Filesystems in LiveCD

J. R. Okajima
2010/12

In the majority of cases, LiveCD uses Squashfs, tmpfs, and AUFS. This report compares usage of squashfs with others to implement LiveCD.

Ancient Age

A long long time ago, there was an approach to implement LiveCD by "shadow directory". It may be a rather old name but used very widely. For example, when we have source files under `src/`, and if we build it under the directory for multiple architectures, then we will need to remove all the built files for each build. It is a waste of time. In order to address this problem, make another directory `obj/`, create symbolic links under it, and build under `obj/`. This is the shadow directory approach. All the built object files are created under `obj/` and the developers do not need to care about the difference of the build directory.

LiveCD implemented by this approach (readonly FS + tmpfs + shadow dir) succeeded to redirect the file I/O, eg. read from readonly FS and write to tmpfs, but it brought huge number of symbolic links. Obviously it was not the best approach.

Squashfs and AUFS

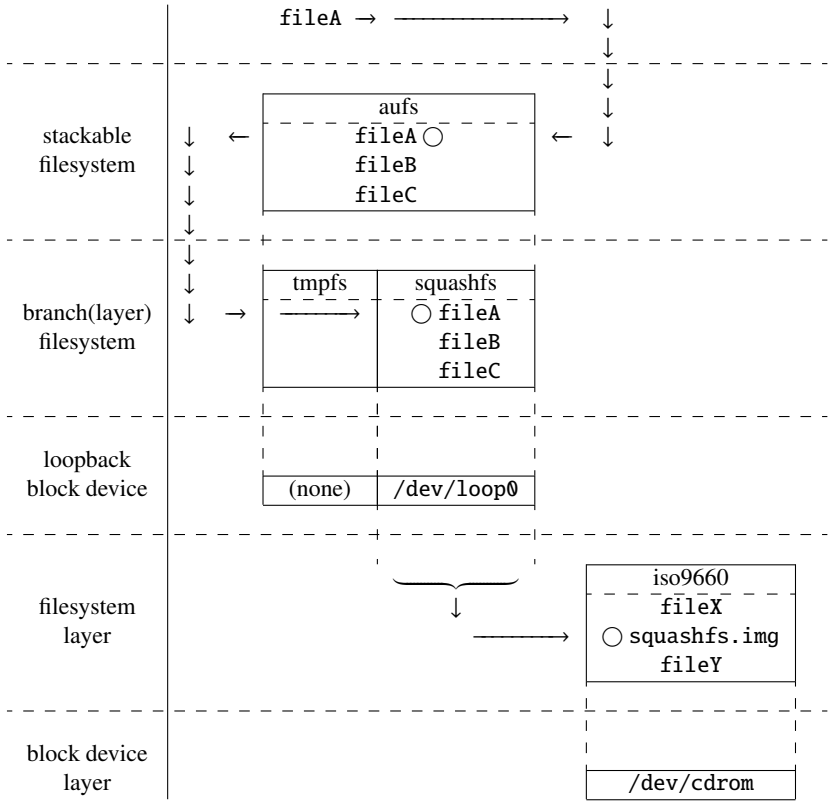
Later LiveCDs adopt a stackable filesystem, eg. Unionfs or AUFS. They both introduce a hierarchy to mounting filesystems, and mount a writable filesystem transparently over readonly filesystems, and provide essential features of redirecting file I/O to layers and internal copy-up between layers which enables modification the files on the readonly filesystems logically. Since readonly filesystems can be specified more than one, LiveCD can handle them as an application packages such like compiler package, office suite. For example, SLAX LiveCD provides this "modularize" and users can choose and add them as AUFS layers.

Current LiveCDs generally uses tmpfs or HDD as an upper writable layer (which is called "branch"), squashfs for the lower readonly layers, and make them stacked by AUFS.

Example of mounting squashfs and aufs

```
(boot script in the initramfs)
```

Figure 1: Example of mounting squashfs and aufs




```
# mount -o ro /dev/cdrom /cdrom
# mount -o ro,loop /cdrom/squashfs.img /squashfs
# mount -t tmpfs none /tmpfs
# cd /tmpfs; mkdir -p tmp var/log; chmod 1777 tmp; cd /
# mount -t aufs -o br:/tmpfs:/squashfs none /aufs
(and then boot the system with /aufs as a root directory)
```

Fig.1 represents the filesystem and block device layers after above script.

For your information, Fig.2 and Fig.3 show the layers of an ordinary mount and a loopback mount individually.

But, as you might know, there are other approaches to implement these features, and this report introduces you “cloop” and “dm-snapshot.”

