Ecole d'ingénieurs et d'architectes de Fribourg
Hochschule für Technik und Architektur Freiburg

File Systems

Youtube: https://youtu.be/qkzU6uXFW80

Hes·so // FRIBOURG FREIBURG
Haute Ecole Spécialisée de Suisse occidentale
Fachhochschule Westschweiz

References

[1]: <Linux Kernel sources>/Documentation/filesystems

[2]: http://www.tldp.org/HOWTO/html_single/SquashFS-HOWTO

[3]: http://squashfs.sourceforge.net

[4]: tree.celinuxforum.org/CelfPubWiki/ELCEurope2008Presentations?action=AttachFile&do=get&target=squashfs-elce.pdf

[5]: http://superuser.com/questions/228657/which-linux-filesystem-works-best-with-ssd //File for SSD card

[6]: https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Storage_Administration_Guide/index.html // very good site

[7]: https://code.google.com/p/cryptsetup/

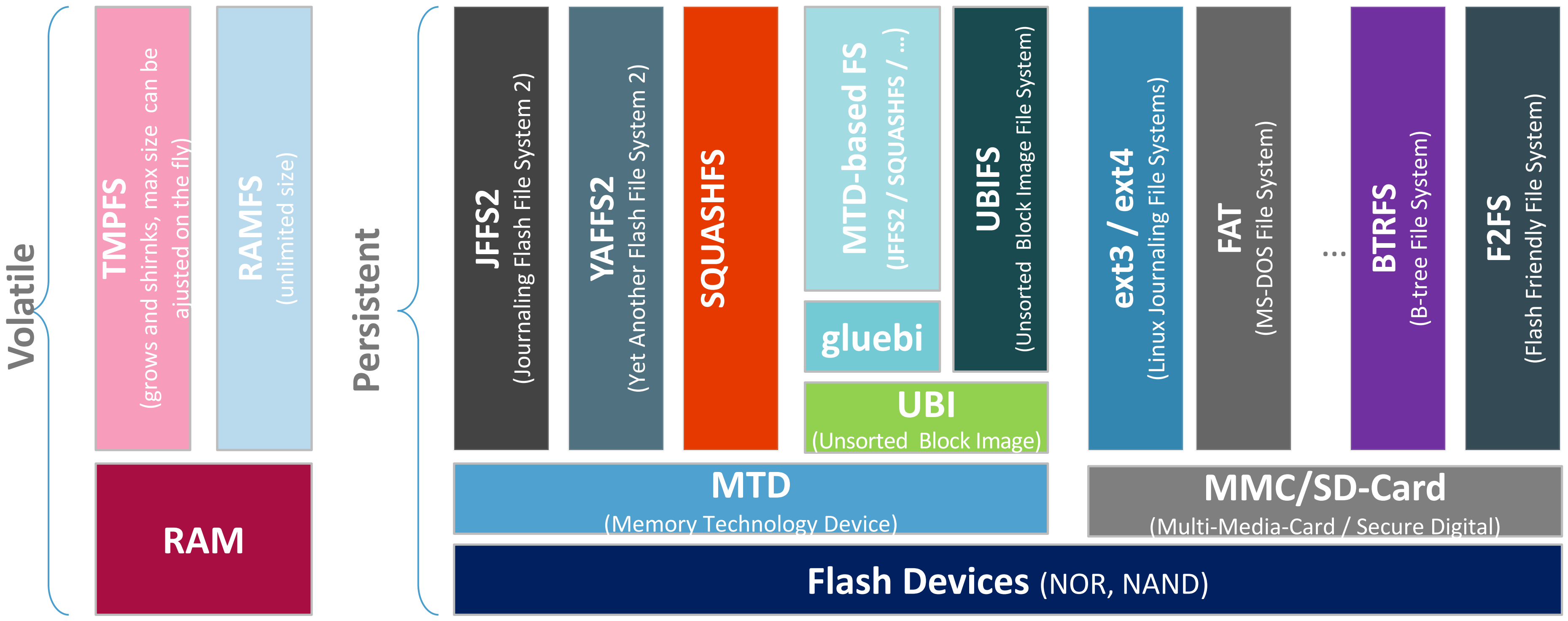
Power off embedded FS

[8]: http://stackoverflow.com/questions/14460091/embedded-file-system-and-power-off

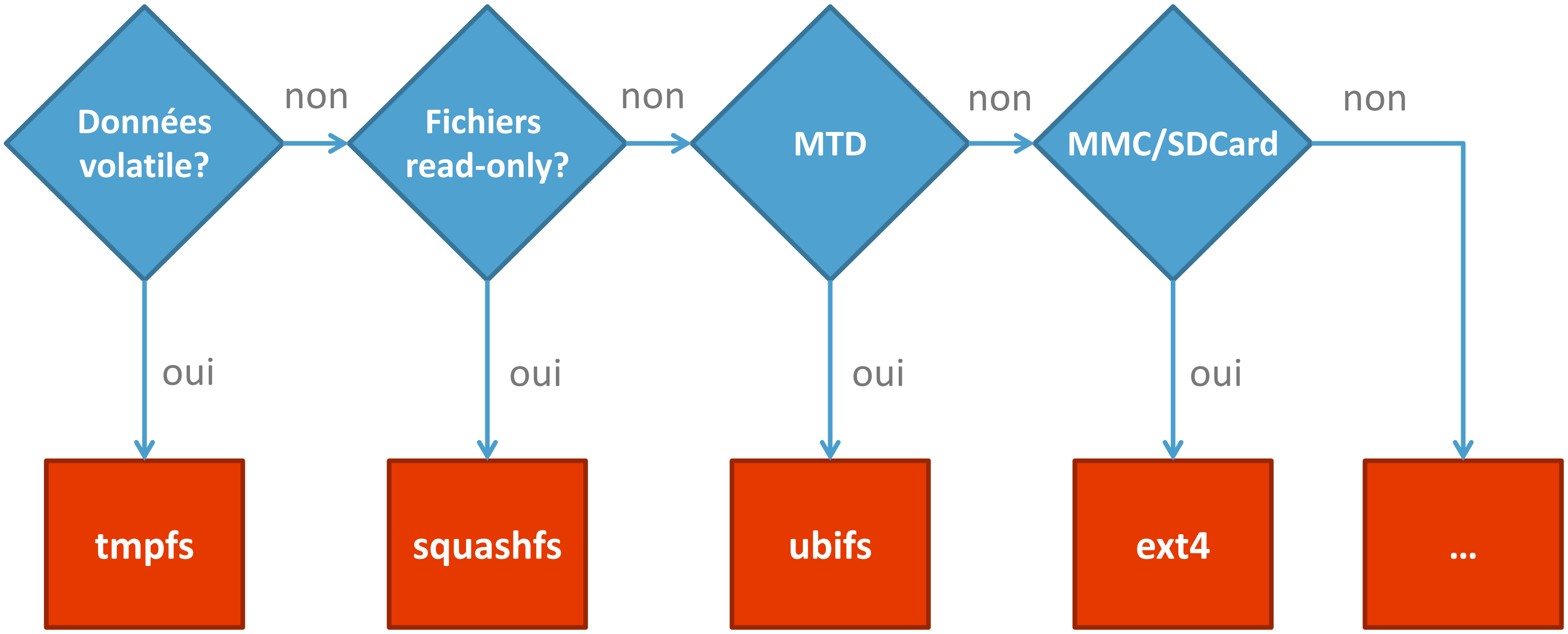
[9]: https://elinux.org/images/0/02/Filesystem_Considerations_for_Embedded_Devices.pdf

