Using deltas to speed up SquashFS ebuild repository updates

Michał Górny

January 27, 2014

1 Introduction

The ebuild repository format that is used by Gentoo generally fits well in the developer and power user work flow. It has a simple design that makes reading, modifying and adding ebuilds easy. However, the large number of separate small files with many similarities do not make it very space efficient and often impacts performance. The update (rsync) mechanism is relatively slow compared to distributions like Arch Linux, and is only moderately bandwidth efficient.

There were various attempts at solving at least some of those issues. Various filesystems were used in order to reduce the space consumption and improve performance. Delta updates were introduced through the *emerge-delta-webrsync* tool to save bandwidth. Sadly, those solutions usually introduce other inconveniences.

Using a separate filesystem for the repositories involves additional maintenance. Using a read-only filesystem makes updates time-consuming. Similarly, the delta update mechanism — while saving bandwidth — usually takes more time than plain rsync update.

In this article, the author proposes a new solution that aims both to save disk space and reduce update time significantly, bringing Gentoo closer to the features of binary distributions. The ultimate goal of this project would be to make it possible to use the package manager efficiently without having to perform additional administrative tasks such as designating an extra partition.

2 Comparison of filesystems used as repository backing store

The design of ebuild repository format make it hard to fit in the design of traditional filesystem. The repository contains many small files that usually leave a lot of sparse space in the data blocks that increases the space consumption and I/O overhead. Combined with random reads throughout the repository, it may degrade performance significantly.

In fact, the *gentoo-x86* repository (the main ebuild repository for Gentoo) as of snapshot dated 2014-01-17 contained 153000 files of total apparent size of 307 MiB. 131000 of those files (85 %) are

smaller than 4 KiB and 78500 (51 %) are smaller than 2 KiB. As a result, those files on a semi-efficient ext4 filesystem consume over 900 MiB.

The commonly accepted solution to this issue is to use another filesystem for the repository. The figure 1 lists common filesystems that are used to store the *gentoo-x86* tree along with the space consumsed by the contents of 2014-01-17 snapshot. The snapshot tarball sizes were added for comparison.

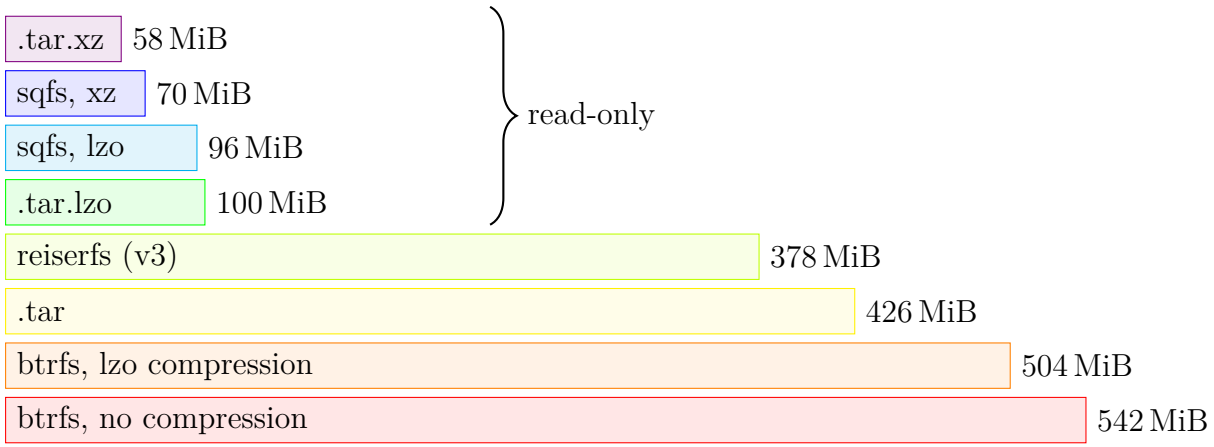

Figure 1: Comparison of gentoo-x86 tree size on various filesystems and in various archive formats

The sizes of SquashFS images and tarballs were obtained directly. In order to efficiently obtain free space of the remaining filesystems, the author has filled up the filesystems with a dummy file (with compression disabled) and subtracted the resulting file size from the partition size.

