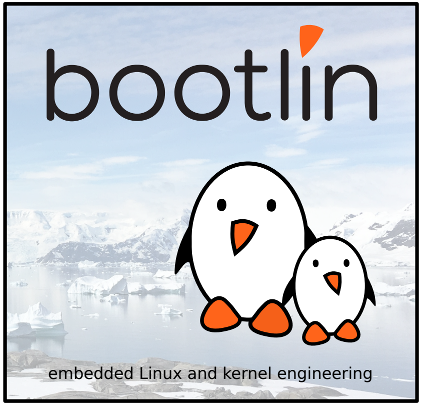
Finding the Best Block Filesystem

Michael Opdenacker
michael.opdenacker@bootlin.com



Strč prst skrz krk

Strč prst skrz krk

Happy to be in the Czech Republic!

Strč prst skrz krk

Happy to be in the Czech Republic!
But let's do the presentation in English this time
;-)

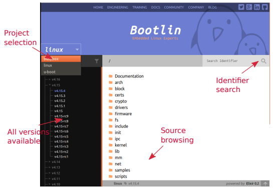
Strč prst skrz krk

Happy to be in the Czech Republic!
But let's do the presentation in English this time
;-)

https://en.wikipedia.org/wiki/Strč_prst_skrz_krk

Michael Opdenacker

- Founder and Embedded Linux engineer at Bootlin:
  - Embedded Linux **expertise**
  - **Development**, consulting and training
  - Strong open-source focus
- About myself:
  - Always happy to learn from every new project, and share what I learn.
  - Initial author of Bootlin's freely available embedded Linux, kernel and boot time reduction training materials (`https://bootlin.com/docs/`)
  - Current documentation maintainer for the Yocto Project
  - Current maintainer of the Elixir Cross Referencer, making it easier to study the sources of big C projects like the Linux kernel. See `https://elixir.bootlin.com`.



Abstract

It can be difficult to find the most appropriate filesystem for your embedded system's eMMC or SD card storage. You can benchmark your system with each of them, but it can be time consuming. In this talk, we will compare all the actively maintained block filesystems supported in the Linux kernel: Ext2, Ext4, XFS, Btrfs, F2FS, SquashFS and EROFS. Each of them will be properly introduced, with its basic design principles and main features.

We will then compare each filesystem in terms of kernel module size and load time, filesystem mount time (important for boot time), filesystem size, as well as read and write performance on a few simple scenarios. We will also look for the best compression algorithms for filesystems with compression options. Performance comparisons will be run both on a 32 bit ARM board and on a 64 bit ARM one, both using a fast SD card as storage device.

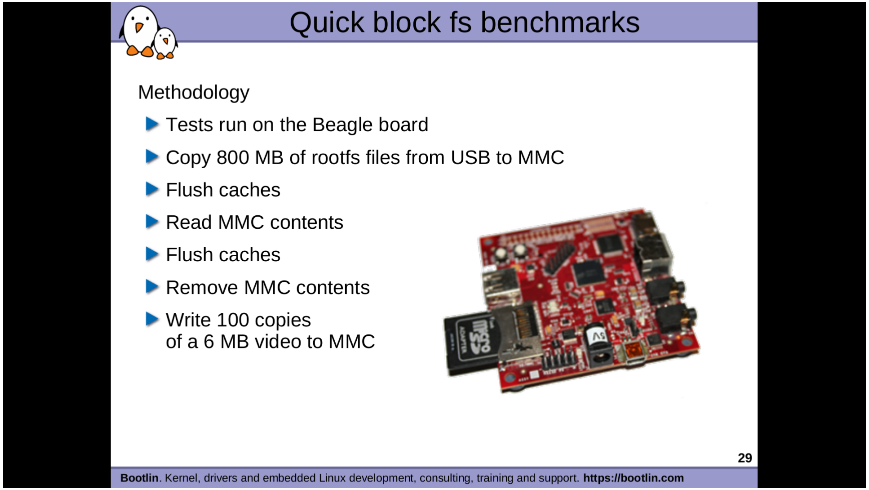
Filesystem performance can really depend on the benchmark, on your storage and on your CPU, so no universal results should be expected. However, you will learn what the best solution is in specific hardware configurations and testcases.

Disclaimer

- I'm not a filesystems expert
- I'm just a regular embedded Linux engineer who was given (a significant amount of) time by his company to do some research on this topic (thanks!)

Main goals

Not my first presentation on this topic:
https://bootlin.com/pub/conferences/2010/elce/elce2010-flash-filesystems.pdf



Quick block fs benchmarks

Methodology

- Tests run on the Beagle board
- Copy 800 MB of rootfs files from USB to MMC
- Flush caches
- Read MMC contents
- Flush caches
- Remove MMC contents
- Write 100 copies of a 6 MB video to MMC

29

Bootlin. Kernel, drivers and embedded Linux development, consulting, training and support. **https://bootlin.com**

Main goals

- Rotating block devices are very rare now, especially in embedded systems.
- Solid State Disks didn't exist when some filesystems were created. Filesystems have also evolved since 2010, in particular new ones have appeared (F2FS, EROFS).
- So, the main goal is to help you find out which filesystems are likely to work best for your embedded projects with Solid State storage.
- We chose to make our tests on MMC/SD, the most common type of storage on embedded systems.

How to know whether a device is Solid State or not?

- Check whether `/sys/block/<device>/queue/rotational` contains `0`.
- Unfortunately, this doesn't work for USB mass storage, always reported as rotational.
- Correct information for MMC/SD.

Available filesystems

Ext2

One of the earliest Linux filesystem, introduced in 1993

- `filesystems/ext2`
- Still actively supported. Low metadata overhead, module size and RAM usage
- But risk of metadata corruption after an unclean shutdown. You then need to run `e2fsck`, which takes time and may need operator intervention. Can't reboot autonomously.
- First successor: `ext3` (2001), addressing this limitation with *Journaling* [1], but wasn't scaling well. Now deprecated.
- *Journalism* reduces corruption and loss of information ;-)
- Date range: December 14, 1901 – January 18, 2038!

Not recommended for embedded systems!