CLOOP — Compression in Loopback Block Device

The module, cloop (compressed loopback block device) brings the compression feature into a block device layer instead of filesystem layer. This module is not merged into

Figure 2: Normal mount

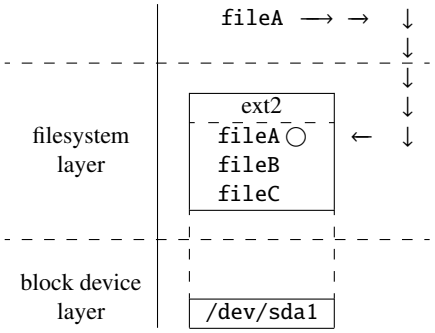
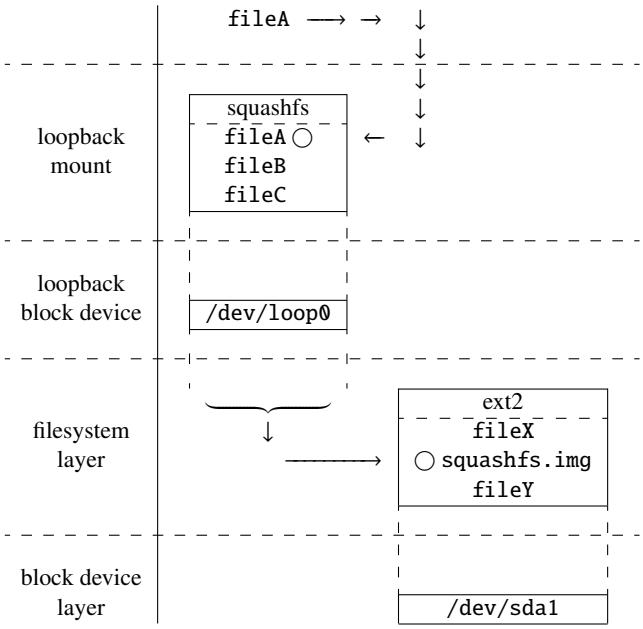


Figure 3: Loopback mount



linux kernel mainline.

`http://debian-knoppix.alioth.debian.org/packages/cloop/cloop_2.636-1.tar.gz`

Since this is a feature of block device layer, the filesystem type never matter. Any type of filesystem can be compressed. Create a filesystem image file, copy the necessary files into it, and then create a cloop image file by `create_compressed_fs` command (aliased `advfs`).

Create and mount a cloop image

```
$ dd if=/dev/zero of=/tmp/ext2.img bs=10M count=1
$ mkfs -t ext2 -F -m0 -q -O dir_index /tmp/ext2.img
$ sudo mount -o loop /tmp/ext2.img /mnt
$ tar -C /bin -cf - . | sudo tar -C /mnt -xpf -
$ sudo umount /mnt
$ create_compressed_fs -q -L9 /tmp/ext2.img /tmp/ext2.cloop.img
$ sudo losetup -r /dev/cloop0 /tmp/ext2.cloop.img
$ mount -o ro /dev/cloop0 /mnt
(and then use /mnt as ext2)
```

Layers do not change essentially from Fig.3, but the differences are

- filesystem independent, we can compress any type of filesystem.
- the device changes from `/dev/loopN` to `/dev/cloopN`. `/dev/cloop0` in the example is created by the cloop module.

And cloop does not support writing, it is readonly. So a stackable filesystem like AUFS will be necessary to support writing logically.

The cloop module is adopted by Knoppix LiveCD and its variants.

DM Snapshot — Snapshot by Device Mapper

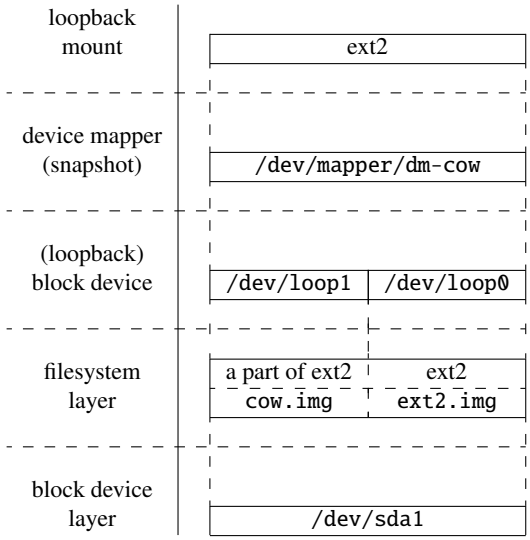
Instead of a stackable filesystem, LiveCD can use the "snapshot" feature of device mapper (DM) which is also a feature in the block device layer. Usually it is used by another abstract tool such as LVM, but we can handle it directly via `dmsetup(8)` command.

To use the snapshot feature, give DM two block devices, "original device" and "COW device (Copy-on-Write)". Then all writes will be redirected to the COW device, and reads will be redirected to either the COW or original device. This redirection is exactly the behaviour which stackable filesystems provide. All the changes are stored in the COW device, and the original device never change.

Block based COW by DM

```
(create a 16MB ext2 image)
$ dd if=/dev/zero of=/tmp/ext2.img bs=1M count=16
$ mkfs -t ext2 -F -q /tmp/ext2.img
$ sudo mount -o loop /tmp/ext2.img /mnt
$ > /mnt/fileA
```

Figure 4: Example of mounting dm-snapshot




```
$ sudo umount /mnt

(create a 1MB sparse file for COW image)
$ dd if=/dev/zero of=/tmp/cow.img bs=1 count=1 \
       seek=$((1024 * 1024 - 1))

(attach loopback block device for each image files)
$ sudo losetup -r /dev/loop0 /tmp/ext2.img
$ sudo losetup /dev/loop1 /tmp/cow.img

(combine two devices)
$ echo 0 $(du /tmp/ext2.img | cut -f1) snapshot \
       /dev/loop0 /dev/loop1 p 0 | sudo dmsetup create dm-cow

(mount the combined device)
$ sudo mount /dev/mapper/dm-cow /mnt

(and then use /mnt as ext2. all the writes to /mnt are stored into
 /tmp/cow.img, and /tmp/ext2.img never change)
```