In fact, only two writable filesystems officially supported by the Linux kernel have made it into the table. Both btrfs and reiserfs were able to achieve that through packing of small files into shared blocks instead of placing each of them in its own separate block. As you can see, reiserfs does that more effectively.

It should be noted that btrfs keeps the packed files inside metadata blocks. With the default metadata scheme profile the metadata is duplicated within the filesystem. The contents of small files are duplicated within it, therefore the filesystem is almost twice as big. In order to achieve the presented size, the *single* metadata profile needs to be forced.

Since metadata (and therefore the small files packed in it) is not compressed within btrfs, enabling compression does not result in significant space savings.

It can be also noted that ReiserFS is actually more efficient than an uncompressed tar archive. This is due to large file header size and alignment requirements that result in numerous padding blocks. However, those deficiencies are usually easily removed by simple compression.

Much better space savings can be achieved through use of SquashFS filesystem which can fit the repository under 100 MiB. It stores files of any size efficiently, features data deduplication and full-filesystem compression. At the price of being read-only, effectively making it a more efficient format

of an archive.

The author has decided to take the LZO-compressed SquashFS filesystem for further consideration. The advantages of using stronger xz compression are relatively small, while LZO is significantly faster. Since SquashFS is read-only, it does not make sense to create a dedicated partition for it. Instead, the packed repository is usually placed on the parent filesystem which makes it easier to maintain.

3 Updating SquashFS repository

The main disadvantage of SquashFS is that it is read-only. As of version 4.2, the mksquashfs tool supports appending new files onto the image with a possibility of replacing the old root directory. Since that includes deduplication against existing files, this is fairly space-efficient way of updating the image. However, it requires having the complete new image available on another filesystem.

It is possible to provide a writable overlay on top of SquashFS by using one of the union filesystems — *unionfs*, *aufs* or *unionfs-fuse*, for example. Then, the apparent changes done to files on the SquashFS image are stored as files on another (writable) filesystem. While this is relatively convenient, the extra files quickly grow in size (mostly due to metadata cache updates following eclass changes) and the performance of resulting filesystem decreases.

There are two major methods of updating the SquashFS image using the rsync protocol:

1. on top of SquashFS image unpacked to a temporary directory,
2. on top of a union filesystem.

In the first method, the SquashFS image is unpacked into a temporary directory on a writable filesystem. Often *tmpfs* is used here to achieve best performance possible. The working copy is updated afterwards and packed back into a SquashFS image.

The second method makes use of writable overlay on top of SquashFS. This makes it possible to update the tree without updating all files, and pack them into a new SquashFS image using the combined union filesystem.

The main advantage of the unionfs method is that the intermediate tree can be used with a limited space consumption and good performance. For example, one may use rsync to update the unionfs copy frequently and rebuild the underlying SquashFS whenever the overlay tree grows large.

As it was presented on figure 2, the update is quite time-consuming. The approximate times were obtained on an Athlon X2 (2x 2 GHz) with *tmpfs* as underlying filesystem and a local rsync server connected via 10 Mibit half-duplex Ethernet (which bandwidth resembled a common DSL link).