Systèmes de fichiers (Cours CSEL)

- Pour les systèmes embarqués, il existe deux catégories de systèmes de fichiers, les volatiles en RAM et les persitents sur des Flash (NOR et de plus en plus NAND).
- Deux technologies principales sont disponible sur les Flash, soit les MTD (Memory Technology Device) ou les MMC/SD-Card (Multi-Media-Card / Secure Digital Card).
- Il existe une multitude de systèmes de fichiers pour chacune de ces technologies.


Volatile
TMPFS (grows and shirnks, max size can be ajusted on the fly)
RAMFS (unlimited size)
RAM
Persistent
JFFS2 (Journaling Flash File System 2)
YAFFS2 (Yet Another Flash File System 2)
SQUASHFS
MTD-based FS (JFFS2 / SQUASHFS / ...)
gluebi
UBIFS (Unsorted Block Image File System)
UBI (Unsorted Block Image)
MTD (Memory Technology Device)
ext3 / ext4 (Linux Journaling File Systems)
FAT (MS-DOS File System)
...
BTRFS (B-tree File System)
F2FS (Flash Friendly File System)
MMC/SD-Card (Multi-Media-Card / Secure Digital)
Flash Devices (NOR, NAND)

Choix d'un système de fichiers (Cours CSEL)



Voir la documentation du noyau Linux pour plus de détails sur les systèmes de fichiers
`Documentation/filesystems/`

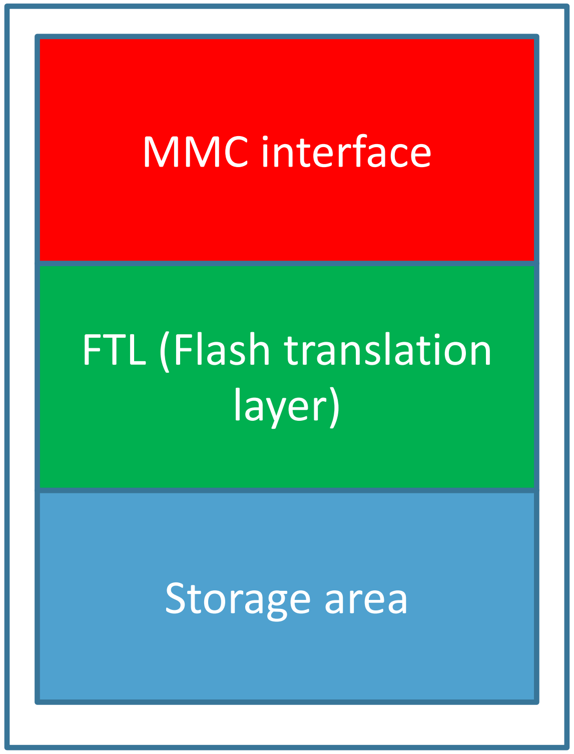
MMC technologies [9]

- **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.

Inside MMC technologies [9]

MMC-eMMC-SD Card is composed by 3 elements:

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



FTL is a small controller running a firmware. Its main purpose is to transform logical sector addressing into NAND addressing.

It also handles:

- Bad block management
- Garbage collection.
- Wear levelling

File systems, embedded systems

Embedded Systems can use different filesystems

- Ext4, ext3, ext2
- BTRFS
- F2FS
- NILFS2
- XFS
- (ZFS)

- Squashfs

- Tmpfs, Cramfs

- UBIFS
- JFFS2
- YAFFS2 .

Journalized file system [9]

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

- 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

Well known journalized filesystems:

- EXT3, EXT4
- XFS
- Reiser4

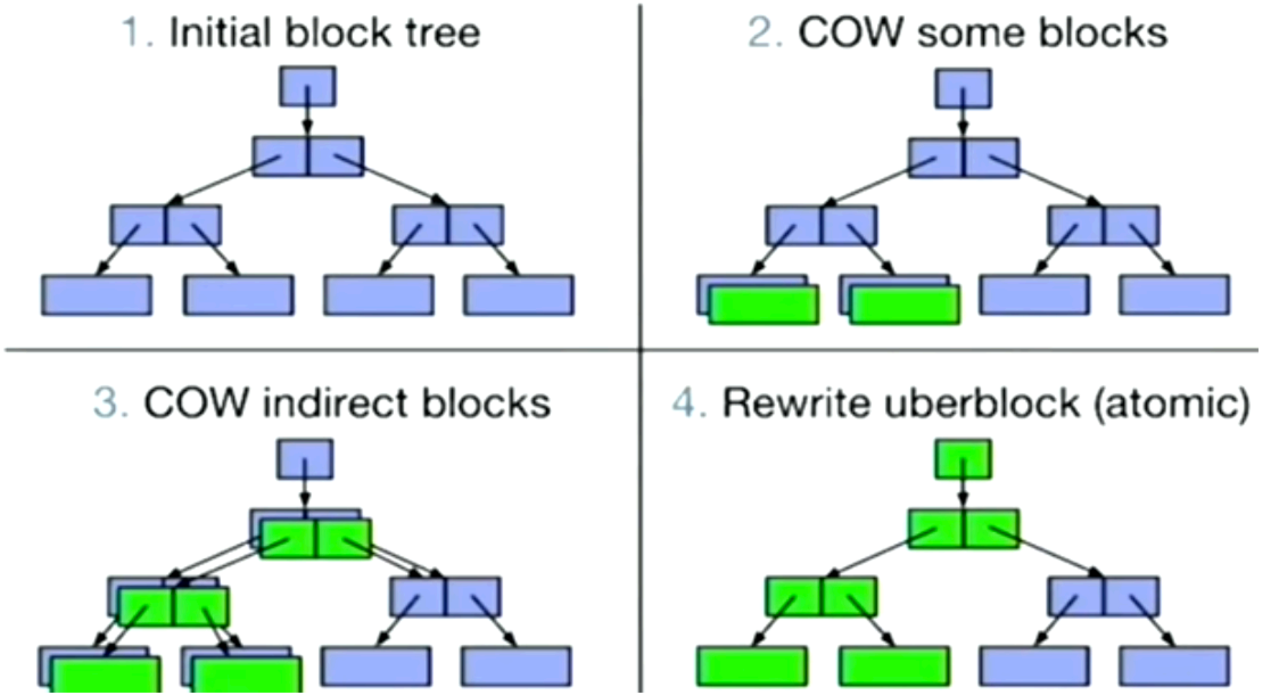
B-TREE/CoW [9]

- B+ tree is a data structure that generalized binary trees.
- CoW (Copy on Write) is used to ensure no corruption occurs at runtime:
  - The original storage is never modified. When a write request is made, data is written to a new storage area
  - Original storage is preserved until modification is fully done : transaction committed
  - If an interruption occurs during writing the new storage area, original storage can be used.

Well known filesystems using CoW:

- ZFS
- BTRFS
- NILFS2

B-tree filesystem [9]



Log filesystem [9]

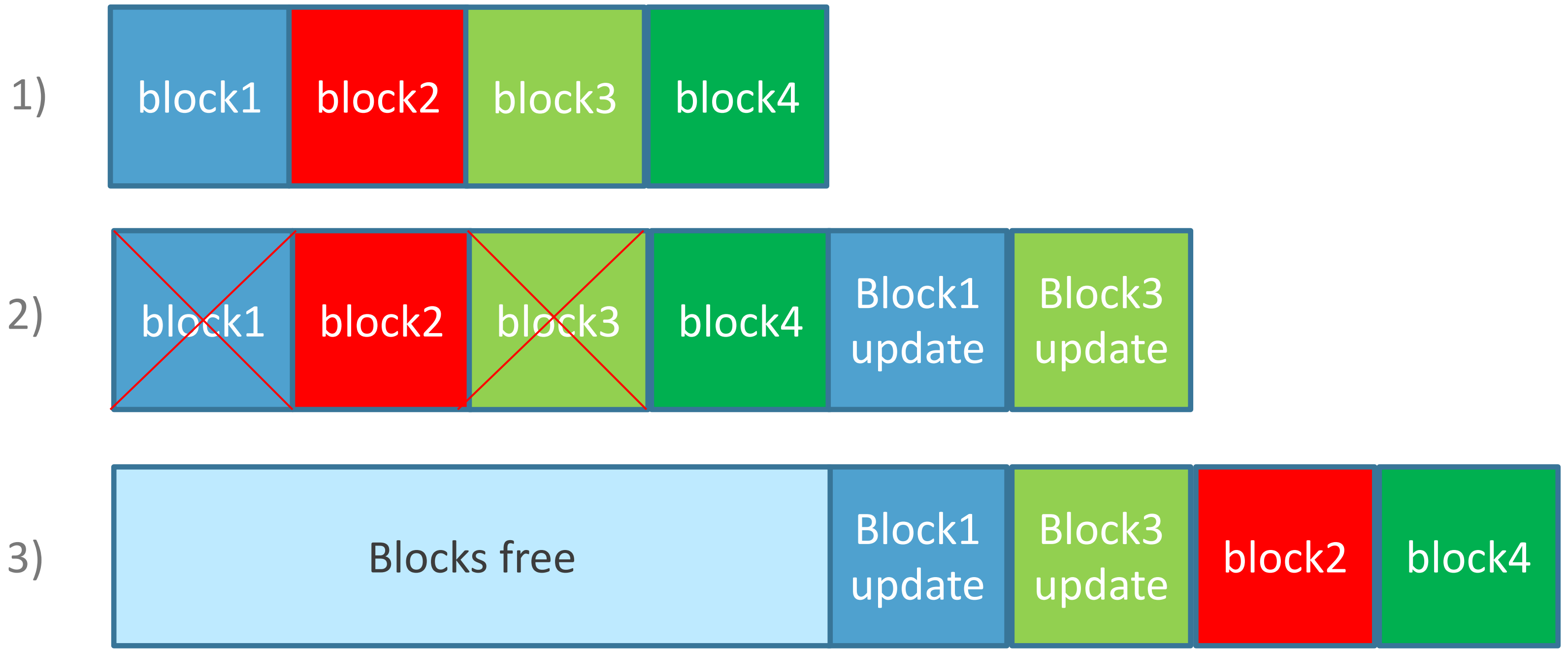
Log-structured filesystem use the storage medium as circular buffer and new blocks are always written to the end.