Although the behaviour looks like a stackable filesystem, the target of the operations is different. DM operates on block device/sector, but a stackable filesystem operates on filesystem/file. For example, there exists a file sized 2KB in a block device whose block size is 512B. In other words, the file consumes 4 blocks. When we write a byte at the offset of 1KB in the file (the first byte in the third block), then DM will write only one block (the third block) to the COW device. (Strictly speaking, some

internal information such like timestamps will be written into the inode block). Now the COW device contains only a part of ext2 while the original device is a perfect ext2 which contains all blocks unchanged. So we cannot mount the COW device alone as a valid filesystem. On the other hand, stackable filesystem operates on file. It copies the whole file (2KB) from the lower layer to the upper layer, and then writes the originally requested data to the file on the upper layer.

In efficiency, DM snapshot is better since it copies only one block. But in some usability, it may become bad. In stackable filesystem, it is easy to see which file was updated if we run `ls(1)` to the upper layer. Additionally `diff(1)` between the lower and upper layers is available too, which may be useful for backup.

In DM snapshot, it is impossible to mount the COW device alone, and of course `ls(1)` will not work for it. The COW device always has to be combined with the same unique original device. If some modifications happen in the original device during it is not combined to the COW device, then some conflict will happen and the COW device will not be usable anymore.

DM operates on the existing devices. It is possible too even if the device was created by previous DM operation. By this feature, we can make the snapshots stacked. In other words, the multiple layers can be specified.

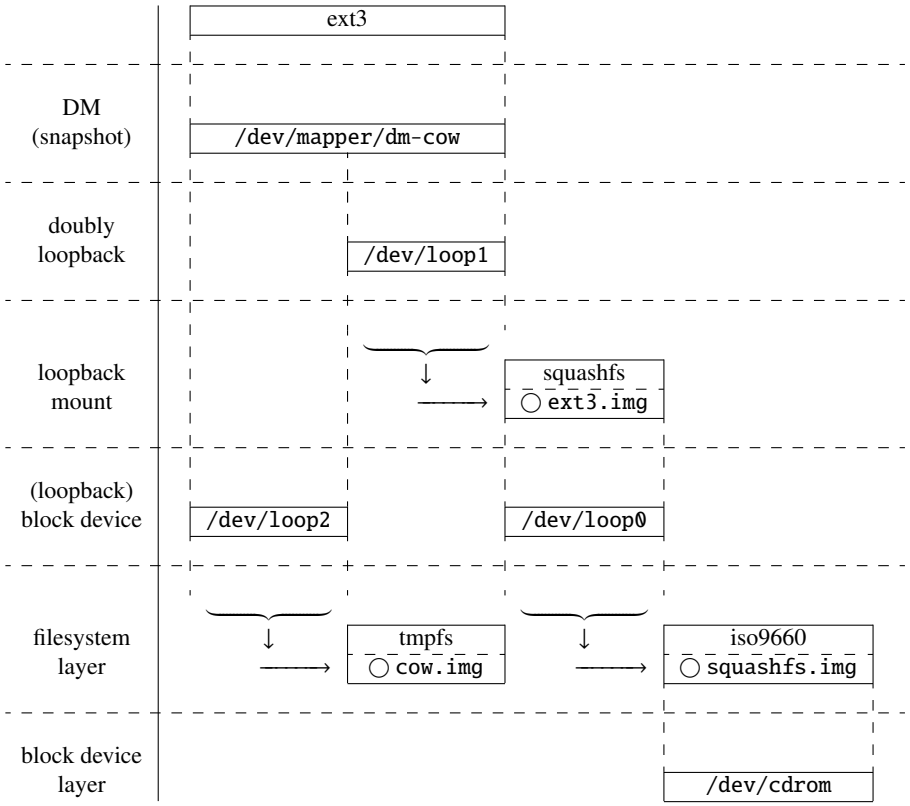
Note that the filesystem on the original device has to be writable one while the original device is readonly, because it is the feature of filesystem to write data to the file. If we create a filesystem which is natively readonly, squashfs or something, on the original device and add the COW device, then we cannot write to it since the filesystem is readonly. But we want compression, in deed. It is very hard to make LiveCD contained the full Linux system without compression. There is a LiveCD which adopts DM, Fedora (at least version 12). Fedora creates an ext3 image file (actually it is ext4), and then creates a squashfs image from this ext3. In this case, Fedora mounts it by nested loopback mounted. Fig.5 shows the layers with a little simplification.

Example of using both of DM and squashfs

```
$ sudo mount -o ro /dev/cdrom /cdrom
$ sudo mount -o ro,loop /cdrom/squashfs.img /squashfs
$ sudo losetup -r /dev/loop1 /squashfs/ext3.img
$ dd if=/dev/zero of=/tmp/cow.img bs=1 count=1 \
       seek=$((1024 * 1024 - 1))
$ sudo losetup /dev/loop2 /tmp/cow.img
$ echo 0 $(du /tmp/ext3.img | cut -f1) snapshot \
       /dev/loop1 /dev/loop2 p 0 | sudo dmsetup create dm-cow
$ sudo mount /dev/mapper/dm-cow /mnt
```

It may be a little annoying, but I think I should describe about this figure a little more. Trace Fig.5 from bottom to top. When we mount CD-ROM, we find only one file named `squashfs.img` in it. After loopback mounting it, we see only one file (again) named `ext3.img`. And we attach `/dev/loop1` to make it usable as a loopback block device. We also create an empty file separately and attach `/dev/loop2` to it. Finally we can create a new device `dm-cow` from `loop1` as the original device and `loop2` as the COW device, and mount it as a writable filesystem finally.