[1] https://en.wikipedia.org/wiki/Journaling_file_system

Ext4

The modern successor of Ext2

- First introduced in 2006, filesystem with Journaling, without `ext3` limitations.
- Still actively developed (new features added). However, considered in 2008 by Ted Ts'o as a "stop-gap" based on old technologies.
- The default filesystem choice for many GNU/Linux distributions (Debian, Ubuntu)
- The `ext4` driver also supports `ext2` and `ext3` (one driver is sufficient).
- Noteworthy feature: transparent encryption (but compression not available).
- Date range: 1901 – 2446. Fine for embedded systems without a 400 Y+ warranty!
- Minimum partition size to have a journal: 8MiB.
- Minimum partition size without a journal: 256KiB (only 32 inodes!).
- Create the filesystem with `mkfs.ext4`.

https://en.wikipedia.org/wiki/Ext4

XFS

A Journaling filesystem

- Since 1994 (started by Silicon Graphics for the IRIX OS)
- Actively maintained and developed by Red Hat now
- Features: variable block size, direct I/O, online growth...
- Minimum partition size: 16MiB (9.7MiB of free space)
- Create the filesystem with `mkfs.xfs`.

https://en.wikipedia.org/wiki/XFS

Btrfs

A copy-on-write filesystem (another Czech word!)

- Pronounced as "better F S", "butter F S" or "b-tree F S", since 2009.
- A modern filesystem with many advanced features: volumes, snapshots, transparent compression...
- Minimum partition size: 109MiB (only 32MiB of free space).
- Create the filesystem with `mkfs.btrfs`.

https://en.wikipedia.org/wiki/Btrfs

F2FS — Flash-Friendly File System

A log-structured filesystem

- Since 2012 (started by Samsung, actively maintained)
- Designed from the start to take into account the characteristics of solid-state based storage (eMMC, SD, SSD)
- In particular, trying to make most writes sequential (best on SSD)
- Support for transparent encryption and compression (LZO, LZ4, Zstd), possible on a file by file (or file type) basis, through extended file attributes.
- Maximum partition size: 16TB, maximum file size: 3.94TB
- Minimum partition size: 52MiB (8MiB free space)
- Create the filesystem with `mkfs.f2fs`.

https://en.wikipedia.org/wiki/F2FS

NILFS — New Implementation of a Log-structured File System

A log-structured filesystem too, also known as NILFS2

- Since 2005 (started by NTT, maintained but not very actively)
- Treating the storage medium as a circular buffer, new blocks are always written to the end.
- Provides continuous snapshotting, easy to restore files modified or deleted at any recent time. This create a weird behavior though: all past files (even erased ones) are kept and this fills up all available space. Needs to run and configure `nilfs_cleanerd` to avoid this (you also need to stop it before you can unmount your partition!).
- Supposed to be great at latency (minimizes seek time) and be the best at handling many small files.
- Minimum partition size: 129MiB (48MiB free space)
- Create the filesystem with `mkfs.nilfs2`.

https://en.wikipedia.org/wiki/NILFS

SquashFS — A Read-Only and Compressed File System

The most popular choice for this usage

- Started by Phillip Lougher, since 2009 in the mainline kernel, actively maintained.
- Fine for parts of a filesystem which can be read-only (kernel, binaries...)
- Used in most live CDs and live USB distributions
- Supports several compression algorithms (Gzip, LZO, XZ, LZ4, Zstd)
- Supposed to give priority to compression ratio vs read performance
- Suitable for very small partitions
- Create a filesystem image with `mksquashfs`.

https://en.wikipedia.org/wiki/SquashFS

EROFS — Enhanced Read-Only File System

A more recent read-only, compressed solution

- Started by Gao Xiang (Huawei), since 2019 in the mainline kernel.
- Used in particular in Android phones (Huawei, Xiaomi, Oppo...)
- Supposed to give priority to read performance vs compression ratio
- EROFS implements compression into fixed 4KB blocks (better for read performance), while SquashFS uses fixed-sized blocks of uncompressed data.
- Unlike Squashfs, EROFS also allows for random access to files in directories.
- Development seems more active than on SquashFS.
- Suitable for very small partitions
- Create a filesystem image with `mkfs.erofs`.

https://en.wikipedia.org/wiki/EROFS

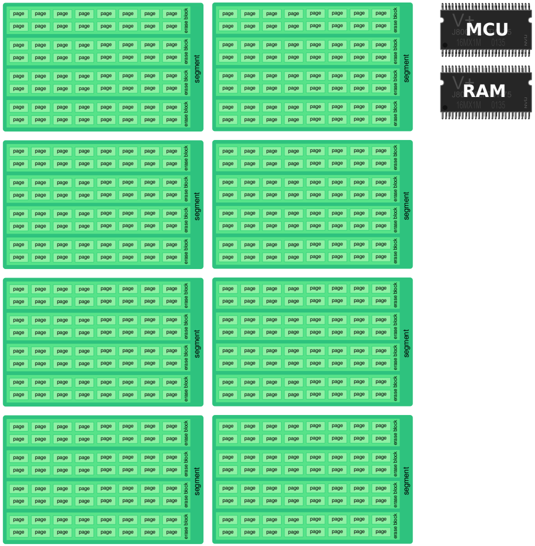
Filesystems we didn’t test

- Ext2: obsolete in 2038
- JFS: supported but legacy
- ReiserFS: lacks support, going away in a few years
- CramFS: supported but legacy
- BcacheFS: not merged yet!

