Reducing SquashFS delta size through partial decompression

Michał Górny

March 5, 2014

1 Introduction

In a previous article titled 'using deltas to speed up SquashFS ebuild repository updates', the author has considered benefits of using binary deltas to update SquashFS images. The proposed method has proven very efficient in terms of disk I/O, memory and CPU time use. However, the relatively large size of deltas made network bandwidth a bottleneck.

The rough estimations done at the time proved that this is not a major issue for a common client with a moderate-bandwidth link such as ADSL. Nevertheless, the size is an inconvenience both to clients and to mirror providers. Assuming that there is an upper bound on disk space consumed by snapshots, the extra size reduces the number of snapshots stored on mirrors, and therefore shortens the supported update period.

The most likely cause for the excessive delta size is the complexity of correlation between input and compressed output. Changes in input files are likely to cause much larger changes in the SquashFS output that the tested delta algorithms fail to express efficiently.

For example, in the LZ family of compression algorithms, a change in input stream may affect the contents of the dictionary and therefore the output stream following it. In block-based compressors such as bzip2, a change in input may shift all the following data moving it across block boundaries. As a result, the contents of all the blocks following it change, and therefore the compressed output for each of them.

Since SquashFS splits the input into multiple blocks that are compressed separately, the scope of this issue is much smaller than in plain tarballs. Nevertheless, small changes occurring in multiple blocks are able to grow delta two to four times as large as it would be if the data was not compressed. In this paper, the author explores the possibility of introducing a transparent decompression in the delta generation process to reduce the delta size.

2 Theory of delta generation and application

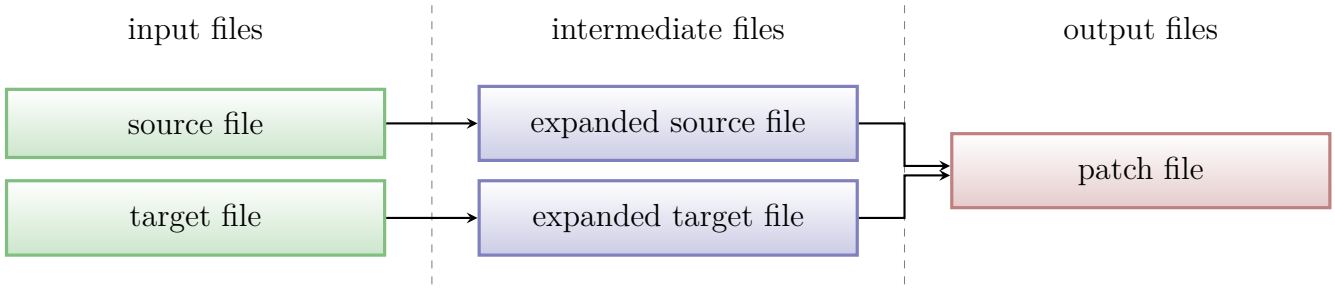

Figure 1: Outline of delta generation procedure

A rough outline of delta generation has been presented on figure 1. The input of the procedure consists of two files: the source file that the client is supposed to have already, and the target file that the client is going to reconstruct using the delta. The output is a patch (delta) file that the client needs to download in order to perform the reconstruction.

The additional decompression step that is going to increase the delta effectiveness introduces two intermediate 'expanded' files. Those files are created through decompressing data blocks in input files, and are used directly as input to the delta generation procedure.