Figure 5: Example of using both of DM and squashfs



Nested Loopback Mount and Squashfs Cache

Now we think about pros and cons of this nested loopback mount scheme.

Firstly the resource consumption comes to our mind. Adding a new block device causes a new caching for it. It may be double-caching. And when we mount something, the system creates an object per mount and adds it to the mount hierarchy which means consuming some memory and necessary to traverse the tree when we access to a file. Obviously an extra loopback device is consumed. Generally the minor device number is assigned from 0 to every block device, and the number is limited. For loopback block device, by default, from 0 to 7, eight devices are created. If we specify a parameter to the loopback block device module, we can extend the limit upto 255. But we should take care the number of consumed loopback block devices since this scheme consumes it twice in roughly speaking. Additionally the kernel thread named loopN is invoked and stays resident for each loopback mount, which leads to more consumption of CPU time, process ID space, and scheduler.

Hold it!

I might write too bad about this scheme. Actually it has a very big advantage. It is caching of the decompressed result in squashfs.

In order to gain higher compression (smaller size in result), `mksquashfs` basically compresses per a specified size (a block size in squashfs. 128KB by default). When the target file is smaller than this size, it is gathered with other small files and compressed together. This is called fragment block in squashfs. Thus when we have `fileA` whose size is 1KB and also a 1KB file named `fileB`, and read `fileA` only, then squashfs internally reads and decompresses `fileB` too.

Squashfs adopts an original cache scheme for fragment blocks instead of generic caching (page cache), which is called fragment cache.

The size of fragment cache is configurable. It is `CONFIG_SQUASHFS_FRAGMENT_CACHE_SIZE`, and its default value is 3.

The decompressed `fileA` will be stored in page cache as a generic file read, but other files in the same fragment block will not. They are stored in squashfs fragment cache only (as compressed). If we read `fileB` during its fragment block is in fragment cache, then it is OK. Otherwise, squashfs reads and decompresses it. Since the size of fragment cache is not so large, when our access pattern is very random, the fragment block will be out of cache easily and the speed performance will be damaged in result. Of course, we can expand the size of fragment cache by configuration, but the fragment cache in squashfs is allocated constantly and never grow nor shrink. So the too large value may lead to a pressure of memory.

On the other hand, by nested loopback mount, the second loopback block device caches all of the decompressed result in page cache including the fragment blocks. So the chances to decompress the same block will reduce, and it will contribute the speed performance. As long as this is nested, it may cache things unnecessary doubly. But the benefit is large and the page cache discards data in the order of less used. In this scheme, the lower cache (squashfs) may be discard first, and then upper cache (ext3 in the above example) is subject to survive. Finally, the gained speed is larger than the

Table 1: combination of modules

lower layer	upper layer	stack
squashfs	tmpfs	aufs
anyfs + cloop		
anyfs but writable + squashfs + nested loopback		dm-snapshot

demerits described previously. And this squashfs does not contain the small files, so the fragment block will never exist.

`mksquashfs` has an option `-no-fragments` which prohibits creating fragment blocks. It is very effective for random access pattern but the size of the squashfs grows.

Performance of Reading and Decompressing

Generally CPU is much faster than block device, and the decompression consumes much CPU time. This issue may become meaningful when reading from the compressed filesystem.

In generic desktop systems, it is faster to read the uncompressed file than the compressed one due to the CPU cost to decompress. When the block device is much slower, very much, the speed difference between CPU and block device grows, and finally there will be a case which is slower to read the uncompressed file than the compressed one. For instance, in the case of the size of a 4KB file becomes 1KB after compression, reading and decompress 1KB may be faster than reading 4KB from a very slow block device.

Comparing the Approaches

Here we have reviewed the necessary features of squashfs, aufs, cloop, and dm-snapshot. Now we are going to consider the combination of them to implement LiveCD. Those factors are compressed lower readonly layer, upper writable layer, and how to combine (stack) them. As described earlier, there is a condition about filesystem for dm-snapshot, finally the cases will be such like Tbl.1.

Ok, we get all necessary features and know how to combine and construct them for LiveCD. How about their performance? To compare these combinations, the actual content will be categorized by two types. It means “how to union” and the perfomance of “compressed readonly layer.”

Comparing “How to Union”

In comparing aufs and dm-snapshot, it is easy to estimate that dm-snapshot shows better performance due to the amount of copy-up described earlier. Also dm-snapshot will be better in `readdir(3)` too. In all other points, the actual operation is to traverse

Table 2: Comparing "how to union"

	`find` (usr, sys, elapsed)	copy-up (usr, sys, elapsed)
ext2	0.89, 1.32, 3.02	0.00, 0.03, 0.36
AUFS	1.13, 3.14, 5.01	0.00, 0.02, 0.39
dm-snapshot	0.89, 1.29, 3.02	0.00, 0.02, 0.36

vanilla kernel 2.6.33, Intel Core2 Duo 3GHz, x86_64, 4GB RAM
Executed on copied `/bin`, `/etc` (1661 files).

layers. Anyway, dm-snapshot will show the better performance everywhere since the difference is the fact that one is block device and the other is filesystem. Accessing a filesystem causes accessing a underlying block device internally. It is definetly an overhead in filesystem in comparing.