Finding the Best Block Filesystem

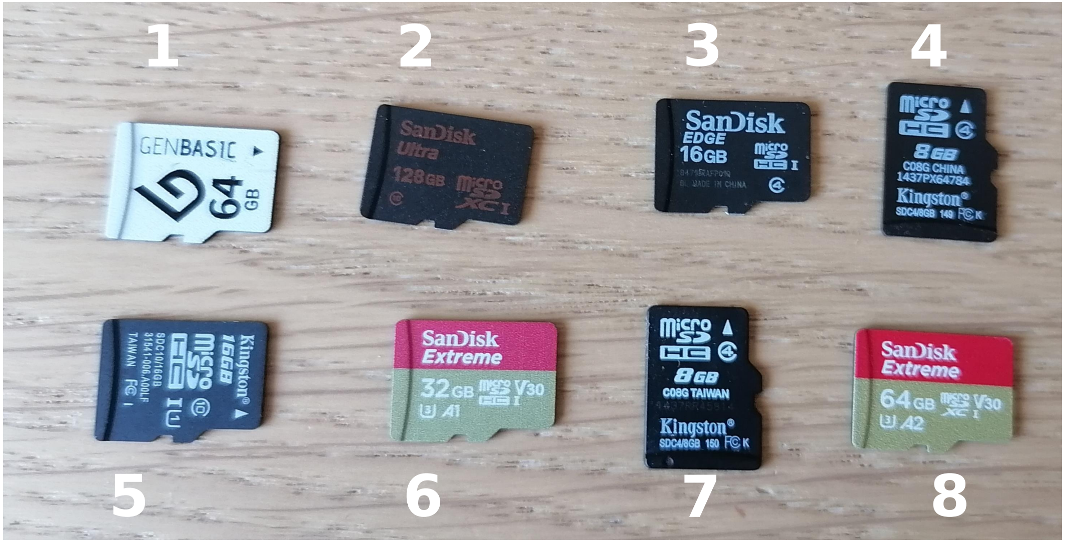
Raw benchmarks

How SD cards are organized



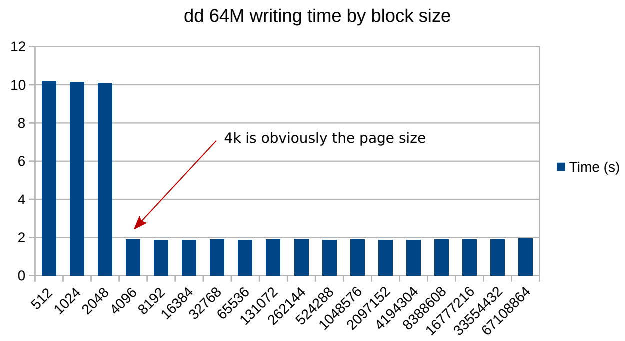
- NAND flash is organized in *pages* (typically 2-4K) and in *erase blocks* (typically 128K).
- However, SD cards group several erase blocks together in *segments* (typically 4M) which allows to address and manage more storage space. The storage will never erase anything smaller than a segment.

Comparing SD cards



How to infer the page size?

```
#!/bin/sh
b=512
while [ $b -le 67108864 ]
do
        echo -n $b
        time -f ",%e" chrt -f 99 dd status=none if=64M.img \
                      of=/dev/mmcblk0 bs=$b conv=fdatasync
        b=$(($b *2))
done
```

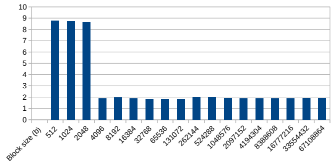


Test writing with increasing block sizes and find the best one:

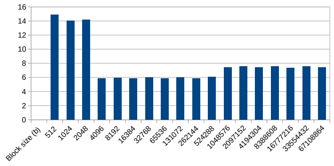
- Any write smaller than 4K is inefficient.
- Writing bigger (sequential) blocks is unnecessary.
- Good to make sure that filesystems use at least 4K blocks.

Raw write tests on 8 SD cards

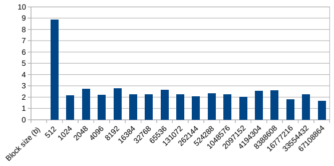
1 - Genbasic 64GB - 64M write time (s) by block size



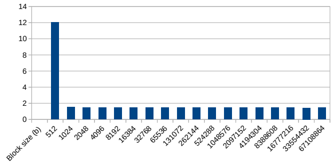
2 - Kingston 16 GB - 64M write time (s) by block size



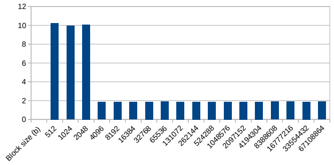
3 - Sandisk Ultra 128GB - 64M write time (s) by block size



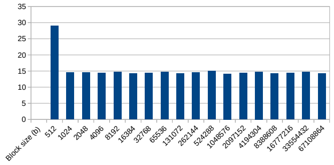
4 - Sandisk Extreme 32GB - 64M write time (s) by block size



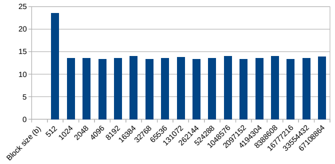
5 - Sandisk Edge 16GB - 64M write time (s) by block size



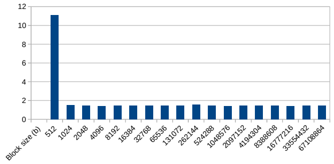
6 - Kingston 8GB Taiwan - 64M write time (s) by block size



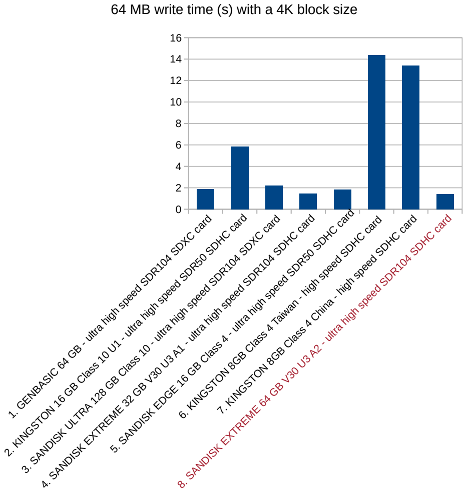
7 - Kingston 8GB China - 64M write time (s) by block size



8 - Sandisk Extreme 64GB - 64M write time (s) by block size



Raw write tests winner(s)



Note: first time I scan an SD card with a scanner!

How to infer the segment size?

A good solution is to use *flashbench* from Arnd Bergmann:

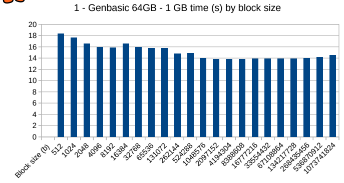
- Available as a Debian/Ubuntu package.
- Can help to find the segment size by reading across several block size boundaries.
- Also offers useful `-f` option for looking for special "FAT" sectors with better performance.
- See *Optimizing Linux with cheap flash drives* https://lwn.net/Articles/428584/

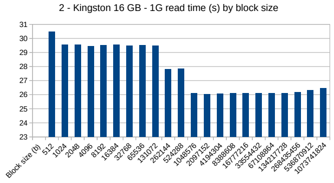
https://git.linaro.org/people/arnd/flashbench.git/