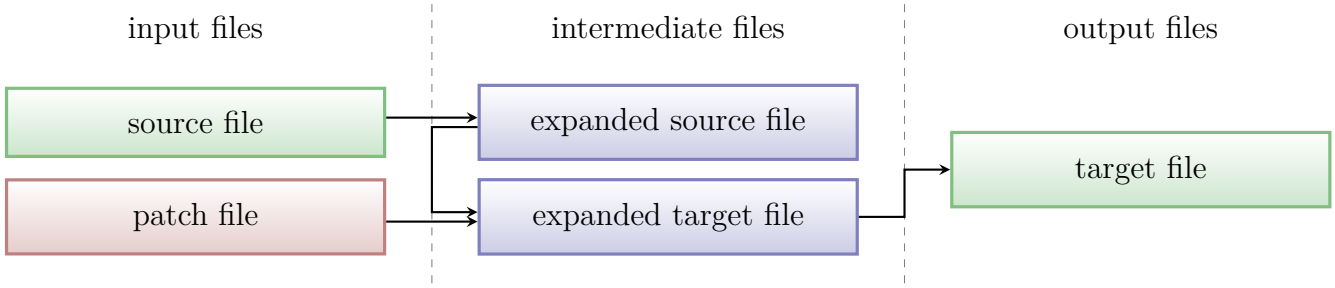

Figure 2: Outline of delta application procedure

The delta application is the complementary process. The input consists of a source file that has the same contents as the source file used during the delta generation, and the patch that is the result of delta generation. The source file is owned by the client as a result of previous download or update, while the delta file is downloaded for the update.

Since the delta was created on top of an expanded source file, the delta client needs to reconstruct this file first. Afterwards, the delta is used to reconstruct the expanded target file. This file is squashed into the final target file.

3 Practical aspects of SquashFS delta generation

The generic steps in the delta generation and application procedures are practically defined within the boundaries of delta tool (algorithm) used. However, the SquashFS 'expansion' and 'squashing'

steps are custom to the task, and the necessary facilities are not provided by any tool yet.

Since SquashFS supports having random non-compressed blocks, the canonical solution would be to use that feature to create a fully-compliant, non-compressed filesystem for deltas. However, this solution has a few drawbacks.

First of all, there is no point in decompressing the filesystem blocks that are equal in both input files. Since both the compressed and decompressed data is simply the same, the delta algorithm handles both cases equally well. In fact, the increase of data size may reduce the effectiveness of delta.

It is actually more reasonable to compare the two filesystems and decompress the blocks that have changed. As a result, the contents of expanded source file change depending on the choice of blocks for decompression. Since the client needs to reconstruct the expanded file exactly the same, the list of decompressed blocks needs to be provided to it — preferably inside the patch file.

The recompression step is easier. The standard SquashFS tools use uncompressed blocks only when the compression results in larger block. Therefore, if the client tries to recompress all the uncompressed blocks in the filesystem and discards those that result in larger data, it is going to obtain the correct target file. However, this implies repeatedly trying to compress blocks that were uncompressed originally. Providing the list of blocks decompressed in target file solves this inefficiency.

Even then, there is one potential issue left. SquashFS is quite complex and not well documented. The filesystem blocks and structures are referenced using byte offsets, and since the process results in data being shifted, all those offsets need to be updated. This makes the tool performing the task quite complex itself, and requires changes whenever the SquashFS format changes.

Considering that the intermediate files are used as delta input/output only, retaining the compliance with SquashFS file format is not beneficial. Actually, it makes both server and client tools more complex, binds both to a specific SquashFS version, and makes the target file reconstruction less efficient.

An alternative to the canonical solution is to use a more generic intermediate file format. Without any strict requirements, it can be designed to achieve the best efficiency. With a little effort, it can also become self-contained — that is, useful without any specific SquashFS awareness in the delta merge tool. As a result, the delta files are more universal and the client is simpler.

In order to achieve this goal, the patch needs to contain enough information for the client to obtain the locations and lengths of the originally compressed blocks and their decompressed counterparts. The author decided to develop the SquashDelta format specifically for this goal.

4 The SquashDelta format

The SquashDelta format defines two file types: the expanded (intermediate) file and the delta file.

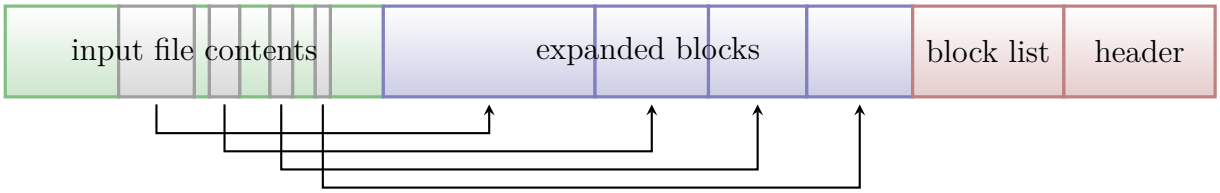

Figure 3: Structure of an expanded file

The expanded file is mostly an extension of a regular SquashFS file (or any other data file containing compressed blocks). The additional data is appended to the end of the file so that SquashFS structure offsets are retained, and that it is possible to expand and squash files in-place without shifting data.