Points

To compare them, we use these cases. Although the loopback mount is used in read LiveCD, we don't take it here just because make things simpler.

1. ext2
2. ext2(RO) + ext2(RW) + aufs
3. ext2 + dm-snapshot

And we measure the elapsed time to do these.

1. `find(1)`
2. append one byte to the existing file to cause the copy-up.

In order to show the difference clearly, we measure these several times since the small difference is hard to distinguish. And the result is Tbl.2. The measurement scripts are included in the attachment. Refer them if you want.

The result shows almost equivalent as we expected. But this measurement looks too small, the number of file and the size of them look small too. So we could not see meaningful difference in comparing copy-up. Comparing `find(1)` shows that `aufs` is slowest, and I am afraid it glows as the number of layers grows. It may be same for `dm-snapshot` too, but the rate will be smaller than `aufs`'s.

Comparing "Compressed Readonly Filesystem"

Old Reports

Squashfs has a long history, and there have been made several reports in the past. A rather new one is made by Wenqiang Song, posted to LKML on 2009/6, titled "Squashfs 4.0 performance benchmark."

http://marc.info/?l=linux-kernel&m=124447049111108&w=2

In the report, he compared squashfs with ext4 in directory lookup, sequential I/O, and random I/O. Here summarizes roughly,

- squashfs is fast as ext4 or more.
- squashfs consumes more CPU(%sys).
- squashfs runs faster in loopback mount.
- limited RAM size makes squashfs slower, even in loopback mount.

And the compared points in the report is equivalent to the older one which took squashfs 2.0, 2.1 and others ("SquashFsComparisons". the last update was done in 2008/5).

http://www.celinuxforum.org/CelfPubWiki/SquashFsComparisons

The report took ext3, iso9660, zisofs(compressed filesystem), cloop, squashfs 2.0, and 2.1. The result shows, in general, squashfs 2.1 is better which means smaller and faster. If we look the reports closer, we also find that squashfs consumes CPU(%sys) more than ext3.

Support for LZMA

Official Support Both of squashfs and cloop supports ZLIB compression in the beginning. And then support for LZMA compression was added. Squashfs tried supporting LZMA for linux-2.6.34, and it looks working.

http://git.kernel.org/?p=linux/kernel/git/pkl/squashfs-lzma.git;a=summary
git://git.kernel.org/pub/scm/linux/kernel/git/pkl/squashfs-lzma.git

But the merging into 2.6.34 mainline was not done, and it is undone still.

While squashfs-lzma was not merged, supporting LZO was merged into linux-2.6.36. And supporting XZ is also being implemented.

the Sqlzma Patch In the history, there were other implementations to make squashfs to support LZMA. Among those patches, the sqlzma patch was widely used.

http://www.squashfs-lzma.org

The sqlzma patch uses the LZMA implementation in the 7-ZIP package which is developed separately. It is not so rare to port the user space code into kernel space, but porting the user space code written in C++ into kernel space is rather rare. It must be a power play. The official squashfs-lzma support for linux-2.6.34 also uses 7-ZIP, but the functions are written in C language. It must be more smooth porting. The reason sqlzma patch took C++ is simple. There did not exist C functions in 7-ZIP when the sqlzma patch was developed.

And the sqlzma patch implements the "hybrid compression" approach. Generally, the compression is not so effective for the random data or binary files. And in some

cases, the compressed result can be larger than the uncompressed original data. In the system which supports both of ZLIB and LZMA, users may want to take the smaller one, but we cannot decide it until both compressions are completed. Thus, based upon the native behaviour that `mksquashfs` compresses per specified size (128KB by default), the sqlzma patch tries compressing in both algorithms and take the smaller result. It compresses data by ZLIB first since it runs faster. When the result becomes smaller than the original data, then it tries compressing the same data by LZMA because such data is expected to be smaller. After comparing the two results, the sqlzma patch takes the smaller one. If the data does not become smaller by the first ZLIB, the sqlzma patch estimates such data will not become smaller even if it is compressed by LZMA, and decides to take the uncompressed original data. In other words, the sqlzma patch chooses one as its final result among the uncompressed original data, the compressed result by ZLIB, and the compressed result by LZMA. Of course the time to compress takes longer. In decompression, it is easy to decide how the data was compressed by the information to indicate whether the block was compressed or not which squashfs natively has, and by the header of the compressed data to indicate the data was compressed by ZLIB or LZMA. There is no necessary to add an extra indicator to represent how the data is compressed.

Support in Cloop Cloop also began supporting LZMA since the end of 2004 by an external package called `advancecomp`. But it is used in user space only (at least in version 2.636), eg. `create_compressed_fs` command support LZMA but the kernel module doesn't.

Support in Kernel Obviously the decompression is done in kernel space while the compression is done in user space, which means kernel has to support the LZMA decompression. The ZLIB decompression is already supported for a long time, but the support for LZMA decompression is rather new. The support began in linux-2.6.30 (2009/6) and the LZO decompression is supported since linux-2.6.33 (2010/2).