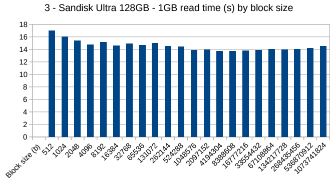
```
$ sudo flashbench -a /dev/mmcblk0
align 4294967296  pre 1.39ms   on 1.47ms   post 1.28ms   diff 139µs
align 2147483648  pre 1.3ms    on 1.47ms   post 1.33ms   diff 158µs
align 1073741824  pre 1.24ms   on 1.36ms   post 1.21ms   diff 139µs
align 536870912   pre 1.26ms   on 1.39ms   post 1.29ms   diff 114µs
align 268435456   pre 1.24ms   on 1.38ms   post 1.25ms   diff 131µs
align 134217728   pre 1.26ms   on 1.38ms   post 1.24ms   diff 130µs
align 67108864    pre 1.4ms    on 1.54ms   post 1.52ms   diff 81.6µs
align 33554432    pre 1.3ms    on 1.48ms   post 1.33ms   diff 161µs
align 16777216    pre 1.35ms   on 1.47ms   post 1.35ms   diff 116µs
align 8388608     pre 1.22ms   on 1.41ms   post 1.32ms   diff 140µs
align 4194304     pre 1.17ms   on 1.32ms   post 1.18ms   diff 147µs
align 2097152     pre 1.21ms   on 1.28ms   post 1.2ms    diff 78.4µs
align 1048576     pre 1.18ms   on 1.26ms   post 1.19ms   diff 81.4µs
align 524288      pre 1.19ms   on 1.27ms   post 1.19ms   diff 77.5µs
align 262144      pre 1.19ms   on 1.28ms   post 1.17ms   diff 96.5µs
align 131072      pre 1.16ms   on 1.25ms   post 1.18ms   diff 81.2µs
align 65536       pre 1.16ms   on 1.25ms   post 1.18ms   diff 74.4µs
align 32768       pre 1.18ms   on 1.28ms   post 1.17ms   diff 103µs
```

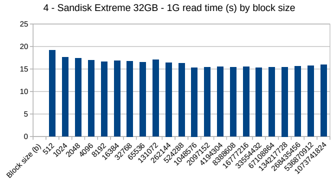
Tests on SD card 2 (KINGSTON 16 GB Class 10 U1)
Here, there is a clear level change at 4M.
That's less obvious with several of the other SD cards we surveyed, but in all cases, 4M turns out to be a safe choice (no risk to take a value that's the double of the actual one).

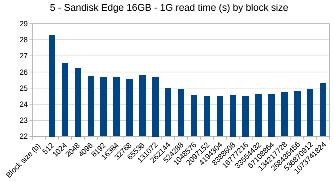
Raw read tests on 8 SD cards

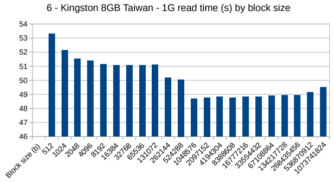


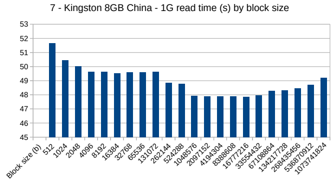


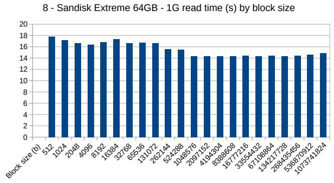












See the level change at 1M!

Finding the Best Block Filesystem

Filesystem benchmarks

Ext4 vs...

Ext4 vs... Ext4

- Ext4 was created for originally rotating block storage. Couldn't find any `mkfs.ext4` option which could help on solid state storage.
- `mkfs.ext4`'s default block size, 4K, is a good value according to our raw write tests.
- Did you know? By default, this command will automatically select suitable `inode_ratio`, `block_size` and `inode_size` values according to the size of the filesystem. You can override this with the `-T` option and profiles in the `/etc/mke2fs.conf` file.

```
...
[fs_types]
        ...
        small = {
                inode_ratio = 4096
        }
        floppy = {
                inode_ratio = 8192
        }
        big = {
                inode_ratio = 32768
        }
        huge = {
                inode_ratio = 65536
        }
        ...
        largefile4 = {
                inode_ratio = 4194304
                blocksize = -1
        }
        hurd = {
             blocksize = 4096
             inode_size = 128
             warn_y2038_dates = 0
        }
```

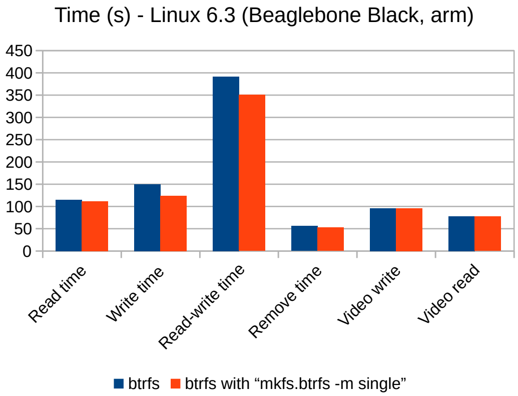
Profiles in `/etc/mke2fs.conf`

XFS vs XFS

- Didn't find options which could increase performance on Solid State storage.
- By default, `mkfs.xfs` picks a 4K block size, which is good with SD cards.

Btrfs vs Btrfs

- By default, `mkfs.btrfs` picks a 4K block size, which is good with SD cards.
- `mkfs.btrfs` does detect an SSD when given an SD card partition.
- In past versions, `mkfs.btrfs` was then creating the filesystem with the `-m single`. This corresponds to not duplicating metadata to two physical locations (`-m dup`).
- However, this information is not trustworthy and we now always have `-m dup` by default.
- See DUP profiles on a single device.



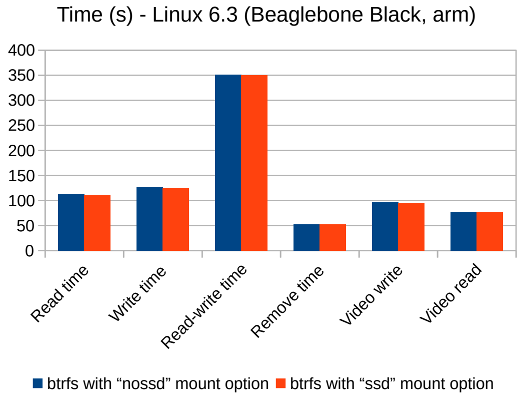
`mkfs.btrfs -m single` reduces space (-3.5%) and increases read and write performance. Keeping this option!