Therefore, the initial segment of an expanded SquashFS file contains a verbatim copy of the input file data, except for the blocks that are being decompressed. During expansion, those blocks are replaced by 'holes' — areas filled with null bytes that can be stored and compressed efficiently.

The corresponding decompressed data is appended to the end of the file, creating expanded block segment. The blocks are placed there sequentially, without any additional metadata or padding. Although there are no strict requirements on the order of the blocks, it is recommended to sort them by offset, to make use of sequential I/O and increase likeliness of delta matches.

The expanded block segment is followed by SquashDelta-specific metadata — the block list that contains offsets and lengths of decompressed blocks, and the header that is used to identify the file and locate the beginning of the block list.

The process of squashing an expanded file starts with reading the header since its offset relative to the end of file is fixed. Afterwards, the block list is read and it is used to reconstruct the original file. The expanded data is compressed and written back to the holes in data segment. Afterwards, the file is truncated in order to discard the additional data, and it becomes byte-equal with the target file.

Figure 4: Structure of the patch file

The patch file is much simpler by design. It starts with the header and the block list for the source file. These are used by the client to reconstruct the expanded source file.

The two metadata blocks are followed by the delta of expanded files, stored in the format of the delta tool used. In the testing environment, it was the vcdiff format used by xdelta3 but support for other formats can be added in the future. The data is placed last, so that the open file descriptor can be safely passed to the delta merge tool after reading the headers.

5 Efficiency of the new solution

In order to estimate the benefits of the new format, the author repeated the tests from his previous article on the new format. The previous plots were supplemented with the new results.