In this report, it is needless to say compare both of squashfs and cloop in ZLIB compression, but it does not sound interesting unless other compressions are not compared. For squashfs, I try incorporating the official LZMA support which had not been merged into 2.6.34. And the sqlzma patch for the old linux-2.6.27+squashfs 3.4.

Parallelizing the Decompression in Squashfs

Natively the decompression in squashfs is serialized. For instance, two processes read `fileA` and `fileB` in squashfs individually at the same time, squashfs waits for the completion of the first decompression before starting the second decompression. It is bad for modern multi core processors. A patch to address this problem was posted to LKML. It supports only the original ZLIB decompression, but I will try comparing it too. The sqlzma patch already implements this parallel decompression.

http://marc.info/?l=linux-fsdevel&m=127884119303014&w=2
[RFC 0/2] squashfs parallel decompression
[RFC 1/2] squashfs parallel decompression, early wait_on_buffer
[RFC 2/2] squashfs parallel decompression, z_stream per cpu

Points

To compare the above issues, decide the compared points like this.

1. ext2
2. squashfs-4.0 GIP
3. squashfs-4.0 GIP `-no-fragments`
4. squashfs-4.0 GIP + parallel decompression patch
5. squashfs-4.0 GIP `-no-fragments` + parallel decompression patch
6. ext2 + cloop GZIP
7. ext2 + squashfs-4.0 GIP + nested loopback
8. squashfs-4.0 LZMA (linux-2.6.34-rc6)
9. squashfs-3.4 + sqlzma patch (linux-2.6.27.4)

It is possible to make it faster and get better compression if we specify some parameters such as block size or the LZMA dictionary size, but I don't want to do it here since it will take more time to decide the parameters. Additionally I expect the tendency of the results will not change. So in this report, all parameters are left as default.

All the kernel version is linux-2.6.33 except squashfs 4.0 LZMA and squashfs 3.4+sqlzma. For squashfs 4.0 LZMA it is 2.6.34-rc6, and for squashfs 3.4+sqlzma 2.6.27.4. And I added some minor fixes to cloop-2.636 since it was failed to compile (`cloop-2.636-2.6.33.patch`).

Also the measurement is as below, and compare the compressed size, the elapsed time to read and decompress, and CPU time. It is another kernel thread to consume most CPU time, but the simple `time(1)` measures only the CPU time of `find(1)` and `cat(1)`. So I measure the CPU time by `vmstat(8)`. I know it becomes a little rough. If we used the kernel profiler or something, it might be possible to measure more correct CPU time.

1. `read(2)` files in the order of `readdir(3)` returned (as sequential access pattern)
2. `read(2)` files in the random order (as random access pattern)

For each case, run `cat(1)` repeatedly as a single process, and run multiple `cat(1)` concurrently. Similar to comparing "How to Union", run the measurement several times. And the results are Fig.6 – Fig.9, and Tbl.3.