Btrfs ssd options

- Btrfs offers SSD related mount options: `ssd`, `ssd_spread` and `nossd`.
- `ssd` is enabled by default if your storage is detected as SSD:

```
[ 915.149865] BTRFS info (device
mmcblk0p3): enabling ssd
optimizations
```

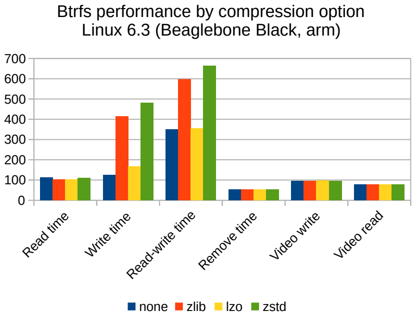
  Otherwise (USB mass storage), you may need to set it manually.
- Didn't manage to use `ssd_spread`, was causing `No space left on device` errors even though there were plenty of space.
- See https://btrfs.readthedocs.io/en/latest/ch-mount-options.html.

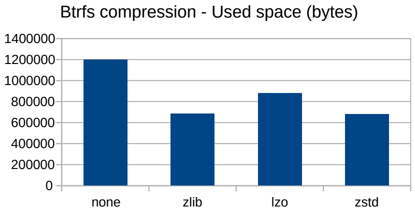


The `ssd` mount option brings minor read and write speedups ($<$ 1%).

Btrfs compression options

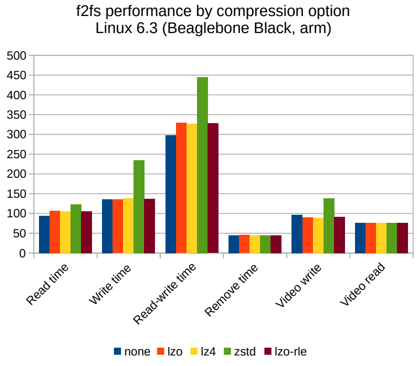
- Btrfs also offers mount options for compression: `compress=none|zlib|lzo|zstd` (default: `none`).
- See mount options.
- Compression helps a bit with read time (- 10% at best), but significantly hurts write time (though acceptably with `lzo`), at least on the 1GHz CPU we tested.
- Compression is smart: giving up if file contents turn out not to be compressible (e.g. video).
- Sticking to `none`, otherwise would use `lzo`, or `zlib` if the priority is the compression rate.
- Here, what you choose will depend on your system, how often you write, what kind of files...





F2FS vs F2FS

- F2FS compression options are not very straightforward to use. They allow for selective compression of files (specific files, files with a given extension). However, it's possible to try to compress all files.
- Need to create the filesystem as follows:
  `mkfs.f2fs -O compression,extra_attr`
- Need to mount the filesystem as follows:
  `-o compress_extension=*,compress_algorithm=lzo|lz4|zstd|lzo-rle`
- Also tried the `lazytime` and `atgc,gc_merge` mount options, but they didn't help with performance, except marginally for video writing.
- See https://www.kernel.org/doc/html/latest/filesystems/f2fs.html and https://wiki.archlinux.org/title/F2FS.



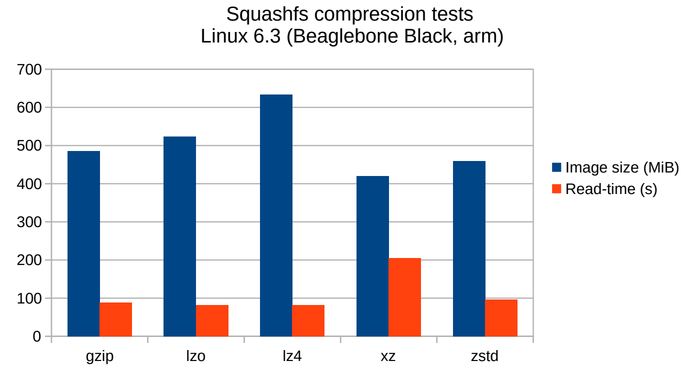
Compression can be useful, but here there are no sizable performance benefits. Keeping the uncompressed version.

nilfs2 vs nilfs2

- Didn't see any `mount` option which could help with performance.
- See https://www.kernel.org/doc/html/latest/filesystems/nilfs2.html
- `mkfs.nilfs2` options:
  - `-b` (block-size) already has a good default value (4K)
  - `-B` (blocks-per-segment) too (8M)

SquashFS vs SquashFS

Comparing the read performance of the various compression options (using a Raspberry Pi OS Lite root filesystem, 1.2 GB, ARM binaries).



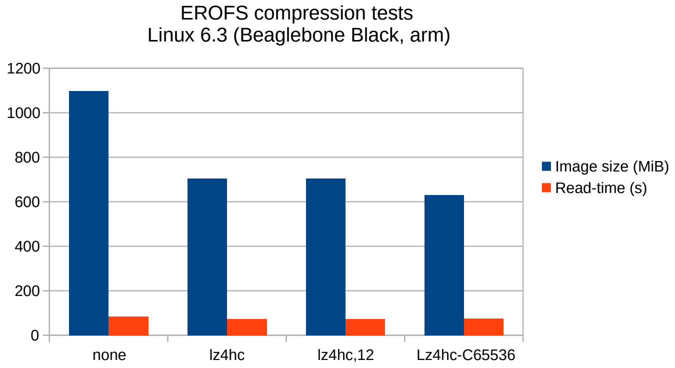
`lzo` looks like the best compromise in terms of speed and space. We will use SquashFS with this compression scheme.

Notes: tried several promising non-default kernel configuration options:

- `SQUASHFS_4K_DEVBLK_SIZE`: enforcing 4K blocks. This slightly degraded performance.
- `SQUASHFS_FILE_DIRECT`: directly decompressing to the file cache. Didn't see any noticeable impact on read performance.
- Haven't tried the multi-threaded decompression options yet.

EROFS vs EROFS (1)

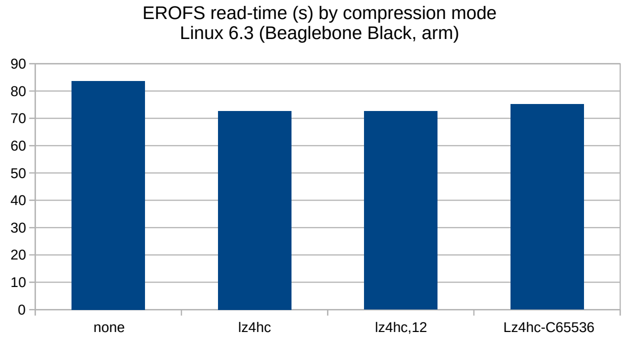
Comparing the read performance of the various compression options (using an Raspberry Pi OS Lite root filesystem (1.2 GB, ARM binaries).