- Log-structured filesystems are often used for flash media since they will naturally perform wear-levelling
- The log-structured approach is a specific form of copy-on-write behavior

Well known log filesystems:

- F2FS
- NILFS2
- JFFS2
- UBIFS

Log filesystem [9]



1) Initial state
2) Block 1-3 are updated, old blocks 1-3 are not used
3) Garbage copies block2 and 4, and frees old block1-2-3-4

Ext2, ext3, ext4 file system [9]

[9]: “Filesystem considerations for embedded devices” is a good study about filesystems used on embedded systems

This file system is very used in different Linux distribution

- 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

BTRFS (B-Tree filesystem) [9]

- BTRFS is a “new” file system compared to EXT. It is originally created by Oracle in 2007. it is a B-Tree filesystem.
- It is considered stable since 2014
- Since 2015 BTRFS is the default rootfs for openSUSE.
- 2017: Red Hat decided to stop supporting BTRFS.
- BTRFS inspires from both Reiserfs and ZFS.
- Theodore Ts’o (ext3-ext4 main developer) said that BTRFS has a better direction than ext4 because "it offers improvements in scalability, reliability, and ease of management"

F2FS (Flash-Friendly File System) [9]

- 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 (reporting free blocks for reuse)

XFS(Flash-Friendly File System) [9]

XFS was developed by SGI in 1993.

- Added to Linux kernel in 2001
- On disk format updated in Linux version 3.10
- XFS is a journaling filesystem.
- Supports huge filesystems
- Designed for scalability
- Does not seem to be handling power loss (standby state) well

NILFS2 (New Implementation of a Log-structured File System) [9]

- Developed by Nippon Telegraph and Telephone Corporation
- NILFS2 Merged in Linux kernel version 2.6.30
- NILFS2 is a log filesystem.
- CoW for checkpoints and snapshots.
- Userspace garbage collector

ZFS (Zettabyte (10^{21})File System) [9]

ZFS is a combined file system and logical volume manager designed by Sun Microsystems

- ZFS is a B-Tree file system
- Provides strong data integrity
- Supports huge filesystems
- Not intended for embedded systems (requires RAM)
- License not compatible with Linux

Conclusions [9]

Performances:

- EXT4 is currently the best solution for embedded systems using MMC
- F2FS and NILFS2 show impressive write performances

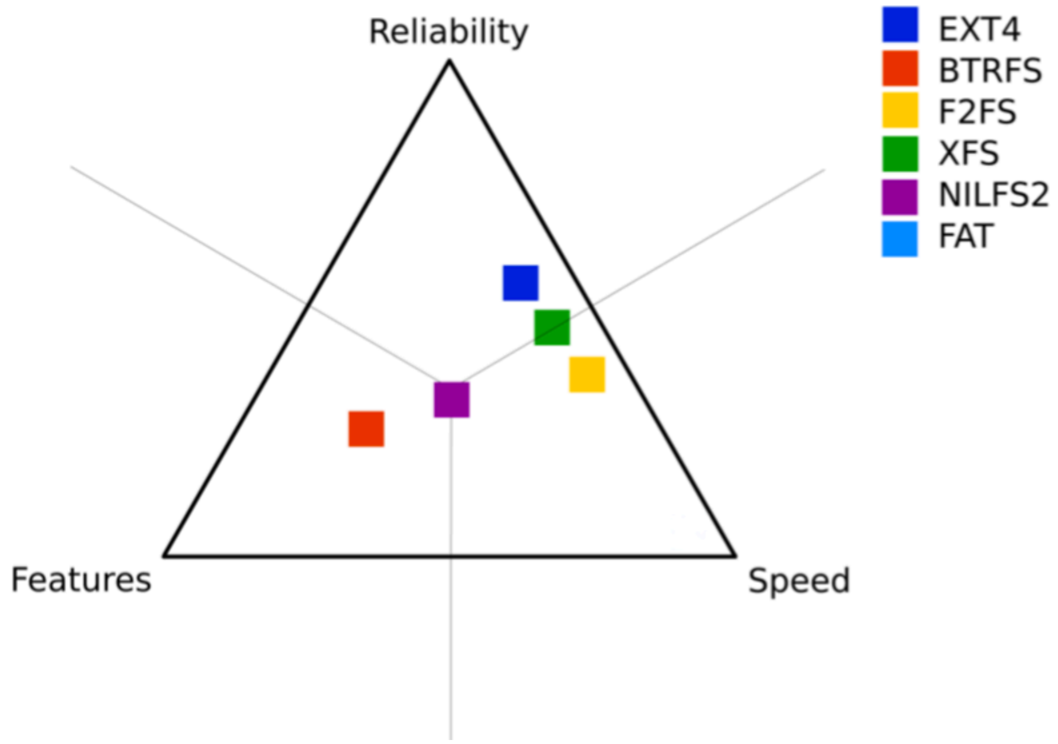
Features:

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

Scalability:

- EXT4 clearly doesn’t scale as well as BTRFS and F2FS

Conclusions [9]



Ext2, ext3, ext4 file systems

- **Ext2** or second extended filesystem is a file system for the Linux kernel.
  - Ext2 is not a journaled file system
  - Ext2 uses block mapping in order to reduce file fragmentation (it allocates several free blocks → reduce fragmentation).
  - After an unexpected power failure or system crash (also called an unclean system shutdown), each mounted ext2 file system on the machine must be checked for consistency with the **e2fsck** program.

Ext2, ext3, ext4 file systems

- **Ext3** file system replaces ext2
  - It was merged in the 2.4.15 kernel on November 2001.
  - Ext3 is compatible with ext2.
  - Ext3 is a journaled file system
  - The ext3 file system prevents loss of data integrity in the event that an unclean system shutdown occurs.

Ext2, ext3, ext4 file systems

Ext4 file system

- ext4 is backward compatible with ext3 and ext2, making it possible to mount ext3 and ext2 as ext4
- Ext4 is included in the kernel 2.6.28 on 11 October 2008
- Ext4 supports Large file system:
  - volume max: 2^{60} bytes
  - File max: 2^{40} bytes
- Ext4 uses extents (as opposed to the traditional block mapping scheme used by ext2 and ext3), which improves performance when using large files and reduces metadata overhead for large files.