Figure 6: Comparing “compressed readonly filesystem” – sequential + single

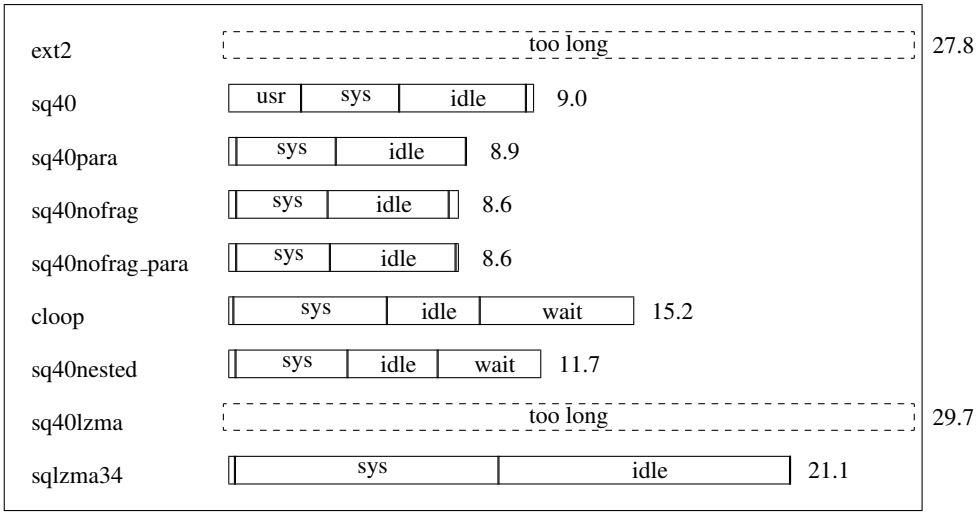


Figure 7: Comparing “compressed readonly filesystem” – sequential + multi

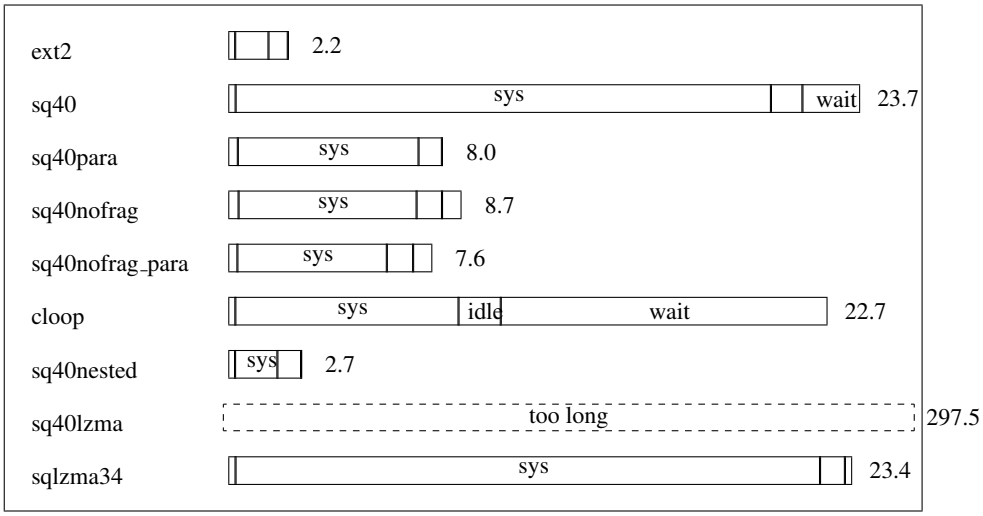


Figure 8: Comparing “compressed readonly filesystem” – random + single

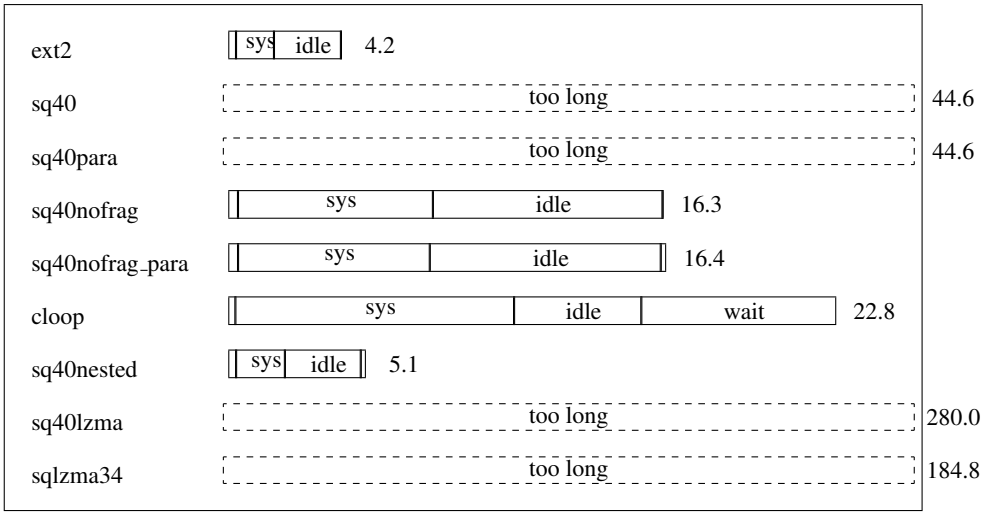


Figure 9: Comparing “compressed readonly filesystem” – random + multi

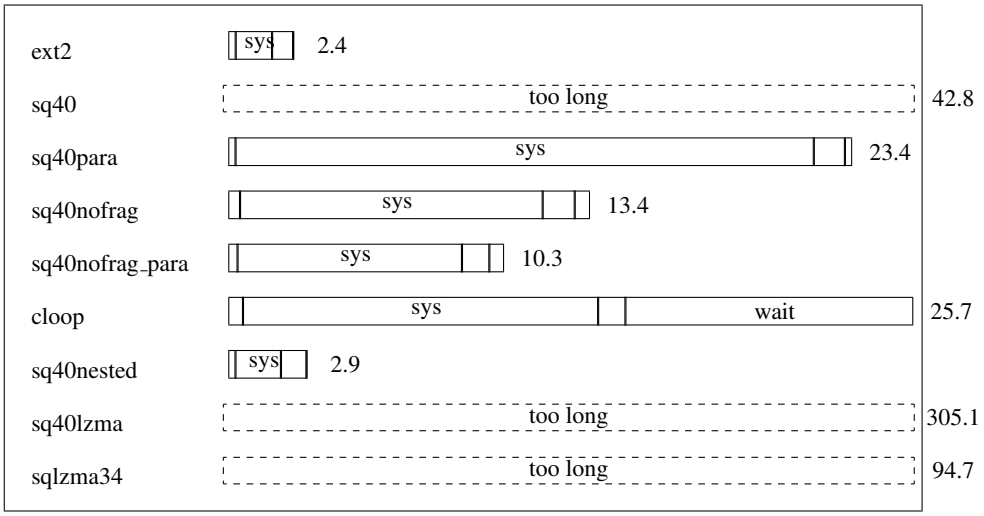


Table 3: Comparing “compressed readonly filesystem”

