  - Userdata partition only
  - System still a ro EXT4

RECOMMENDATIONS

Next steps:

- re-do all tests with larger datasets
- compare on lower class (Class 4) and see if the gaps are smaller
- benchmark to extract flash tuned params and compare tuned versions

RECOMMENDATIONS



USEFUL LINKS

Filesystem performances on various kernel versions:

- http://www.phoronix.com

Benchmarking

- fio-output-explained.html
- EMMC-SSD_File_System_Tuning_Methodology_v1.0.pdf

Filesystem technical documents:

- BTRFS: http://lwn.net/Articles/576276/
- F2FS: http://haifux.org/lectures/293/f2fs.pdf
- F2FS: http://lwn.net/Articles/518988
- NILFS: http://www.nilfs.org/papers/overview-v1.pdf

QUESTIONS