We tried 4 options to create an EROFS image with `mkfs.erofs`:

- No option: no compression
- `-zlz4hc`: compresses with the *MicroLZMA* compressor (Linux 5.16+)
- `-zlz4hc,12`: compresses with the *MicroLZMA* compressor, best compression
- `-C65536`: adding *big pcluster* feature for bigger clusters (Linux 5.13+)

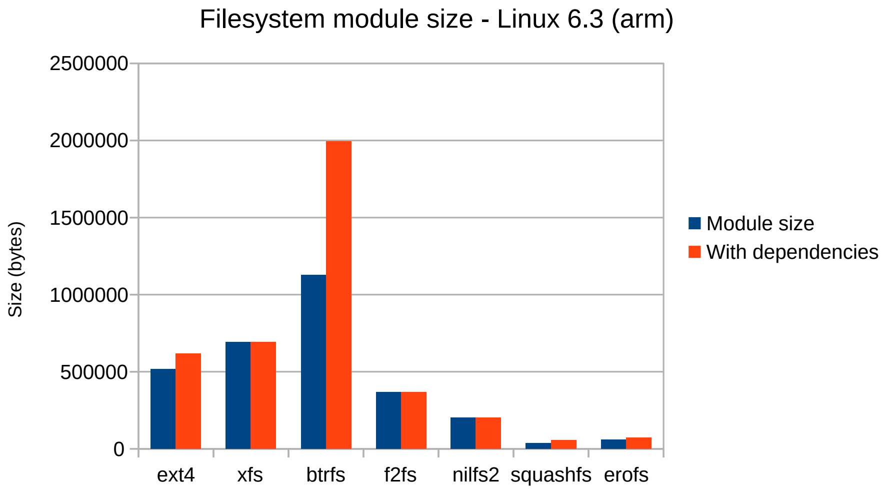
EROFS vs EROFS (2)



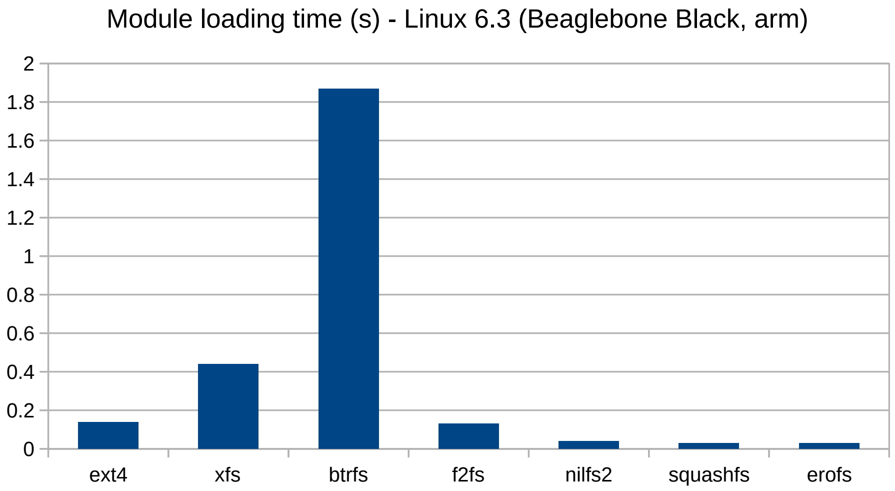
- We will continue tests with `-zlz4hc,12`, giving the best time results.
- There just seems to be a penalty on the host machine creating the image!