	size (blocks, KB)			elapsed time (sec)	CPU (dual core) %usr	%sys	%idle	%io-wait
ext2	—	seq	single	27.8	4	22	59	15
			multi	2.2	10	57	33	0
		rand	single	4.2	6	34	60	0
			multi	2.4	10	57	33	0
sq40	437512, 218536	seq	single	9.0	3	41	53	3
			multi	23.7	1	85	5	9
		rand	single	44.6	1	48	51	0
			multi	42.8	1	87	7	5
sq40para (parallel decompression)	ditto	seq	single	8.9	3	42	55	0
			multi	8.0	4	85	11	0
		rand	single	44.6	1	49	50	0
			multi	23.4	1	93	5	1
sq40nofrag	458264, 228904	seq	single	8.6	3	40	53	4
			multi	8.7	4	77	11	8
		rand	single	16.3	2	45	53	0
			multi	13.4	3	85	9	4
sq40nofrag_para (parallel decompression)	ditto	seq	single	8.6	3	41	55	1
			multi	7.6	4	74	13	9
		rand	single	16.4	2	44	53	1
			multi	10.3	3	82	10	5
cloop	521632, 260552	seq	single	15.2	1	38	23	38
			multi	22.7	1	37	7	54
		rand	single	22.8	1	46	21	32
			multi	25.7	2	52	4	42
sq40nested	515336, 257412	seq	single	11.7	2	36	29	33
			multi	2.7	8	59	33	0
		rand	single	5.1	5	36	56	3
			multi	2.9	8	59	33	0
sq40lzma (2.6.43-rc6)	391616, 195612	seq	single	29.7	1	47	51	1
			multi	297.5	0	70	8	22
		rand	single	280.0	0	50	50	0
			multi	305.1	0	70	25	5
sqlzma34 (2.6.27.4)	390592, 195100	seq	single	21.1	1	47	52	0
			multi	23.4	1	94	4	1
		rand	single	184.8	0	50	50	0
			multi	94.7	0	98	1	1

Intel Core2 Duo 3GHz, x86_64, 4GB RAM
Executed on copied `/usr/share` (59212 files).

File Size Obviously and expectedly, adding the `-no-fragments` option makes the squashfs image smaller. But it is highly depends upon the number of smaller files. With using LZMA (even without `-no-fragments`), we get smaller (reduced by 11.1%). By the sqlzma patch which adopts the hybrid algorithm is the smallest (reduced by 11.2%).

The cloop image file looks larger here, but it may be due to this measurement. The original filesystem image was ext2 and since it is annoying for me to decide the just fitting size, the size of the ext2 image file contains unused region. (in this measurement, the original filesystem size was 732MB, and 75MB is unused). Although I have not checked how I could reduce the size if I cut this unused region, I guess it will become smaller. In squashfs, the unused region is zero or smallest. If I use iso9660 instead, the size may be smaller too. Of course, specifying the parameters like block size is much easeir way.

The same original ext2 image is used for `sq40nested` which will gets the benefit of no fragment blocks at all and nested caching. While using the same data, `sq40nested` gets smaller result. I guess the reason is the default block size and the filesystem type.

Elapsed Time See `sq40` (Squashfs4.0 in vanilla linux-2.6.33) first, and we find other than the sequential access by single-process is slow. Additonally multi-process case is worse. The test system is dual core. Ideally multi-process cases complete in half of single-process. But the result shows differently and looks like CPU is not utilized enough. I guess why the random access is slow is the fragment blocks and why multi-process is slow is that the parallel decompression is unimplemented. Of course, if the test system was single core, then multi-process and parallel decompression would be meaningless.

In `sq40nofrag`, the degrade in multi-process case is almost equivalent to single-process. And the random access shows twice as the sequntial access approximately. I guess that the unimplemented parallel decompression is the reason.

To see how the parallel decompression is good, compare `sq40para` and `sq40nofrag_para` with `sq40` and `sq40nofrag` individualy. The image file is same for each pair. Both of `sq40para` and `sq40nofrag_para` are patched by the parallel decompression patch (described earlier), and `-no-fragments` is added to `sq40nofrag_para`. In single-process, there is no big differennce. But in multi-process, the parallel decompression contributes much. Of course, it consumes CPU much.

`sq40nested` which is nested loopback mount shows excellent. The cache is powerfull and fast as uncompressed `ext2`. But we should care about the installed RAM. If it is not so large, the performance will be damaged.

`cloop` took longer time. It may be due to the difference of the block size and the file system type. But in multi-process, the additional reason is that the unsuppported parallel decompression. Note that the degrade in the random access is small as the difference of layer (remind that cloop is implemented in the block device layer) is benefit.

In `ext2`, the first single-process sequential access shows very bad. I guess this is due to my measurement since it is the first one of all. All file access in this case causes accessing disk. Actually CPU(%io-wait) shows large value. All (almost?) files are cached by this first case, and we cannot see CPU(%io-wait) in the succeeding cases.

FYI: Suppport for LZMA Just for your information, I measured `sq40lzma` (squashfs 4.0 LZMA for linux-2.6.34-rc6) and `sqlzma34` (sqlzma patch for squashfs 3.4 in 2.6.27.4) too. As we expected, LZMA decompression is slow. `sq40lzma` takes twice longer time than `sq40` (squashfs4.0 ZLIB in 2.6.33). While `sqlzma32` implements the hybrid compression and the parallel decompression, and consumes almost all CPU time, it is still slow.

Note that `sq40lzma` is not officially merged into linux kernel mainline, and it may not be supported now. It is not fair to judge `sq40lzma` is not good by this simple measure and comparison only. Squashfs is aggressivly supporting various compressions other than LZMA, and actually some of them are merged into mainline.

In this report, I have compared squashfs and other technologies around LiveCD. All of cloop, dm-snapshot, squashfs and aufs are not bad. Obviously, the important thing is up to users who decides their best compression and implementation after prioritize the size, decompression speed, memory consumption and others.

Hope this helps.