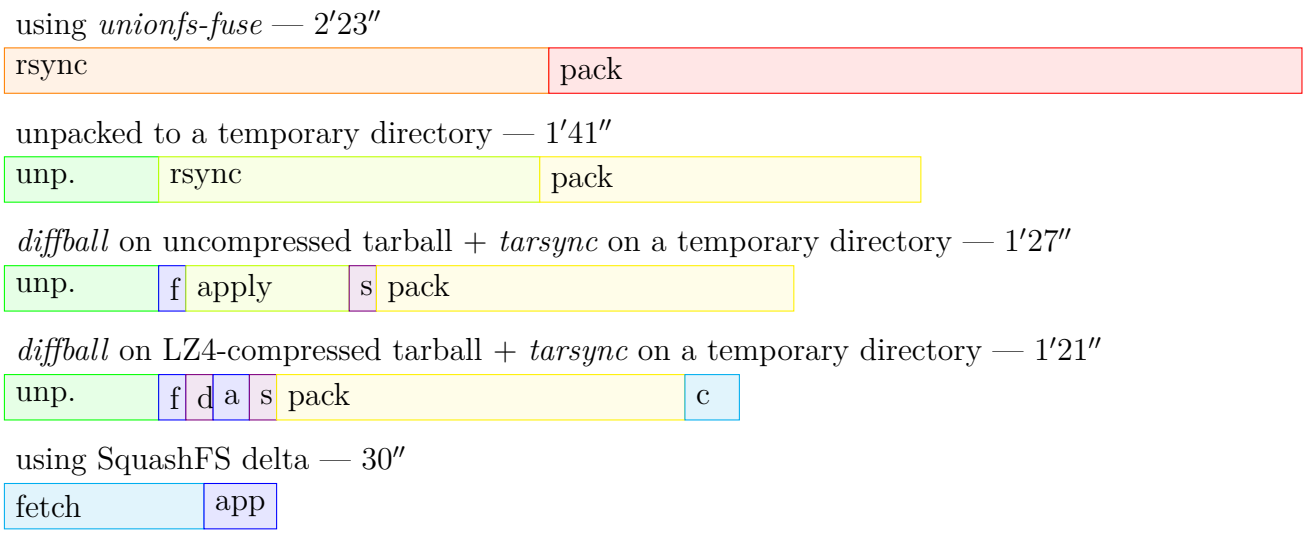

Figure 2: Comparison of gentoo-x86 weekly update time (2014-01-10 to 2014-01-17) using various update methods

The first interesting fact is that update using *unionfs-fuse* was actually slower than one using a temporary filesystem. This means that the week's worth of updates is enough to noticeably degrade the performance of the filesystem.

Then, it can be noticed that in the more efficient case rsync run and filesystem repacking took the same amount of time — about 42 s each. Improving either of the two would result in reducing update time.

One of the solutions aiming to replace rsync with a more efficient solution was Brian Harring's *emerge-delta-webrsync* script which works on top of daily portage snapshots (.tar archives). The script downloads a series of patches that are used by the *diffball* tool to update the snapshot to the newest version. Afterwards, the *tarsync* tool updates the repository efficiently using the tarball.

The main advantage of this solution is very small download size that averages to 300 KiB per day, and very quick filesystem update with tarsync (especially when done on tmpfs). Assuming that the patches are applied against tarball that is kept uncompressed on a hard disk, patch applying takes almost 20 s (on tmpfs it nears 4 s). Of course, one must account the additional space consumption of 400 MiB for the tarball.

This result can be further improved by using *lz4c* to compress the tarball. With the minimal compression (-1), it is able to get the tarball down to around 130 MiB. It takes around 3 s to uncompress the tarball from hard disk to tmpfs, and up to 6 s to write a new compressed tarball back to the disk. As a result, the compression not only makes the process faster but also gives significant space savings. The squashfs and .tar.lz4 pair sum up to around 230 MiB.

At this point, the only significant time consumer left is the SquashFS repacking process. Replacing that process with a direct update can reduce the whole update process to two steps: fetching

the update and applying it. Sadly, there are currently no dedicated tools for updating SquashFS. Therefore, the author decided to use a general-purpose *xdelta3* tool.

As it can be seen in the figure, this step practically changed the sync process from I/O and/or CPU-bound to network-bound, with most of the time involved in fetching the delta. This is mostly due to the fact that the patching process is done on top of compressed data, with any change in the repository resulting in rewrite of the surrounding block. As a result, day's worth of updates sizes up to 10 MiB and week's worth around 20 MiB.

Therefore, the delta update is highly inefficient network-wise. Nevertheless, even with a 5 Mibit link it is actually beneficial for the user.

4 Searching for the perfect delta

With the direct SquashFS delta update method, the update time is mostly determined by network throughput. Assuming that both the client's and server's bandwidth is limited and the union of the two is relatively low, it is beneficial to work on reducing the actual download size.