Ext4 commands

# Create a partition (rootfs), start 64MB, length 256MB

```
sudo parted /dev/sdb mkpart primary ext4 131072s  655359s
```

# Format the partition with the volume label = rootfs

```
sudo mkfs.ext4 /dev/sdb1 -L rootfs
```

# Modify (on the fly) the ext4 configuration

```
sudo tune2fs <options> /dev/sdb1
```

# check the ext4 configuration

```
mount
sudo tune2fs -l /dev/sdb1
sudo dumpe2fs /dev/sdb1
```

# mount an ext4 file system

```
mount –t ext4 /dev/sdb1  /mnt/test    // with default options
mount –t ext4 –o defaults,noatime,discard,nodiratime,data=writeback,acl,user_xattr
/dev/sdb1    /mnt/test
```

Ext4, ext2 – buildroot, busybox

In order to have mkfs.ext2 and tune2fs program on the NanoPi, it is necessary to configure busybox

```
cd workspace/nano/buildroot
make busybox-menuconfig
```

Go to "Linux Ext2 FS Progs" → [*] tune2fs

Go to "Linux System Utilities" → [*] mkfs.ext2

Ext4 mount options and MMC/SD-Card

- filesystem options can be activated with the mount command (or to the /etc/fstab file).

- These options can be modified with tune2fs command

- MMC/SD-Card constraints: In order to improve the longevity of MMC/SD-Card, it is necessary to reduce the unnecessary writes

- Journaling: the journaling guarantees the data consistency, but it reduces the file system performances

- Mount options to reduce the unnecessary writes:
  - noatime: Do not update inode access times on this filesystem (e.g., for faster access on the news spool to speed up news servers)
  - nodiratime: Do not update directory inode access times on this filesystem
  - relatime: this option can replace the noatime and nodiratime if an application needs the access time information (like mutt)

Ext4 mount options and MMC/SD-Card

Mount options for the journaling:

- Data=journal: All data is committed into the journal prior to being written into the main filesystem (It is the safest option in terms of data integrity and reliability, though maybe not so much for performance

- Data=ordered: This is the default mode. All data is forced directly out to the main file system before the metadata being committed to the journal

- Data=writeback: Data ordering is not preserved - data may be written into the main filesystem after its metadata has been committed to the journal. This is *rumoured* to be the highest throughput option. It guarantees internal filesystem integrity, however it can allow old data to appear in files after a crash and journal recovery.

- Barrier=1: Write barriers enforce proper on-disk ordering of journal commits, making volatile disk write caches safe to use, at some performance penalty.

Ext4 mount options and MMC/SD_SDCard

- Discard: Use discard requests to inform the storage that a given range of blocks is no longer in use. A MMC/SD-Card can use this information to free up space internally, using the free blocks for wear-levelling.

- acl: Support POSIX Access Control Lists

- user_xattr: Support "user." extended attributes

- default: rw, suid, dev, exec, auto, nouser, and async
  - rw: read-write
  - suid: Allow set-user-identifier or set-group-identifier bits
  - dev: Interpret character or block special devices on the filesystem
  - exec: Permit execution of binaries
  - auto: Can be mounted with the -a option (mount –a)
  - nouser: Forbid an ordinary (i.e., non-root) user to mount the filesystem
  - async: All I/O to the filesystem should be done asynchronously

/etc/fstab file

- File /etc/fstab contains descriptive information about the filesystems the system can mount

- **NanoPi example: /etc/fstab**

```
# cat /etc/fstab
# /etc/fstab: static file system information.
#
# <file system> <mount pt>     <type>   <options>                     <dump> <pass>
/dev/root       /              ext4     rw,noauto                          0     1
proc            /proc          proc     defaults                           0     0
devpts          /dev/pts       devpts   defaults,gid=5,mode=620            0     0
tmpfs           /dev/shm       tmpfs    mode=0777                          0     0
tmpfs           /tmp           tmpfs    defaults,nosuid,noexec,nodev,rw    0     0
sysfs           /sys           sysfs    defaults                           0     0
```

/etc/fstab file

<file system>: block special device or remote filesystem to be mounted

<mount pt>: mount point for the filesystem

<type>: the filesystem type

<options>: mount options associated with the filesystem

<dump>: used by the dump (backup filesystem) command to determine whichfilesystems need to be dumped (0 -> no backup).

<pass>: used by the fsck (8) program to determine the order in which filesystem checks are done at reboot time. The root filesystem should be specified with 1, and other filesystems should have a 2. if <pass> is not present or equal 0 -> fsck willassume that the filesystem is not checked.

Field options: It contains at least the type of mount plus any additional options appropriate to the filesystem type.

Common for all types of file system are the options (man mount):

auto	Can be mounted with the -a option (mount -a)
defaults	Use default options: rw, suid, dev, exec, auto, nouser, and async.
nosuid	Do not allow set-user-identifier or set-group-identifier bits to take effect.
noexec	Do not allow direct execution of any binaries on the mounted file system
nodev	Do not interpret character or block special devices on the file system.

I/O Scheduler for block devices

- You can find information here: <Linux kernel source>/Documentation/block

- It is possible to change the I/O scheduler for the block devices
- It exists 3 scheduler rules: noop, deadline, cfq, the default is cfq

```
cat /sys/block/XXXX/queue/scheduler                 // check the scheduler
echo “noop” >  /sys/block/XXXX/queue/scheduler      // change to noop scheduler
dd if=/dev/zero of=test bs=1M count=200             // copy 200MB to the file “test”
```

Scheduler	Time for 200MB
noop	2.80s
deadline	2.80s
cfq	3.05s

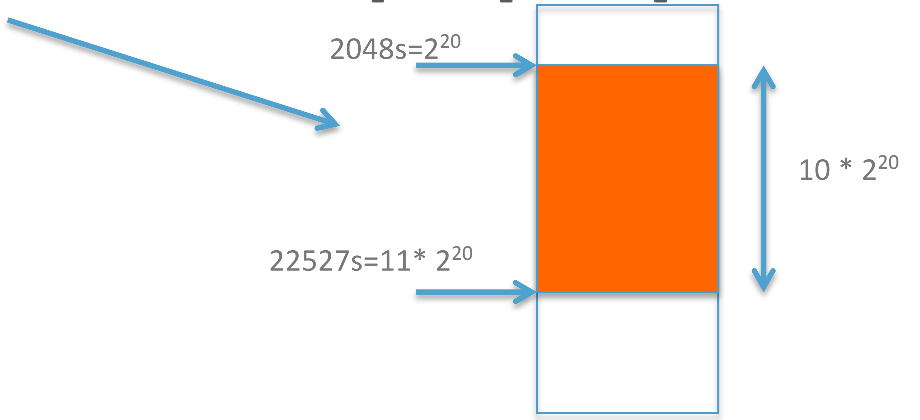
MMC/SD-Card partition alignment

- Partition alignment is critical for MMC/SD-Card as, being memory-based devices, data is written and read in blocks known as pages. When partitions aren't aligned, the block size of filesystem writes isn't aligned to the block size of the MMC/SD-Card