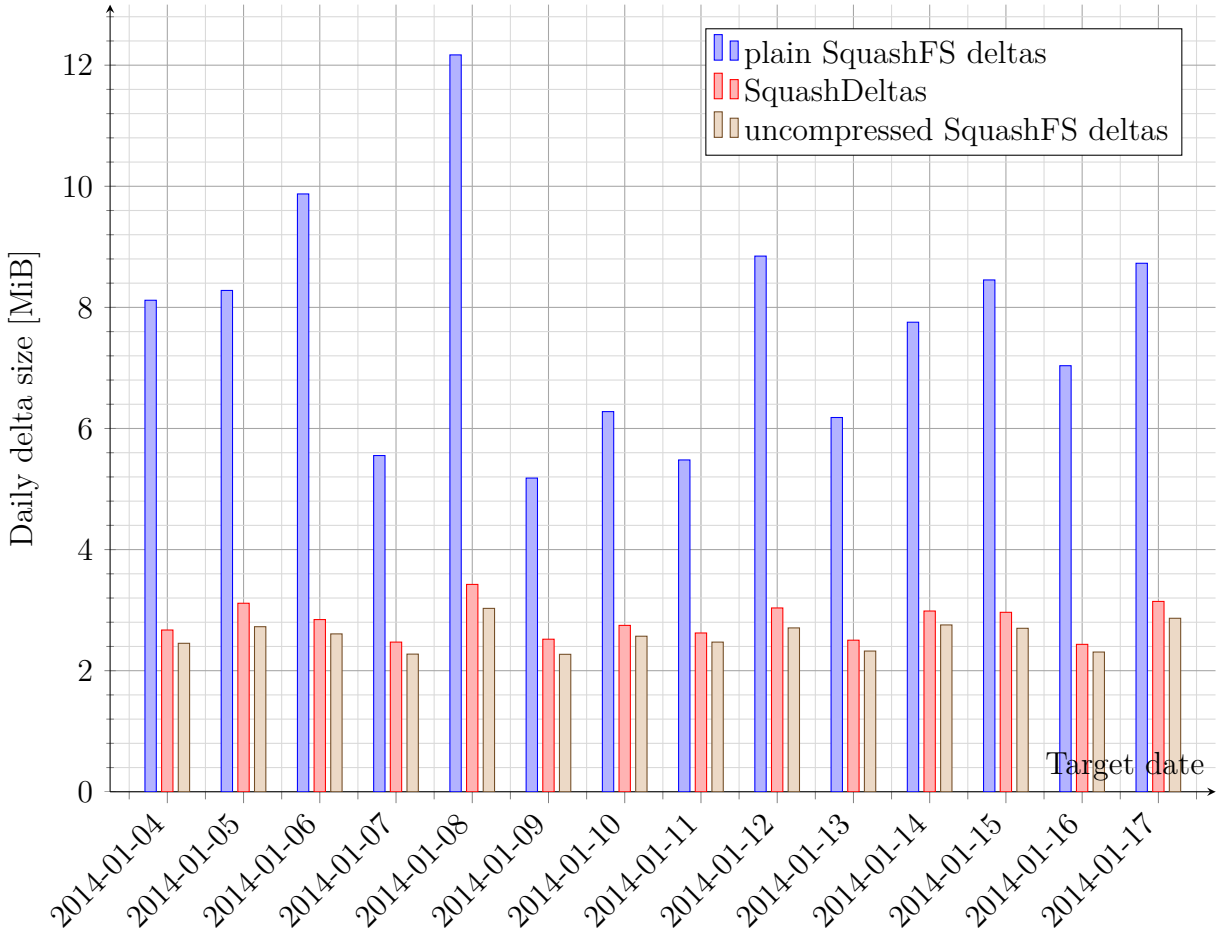

Figure 5: Daily delta sizes for different kinds of SquashFS deltas

In figure 5, the daily snapshot sizes in three SquashFS variants are compared — compressed, partially decompressed and completely decompressed. SquashDelta files are only a bit larger than deltas made using uncompressed files, and therefore two to four times smaller than the regular deltas.

The sizes of complete updates are compared in figure 6. SquashDeltas grow less rapidly over time than plain deltas, resulting in much better efficiency of updating both few days old and few months old systems.

The rapid increase of regular delta size is mostly related to the spread of changes in the filesystem