Since the delta files are relatively small, applying compression to delta files will not affect the update time noticeably. The best compression level offered by xdelta3 along with *djw* secondary compression allows to reduce the delta size to approximately 60 % of the uncompressed size.

The author has considered using either of the two types of patches:

1. daily patches,
2. combined patches.

The daily patch method is easier to implement and lighter server-side. It is used by the emerge-delta-webrsync tool. For each new daily snapshot, a single patch updating the previous day's snapshot is created. The client downloads all patches following the snapshot owned by him, and applies them in order to update it.

The alternate solution is to create combined patches, each of them containing all changes between the old snapshot and the current one. As a result, whenever a new daily snapshot is created the server would have to create new deltas for all the supported old snapshots. However, the client would have to download one patch only, without unneeded intermediate tree state.

Figure 3 plots the cumulative download size depending on the previous snapshot version and the update method. Along with the two SquashFS-based methods, the numbers for rsync and tar deltas (used by emerge-delta-webrsync) were provided.

The SquashFS deltas are relatively large, with noticeable constant term in the size. As a result, the cumulative download size for daily deltas grows rapidly and exceeds the size of the actual

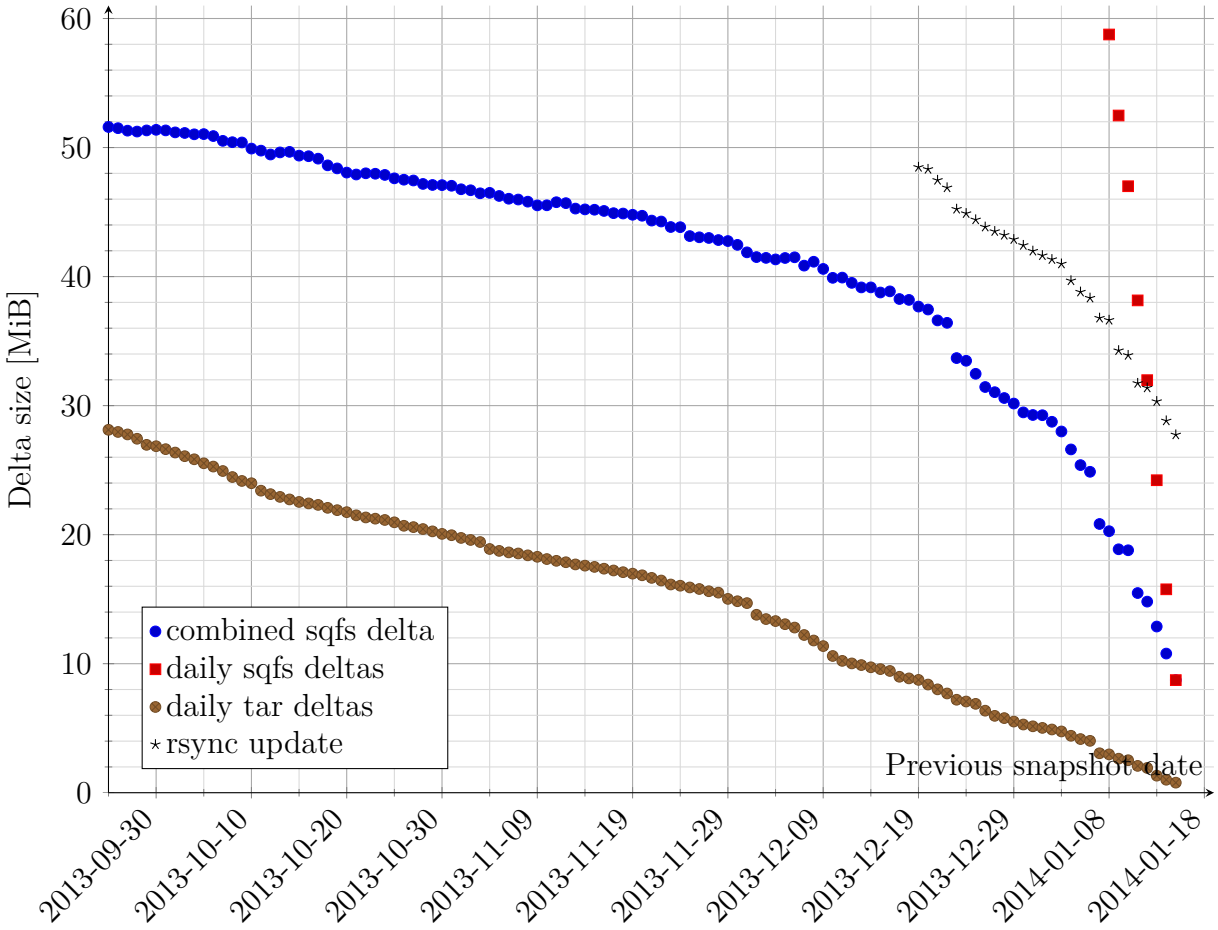

Figure 3: Plot of delta download size for update to 2014-01-17 snapshot

SquashFS image in less than two weeks. The delta update is no longer beneficial then. This practically disqualifies this method.

The combined delta method is much more network-efficient since the constant term is applied only once. The author has been able to produce deltas smaller than the target snapshot even with a year's old snapshot. However, with the size affecting both download and delta decompression time, downloading the new image may become more beneficial earlier.

Sadly, the combined deltas have much stronger impact on the server. Since for each new snapshot all deltas need to be regenerated, the delta generation increases the server load significantly. Additionally, it requires keeping a copy of each supported past snapshot which increases the disk space consumption.

Supposedly, an improvement in delta size could be gained via using the SquashFS append mode to update the images instead of recreating them. Since this does not repack the existing SquashFS fragments but appends changed files to the end of the archive, the resulting deltas may actually be smaller. The author has done two tests of that approach — starting with 2014-01-03 and 2014-01-10 image. The results of both are presented in figure 4.

Surprisingly, the deltas are smaller only for 3 – 4 days after the initial snapshot. Afterwards, they