- To easily guarantee proper data alignment, **the starting sector of each partition must be a multiple of 2^{20}** (= 1'048'576) Bytes

- E.g. for sdb (the sector size is generally 512 bytes)
  - `# sudo fdisk /dev/sdb` //press n for a new partition, p for primary and enter a start sector of at least 2048 (2048 * 512 = 1'048'576)

  Or
  - `sudo parted /dev/sdb mkpart primary ext4 2048s 22527s` // s=1 sector = 512 bytes



F2FS

On PC

- Create new partition with fdisk or parted commands
- `sudo dnf install f2fs-tools.x86_64` // install tools for f2fs
- `sudo mkfs.f2fs -l usr_f2fs /dev/sdb3` // format partition 3

On NanoPi

- `mount /dev/mmcblk0p3 –t f2fs /mnt`
- `Or with /etc/fstab`
  - `#/dev/mmcblk0p3 /mnt        f2fs    defaults        0        0`

BTRFS

On PC

- Create new partition with fdisk or parted commands
- `sudo dnf install btrfs-progs.x86_64` // install tools for btrfs
- `sudo mkfs.btrfs /dev/sdb3` // format partition 3
- `sudo btrfs filesystem label /deb/sdb3 usr_btrfs`

On NanoPi

- `mount /dev/mmcblk0p3 –t btrfs /mnt`
- Or with /etc/fstab
  - `#/dev/mmcblk0p3 /mnt      btrfs    defaults      0      0`

NILFS2

On PC

- Create new partition with fdisk or parted commands
- `sudo dnf install nilfs-utils.x86_64` // install tools for btrfs
- `sudo mkfs.nilfs2 -L usr_nilfs /dev/sdb3` // format partition 3

On NanoPi

- `mount /dev/mmcblk0p3 –t nilfs2 /mnt`
- `Or with /etc/fstab`
  - `#/dev/mmcblk0p3 /mnt        nilfs2    defaults        0        0`

XFS

On PC

- Create new partition with fdisk or parted commands
- `sudo dnf install xfsprogs.x86_64` // install tools for btrfs
- `sudo mkfs.xfs -f -L usr_xfs /dev/sdb3` // format partition 3

On NanoPi

- `mount /dev/mmcblk0p4 -t xfs /mnt/xfs/`
- `Or with /etc/fstab`
  - `#/dev/mmcblk0p3 /mnt         xfs    defaults          0         0`

Squash File System [1], [2], [3], [4]

- Squashfs is a compressed read-only filesystem for Linux.
- Squashfs versions:
  - Squashfs 2.0 and squashfs 2.1: 2004, kernel 2.2
  - Squashfs 3.0: 2006, kernel 2.6.12
  - Squashfs 4.2: 2011, kernel 2.6.29
- It uses gzip, lzma, lzo, lz4 and xz compression to compress files, inodes and directories.
- SquashFS 4.0 supports 64 bit filesystems and files (larger than 4GB), full uid/gid information, hard links and timestamps.
- Squashfs is intended for general read-only filesystem use, for archival use, and in embedded systems with small processors where low overhead is needed

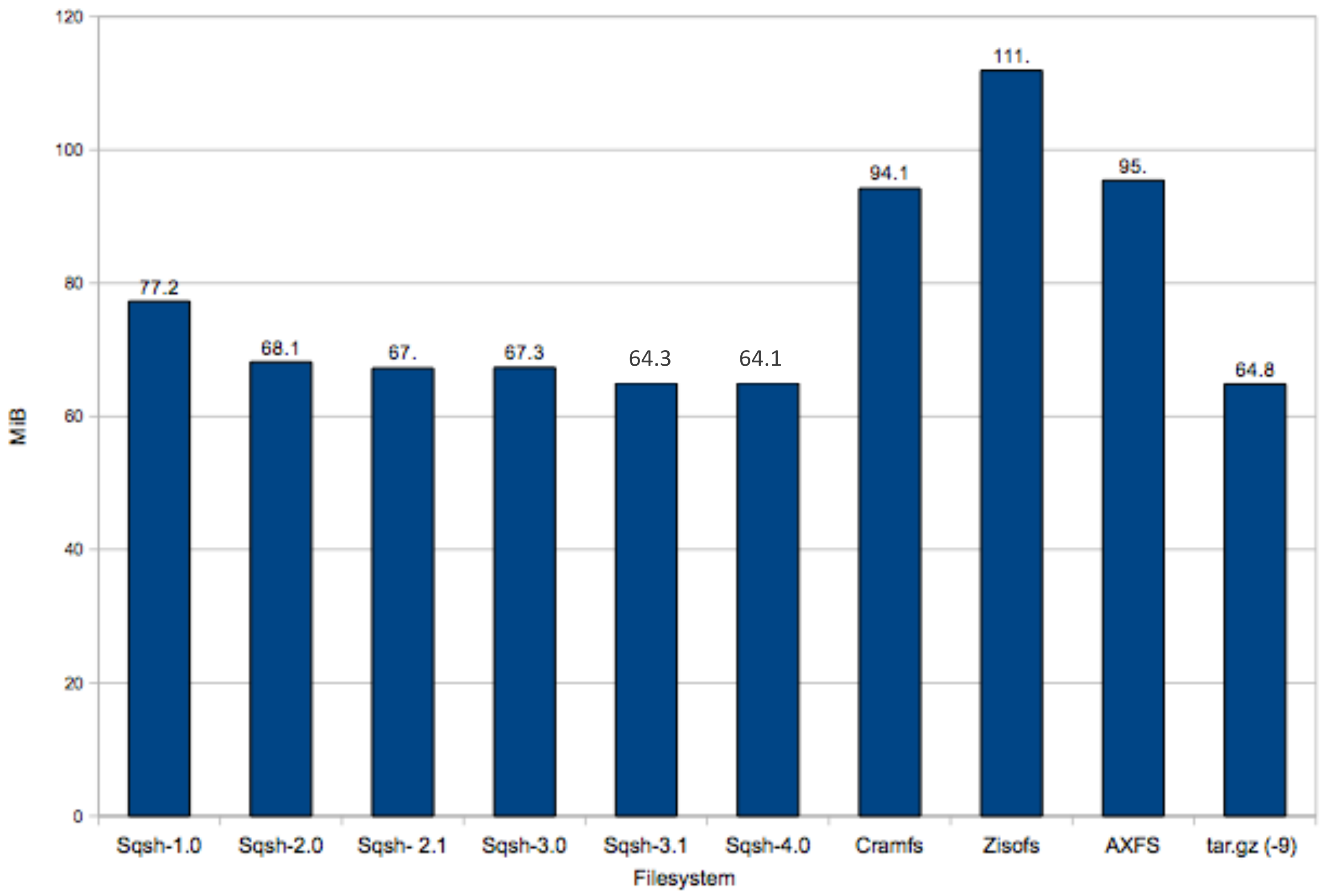
Squashfs vs Cramfs [1]