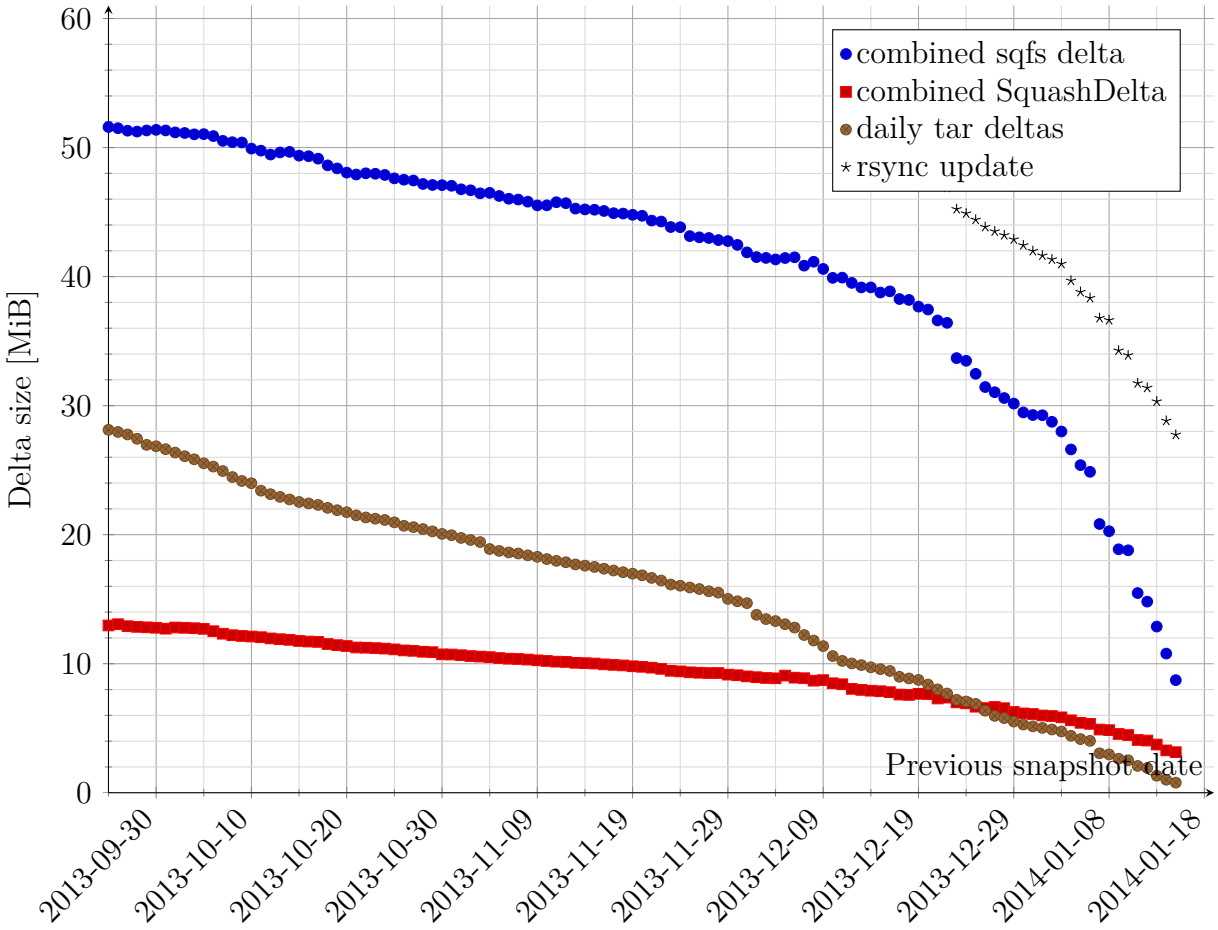

Figure 6: Plot of delta download size for update to 2014-01-17 snapshot

that causes growing number of blocks to be rewritten. The SquashDelta format efficiently compensates for that.

Interestingly, after a period of one month the SquashDelta update becomes smaller than the one performed using emerge-delta-webrsync. This is only because 'merged' deltas are used for SquashFS while diffballs are done daily and therefore their size accumulates over time.

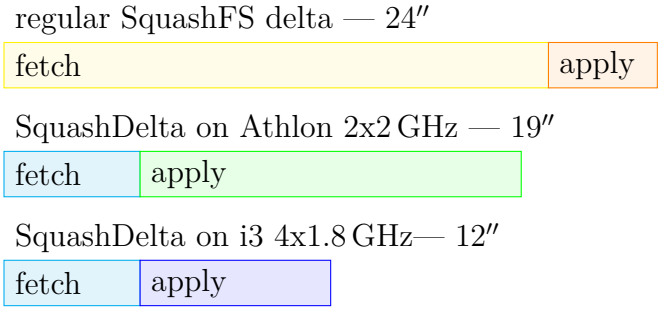

Figure 7: Comparison of gentoo-x86 weekly update time (2014-01-10 to 2014-01-17) using different types of SquashFS images and different CPUs

While SquashDelta is more efficient in terms of network I/O, the additional task of recompressing some of the filesystem blocks increases the update time significantly. In figure 7, the author compares

complete update time for various types of snapshots and CPUs.

On both of the tested systems, the SquashDelta update through a 10 Mibit network link was faster than plain update. The number of CPUs (cores) is also a significant factor since squashmerge is multi-threaded.

The exact results depend on compression algorithm and compression level in use. In this example, LZO with compression level 4 was used.

6 Summary

The use of binary deltas to directly update SquashFS images proved being a viable alternative to indirect repository update methods. It offered reduced complexity, memory and I/O use, and often faster updates. However, it involved unnecessarily large delta size that limited the speed of updates and increased mirror space consumption.

Those deficiencies were mostly related to the inefficiency of generating deltas against compressed data. The author created the SquashDelta format to overcome those issues through making it possible to transparently decompress input blocks during delta construction.

The squashdelta tool decompresses unique blocks in both files, providing a more suitable input for the delta generator. The resulting patches are two to four times smaller, and the decrease in update time can be noticed even on a moderately efficient CPU with a 10 Mibit network connection. The benefits are even larger when a network link with narrower throughput or more efficient CPU is used.

The SquashDelta format itself is quite universal and self-contained. Most importantly, this means that the delta generation tool can be updated to support newer versions of SquashFS while retaining compatibility with the same version of the delta merge tool. In fact, it is even possible to add support for another input file format while using the same file reconstruction tool.

The update efficiency can possibly be further improved by improving the block selection algorithm. Currently, all unique blocks in input files are decompressed. This includes new and removed blocks that do not have a common part, and decompressing those does not improve the delta efficiency. Excluding those blocks from the recompression process could decrease update time.

Further delta size reduction could potentially be obtained by replacing the binary delta of SquashFS metadata structures with a dedicated update algorithm. However, this would require more awareness of the SquashFS format and increase the complexity of both tools. The final gain may not outweigh the effort needed to implement it.