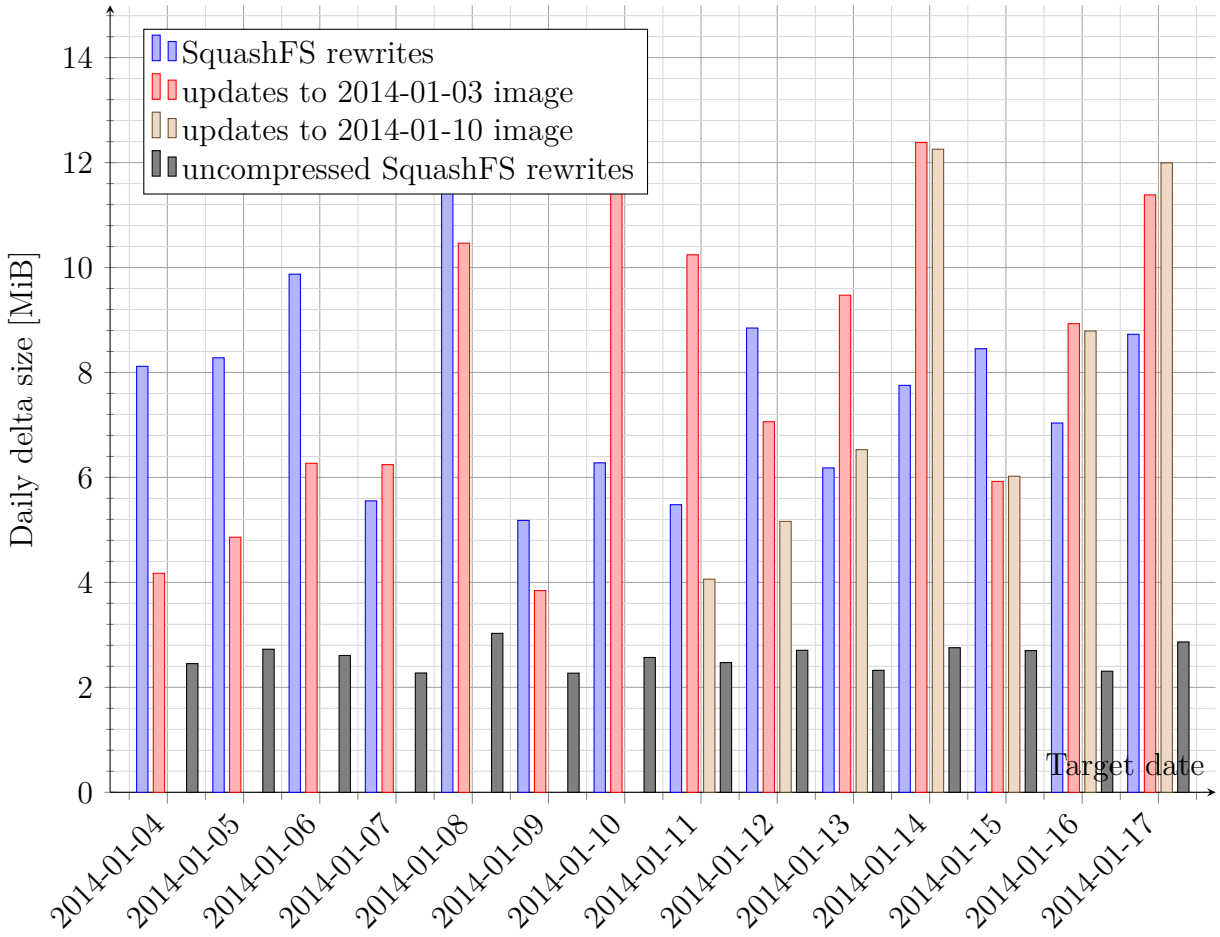

Figure 4: Daily delta sizes for SquashFS patches depending on update method

grow rapidly in size and exceed the size of plain dailies. Aside of that, the resulting SquashFS grows rapidly as well, doubling its size (and therefore the size consumption) in two weeks. Therefore, this attempt can be considered failed.

Another potential improvement would be to create deltas on top of uncompressed data and re-compress the patched SquashFS fragments as necessary. For a rough estimation, xdelta3 patches were created on top of uncompressed SquashFS images. They proved to be roughly four to five times smaller than the regular deltas.

Currently, this can only be achieved through unpacking and re-creating the SquashFS — which makes this method more time-consuming and network-inefficient than tarball patching. Therefore, a dedicated tool aware of the SquashFS format would need to be created first. The author is planning to work on that.

5 Summary

First of all, the Gentoo repositories are not suited for use on a shared filesystem. Even if the filesystem in consideration has good support for small files, the fragmentation will quickly result in loss of performance. So far, the most efficient solution, both in terms of performance and space efficiency,

is to use a read-only SquashFS filesystem image.

The rsync protocol that is currently used by default in Gentoo is inefficient both network- and I/O-wise. It requires a filesystem with random write support which disqualifies SquashFS and requires additional effort in order to perform the update. However, it has a single significant benefit — it allows updating the repository to the most recent state independently of its current version or local modifications.

The most bandwidth-efficient method of updating repositories is through use of tarball deltas and a tool similar to emerge-delta-webrsync. With enough RAM to hold a backing tmpfs, it may be additionally faster than rsync. The disadvantage of this method is that disk space consumption is additionally increased by the necessity of keeping the repository snapshot tarball.

The most efficient way of updating SquashFS repository images is to use direct SquashFS deltas. While unnecessarily large and therefore network-inefficient, they are very easy to apply. This makes them a good choice when a reasonably high network bandwidth is available and the amount of downloaded data is of no concern.

However, there is still room for improvement. A dedicated delta generation tool that would support uncompressing SquashFS blocks will likely result in significant decrease of the delta size. Similarly, a tool to convert SquashFS filesystems into consistent tarballs would allow improving the diffball method not to require keeping a separate tarball.

Until this issue is solved, the SquashFS deltas will not become the default sync method in Gentoo. But even in their current, inefficient form they may find many supporters. The sole use of SquashFS will find even more supporters, especially if portage starts to support specifying a SquashFS image path rather than a mounted filesystem.