Squashfs filesystem features versus Cramfs:

	Squashfs	Cramfs
Max filesystem size:	2^{64}	256 MiB
Max file size:	~ 2 TiB	16 MiB
Max files:	unlimited	unlimited
Max directories:	unlimited	unlimited
Max entries per directory:	unlimited	unlimited
Max block size:	1 MiB	4 KiB
Metadata compression:	yes	no
Directory indexes:	yes	no
Sparse file support:	yes	no
Tail-end packing (fragments):	yes	no
Exportable (NFS etc.):	yes	no
Hard link support:	yes	no
"." and ".." in readdir:	yes	no
Real inode numbers:	yes	no
32-bit uids/gids:	yes	no
File creation time:	yes	no
Xattr support:	yes	no
ACL support:	no	no

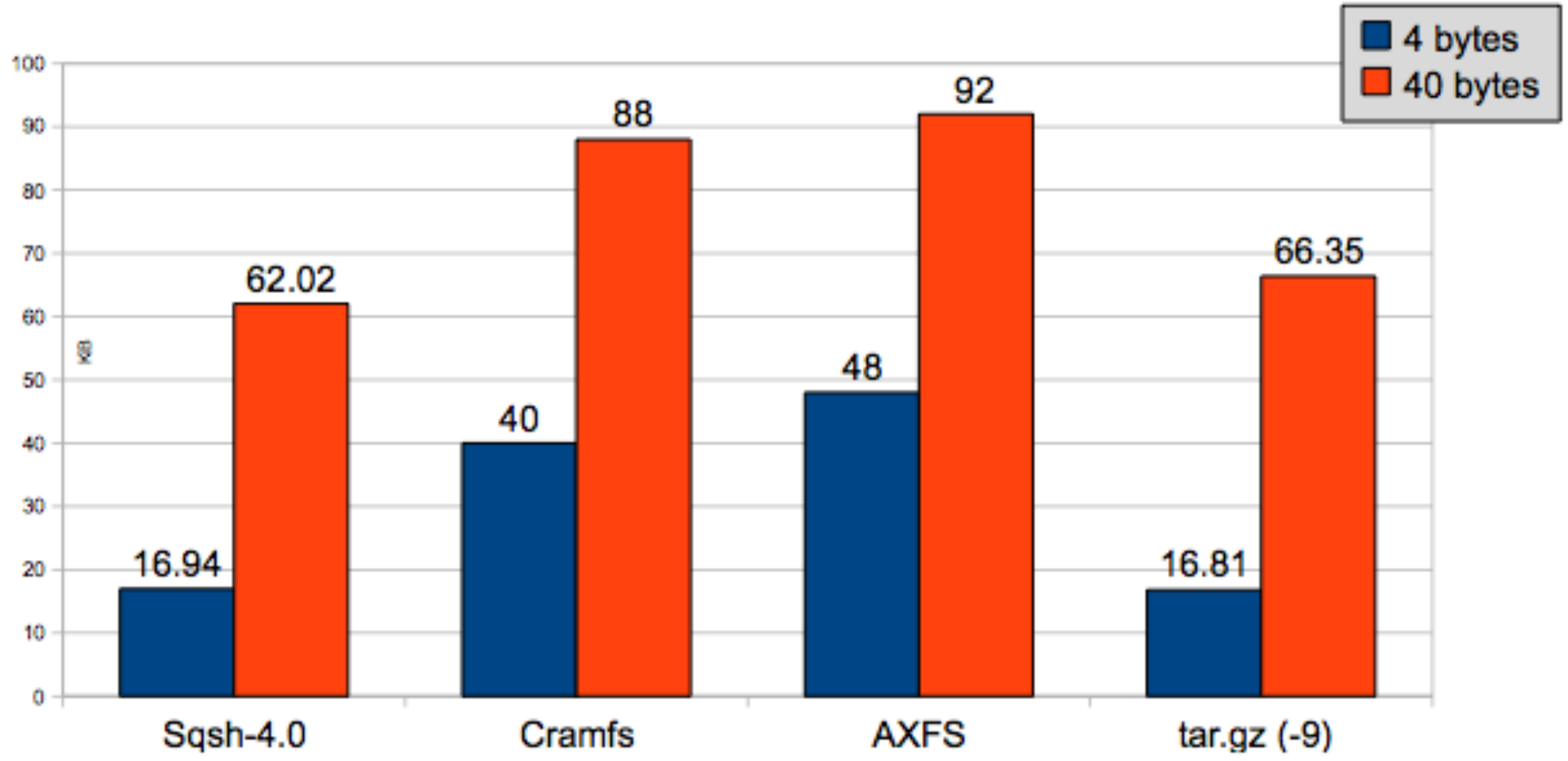
Squashfs compression [1]

File size with different compression techniques



Squashfs compression [1]