Filesystem benchmarks - Real comparisons

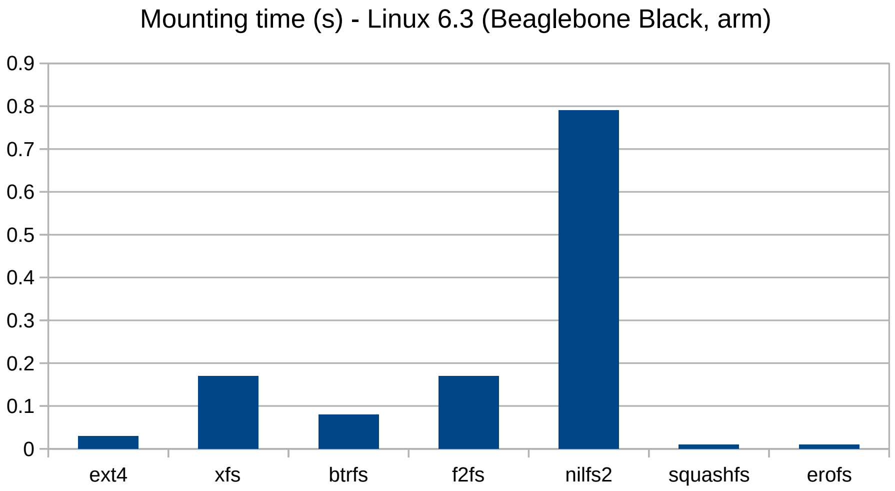
Kernel module size



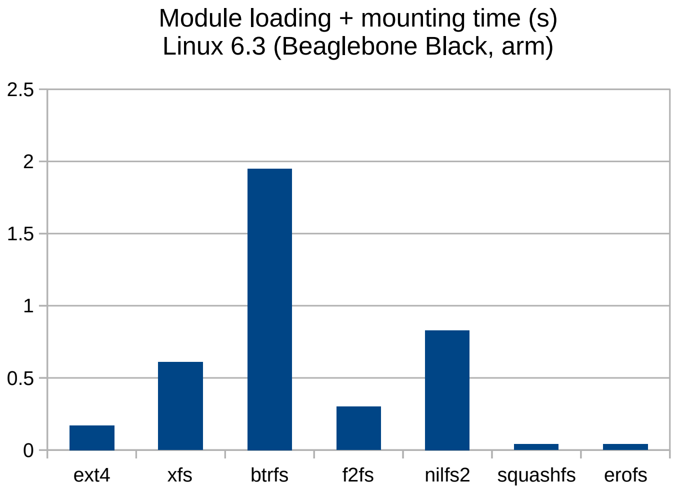
Module loading time



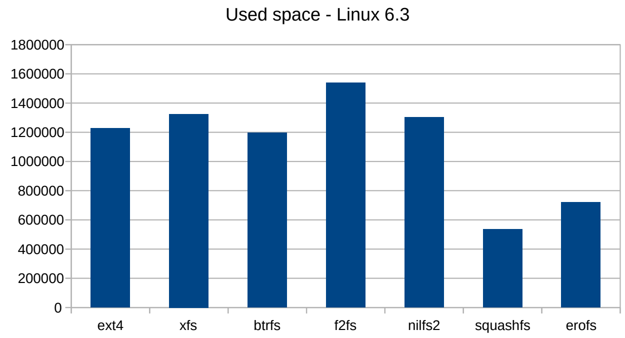
Mounting time



Boot time: module loading and mounting time

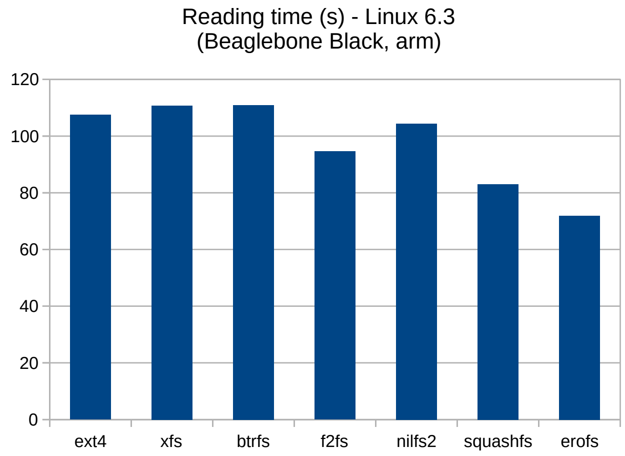


Used space



Contents: Raspberry Pi OS Lite root filesystem, 1.2 GB, ARM binaries.

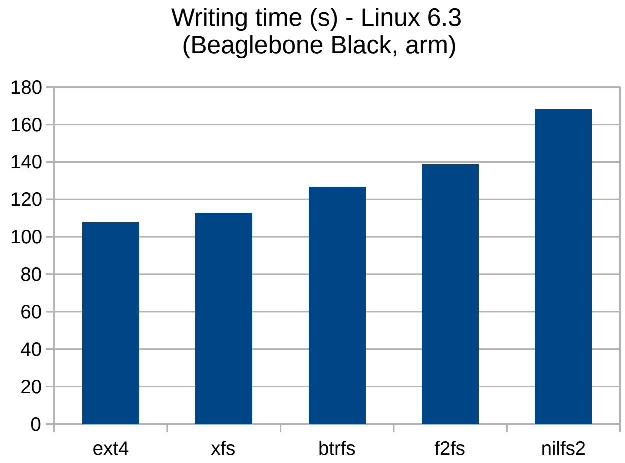
Reading time



Reading all the files in the filesystem (contents of a Raspberry Pi OS Lite root filesystem, 1.2 GB, ARM binaries).

```
#!/bin/sh
tar cf /dev/null /mnt/data
```

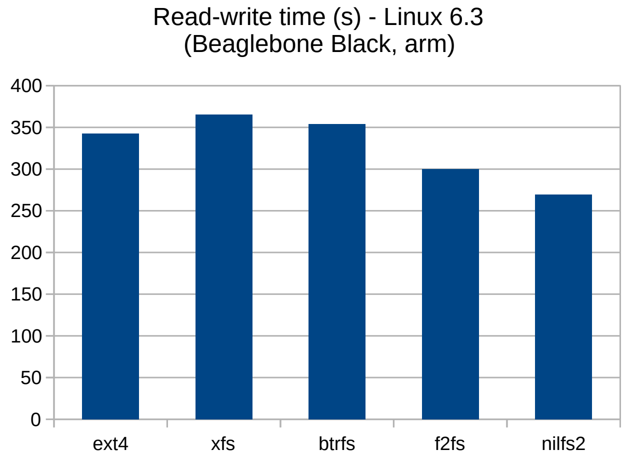
Writing time



Copying a Debian ARM root filesystem (319M) 5 times from a pre-loaded ramfs to the target filesystem.

```
#!/bin/sh
num=5
for i in `seq $num`
do
    cp -r /mnt/ramfs/debian-arm /mnt/data/$i
    echo "Copying $i / $num"
done
sync
```

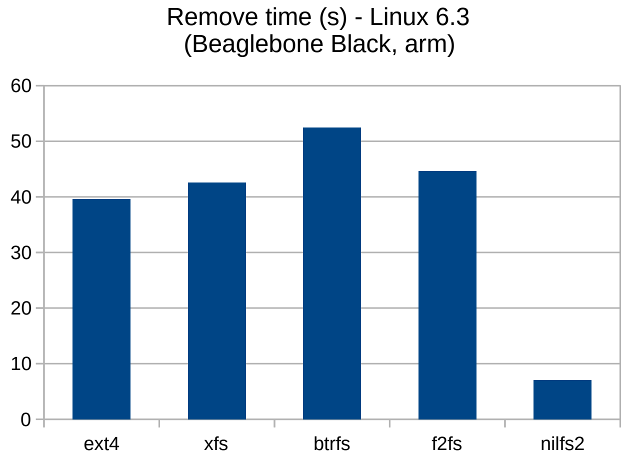
Reading, removing and writing time



Continued from the previous tests. Remove 1 directory out of 5 (Debian ARM root filesystem, 319M), copy the oldest remaining one to a new one, all this 5 times.

```
#!/bin/sh
num=5
i=1
while [ $i -le $num ]
do
    echo "Loop $i..."
    rm -rf /mnt/data/$i
    i=`expr $i + 1`
    cp -r /mnt/data/$i /mnt/data/`expr $i + 4`
done
sync
```