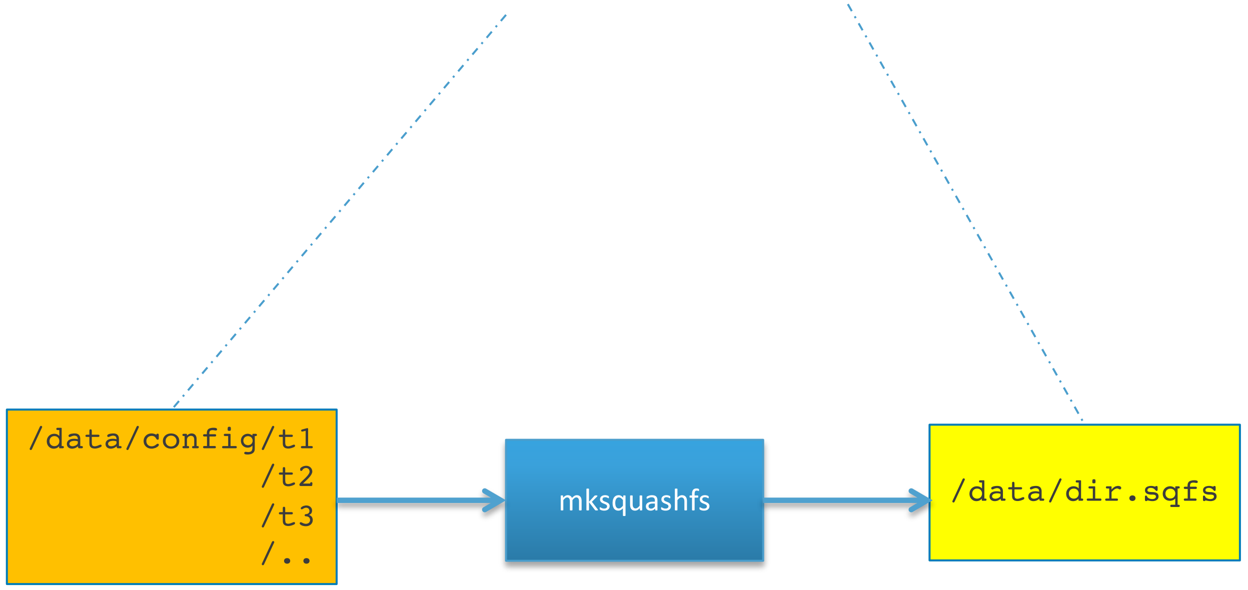
1200 small files size with different compression techniques



Create and use squashed file systems [2]

1. Create the squashed file system `dir.sqsh` for the regular directory `/data/config/`

```
bash# mksquashfs /data/config/ /data/dir.sqsh
```



Create and use squashed file systems [2]

2. The mount command is used with a loopback device in order to read the squashed file system `dir.sqsh`

```
bash# mkdir /mnt/dir
bash# mount –o loop –t squashfs /data/dir.sqsh /mnt/dir
bash# ls /mnt/dir
```

3. It is possible to copy the `dir.sqsh` to an unmounted partition (e.g. /dev/sdb2) with the `dd` command and next to mount the partition as squashfs file system

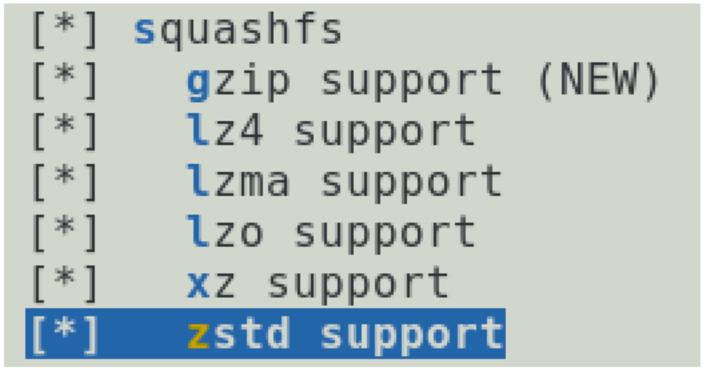
```
bash# umount /dev/sdb2
bash# dd if=dir.sqsh of=/dev/sdb2
bash# mount /dev/sdb2 /mnt/dir -t squashfs
bash# ls /mnt/dir
```

Squashfs - NanoPi - Buildroot

cd workspace/nanopi/buildroot

make menuconfig

- Go to: Target Packages → Filesystem and Flash utilities : choose squashfs



```
[*] squashfs
[*]   gzip support (NEW)
[*]   lz4 support
[*]   lzma support
[*]   lzo support
[*]   xz support
[*]   zstd support
```

Tmpfs [1]

- Tmpfs is a file system which keeps all files in virtual memory.
- Everything in tmpfs is temporary in the sense that no files will be created on your hard drive. If you unmount a tmpfs instance, everything stored therein is lost.
- tmpfs puts everything into the kernel internal caches and grows and shrinks to accommodate the files it contains and is able to swap unneeded pages out to swap space. It has maximum size limits which can be adjusted on the fly via 'mount -o remount ...'
- If you compare it to ramfs you gain swapping and limit checking. Another similar thing is the RAM disk (/dev/ram*), which simulates a fixed size hard disk in physical RAM, where you have to create an ordinary filesystem on top. Ramdisks cannot swap and you do not have the possibility to resize them

Tmpfs [1]

- glibc 2.2 and above expects tmpfs to be mounted at /dev/shm for POSIX shared memory (shm_open, shm_unlink). Adding the following line to /etc/fstab should take care of this:

```
tmpfs    /dev/shm          tmpfs    defaults          0  0
```

- It is very convenient to mount /tmp, /var/tmp as tmpfs file system. Add this line to /etc/fstab

```
tmpfs    /tmp             tmpfs     mode=1777         0  0
tmpfs    /var/tmp         tmpfs     mode=1777         0  0
```

Devtmpfs [1]

- devtmpfs is a file system with automatically populates nodes files (/dev/…) known by the kernel.
- This means you don't have to have udev running nor to create a static /dev layout with additional, unneeded and not present device nodes.
- Instead the kernel populates the appropriate information based on the known devices.
- The kernel executes this command: `mount -n -t devtmpfs devtmpfs /dev`
- `/dev` is automatically populated by the kernel with its known devices

```
# ls /dev
autofs             ptypf            tty47
btrfs-control      random           tty48
bus                rtc0             tty49
console            shm              tty5
cpu_dma_latency    snapshot         tty50
…
```