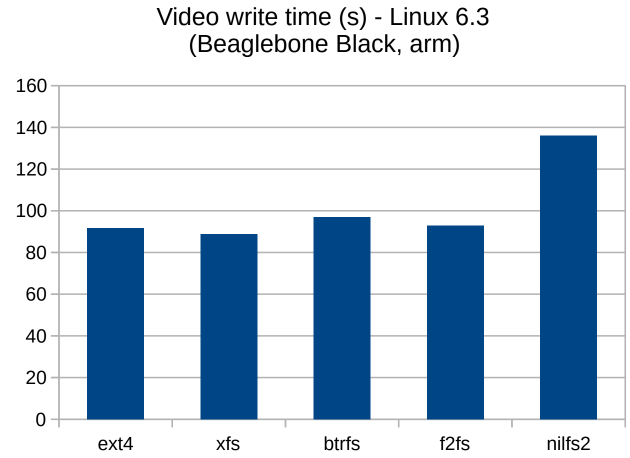
Removing time



Continued from the previous tests. Remove all 5 directories (Debian ARM root filesystem, 319M).

```
#!/bin/sh
/bin/rm -rf /mnt/data/*
sync
```

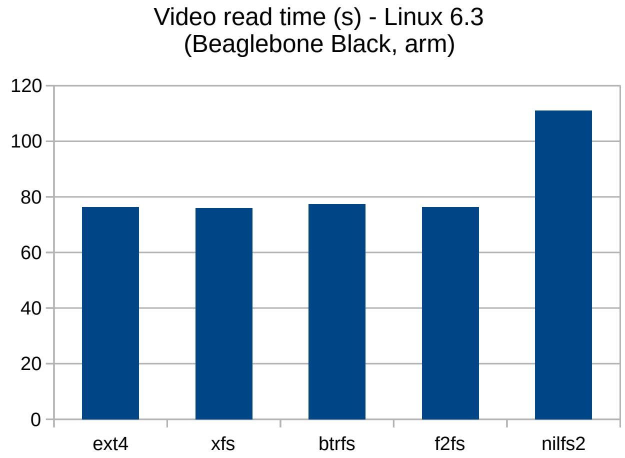
Sequential write time



New test. Copy a preloaded video (`big_buck_bunny_720p_surround.avi`, 317M) to the filesystem, 5 times.

```
#!/bin/sh
for i in `seq 5`
do
    cp /mnt/ramfs/video.avi /mnt/data/video.avi.$i
done
sync
```

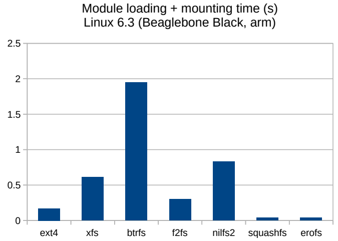
Sequential read time

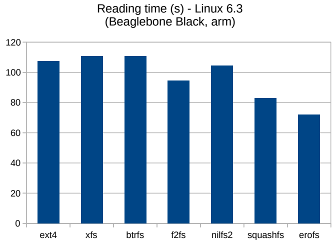


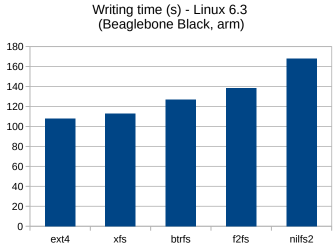
Continued from the previous test. After unmounting and remounting the filesystem, read the stored videos (`big_buck_bunny_720p_surround.avi`, 317M, 5 times)

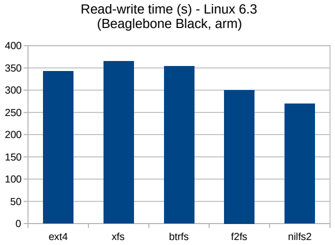
```
#!/bin/sh
cat /mnt/data/video.avi.* > /dev/null
```

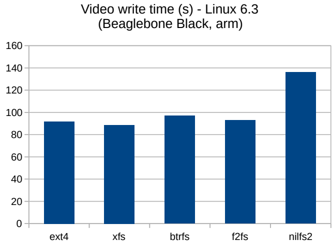
The BIG picture

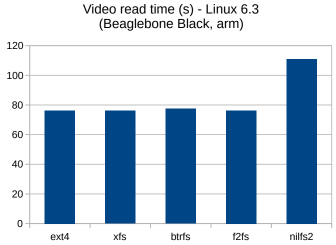












Our observations

	Boot time	Mount time	Read	Seq. read	Write	Seq. write	read write delete	Delete	Space
ext4	very good	very good	fair	good	best	very good	good	good	good
xfs	bad	average	fair	good	very good	best	average	fair	fair
btrfs	worst	good	fair	good	good	good	fair	worst	good
f2fs	fair	average	good	good	fair	very good	very good	average	worst
nilfs2	bad	worst	fair	worst	worst	worst	best	best	fair
squashfs	excel-lent	best	very good						best
erofs	best	best	best						very good

Caution: specific to our own tests!

What to remember

- Seems like Ext2 is going away (2038 limit)
- EROFS seems to be the fastest read-only filesystem as expected.
- SquashFS is great to minimize space while keeping very good read performance.
- Ext4 remains a very good default choice for read-write filesystem in all aspects.
- F2FS seems to be the second best choice.
- Btrfs turns out to be bulky, complicated and powerful, but created with `-m single`, is a solid choice too (except for boot time).
- XFS is a pretty good choice too, and easy to use.
- Compression doesn't seem to help with performance (at least with our rather slow, single-core CPU).
- Nilfs2 can give you great results, but also the worst ones, depending on your usage scenario.

Always try with your own hardware and applications!

Lessons learned

- Flashing an SD card with `dd`: the **page size** (usually 4K) is the best block size. Don't exceed 1M.
- Reading an SD card with `dd`: the **segment size** is one of the best block sizes. Bigger or smaller blocks degrade performance.
- Tried to use `flashbench -f` to look for special "FAT" segment with better performance. Didn't find anything noticeable.
- Anyway, for Journaling filesystems, the journal cannot fit in the first sectors,s as it is too big anyway. You may store it elsewhere though, if you have faster storage.

Limitations

- Haven't tested on ARM64 yet (faster CPU), and with multi-core CPUs either
- Haven't tested real random writing (modifying random files in place) and reading.
- Tests made only on one SD card, not on eMMC, and neither on USB nor on NVME, SATA...

Further resources

- Peter Chubb: SD cards and filesystems for embedded systems (2015)
  http://mirror.linux.org.au/pub/linux.conf.au/2015/Case_Room_2/Friday/SD_Cards_and_filesystems_for_Embedded_Systems.webm
- Richard Weinberger: EROFS vs. SquashFS: A Gentle Benchmark (2022)
  https://blog.sigma-star.at/post/2022/07/squashfs-erofs/

Questions? Suggestions? Comments?

Na shledanou!

Michael Opdenacker
michael.opdenacker@bootlin.com