Speeding up Incremental Transfer Patch Generation [external]

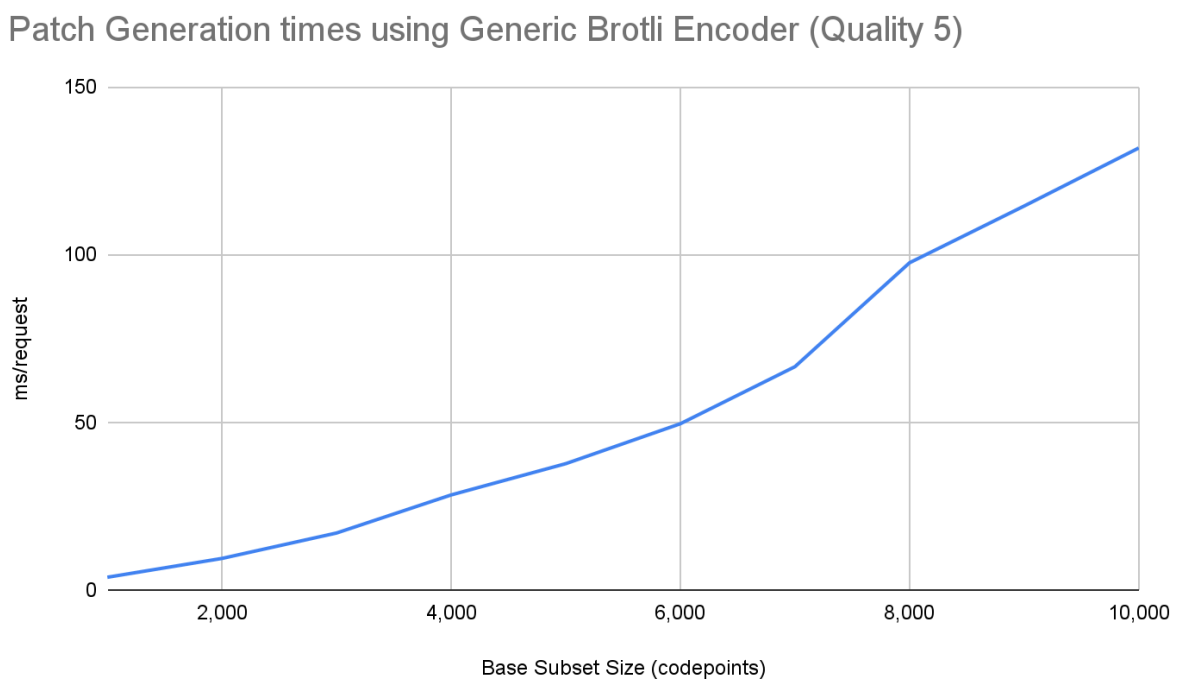
Author: Garret Rieger
Date: Oct. 18th, 2022

Introduction

In incremental font transfer a request for a patch is satisfied roughly by:

1. Compute a subset of the font that matches what the client currently has. (base subset)
2. Compute a subset of the font that matches the state the client wishes to be in. (derived subset)
3. Compute a patch/binary diff between the two subsets using shared brotli.

The simplest implementation of patch generation in step 3 is to use the generic brotli encoder with the base subset set as a custom dictionary. Experiments on encoding the with brotli have shown that the encoding time is roughly proportional to the size of the custom dictionary:



For large bases, brotli encoding times are over 100ms which is too slow for dynamic usage. This document discusses and experiments with a couple of ideas for improving the speed of step 3:

1. What are the performance vs size tradeoffs for the different brotli quality levels when using for patch generation?

2. How a custom brotli encoder, that is more aware of the incremental font transfer use case, can be used to:
    a. Cache pre-compressed data.
    b. Provide guidance to the brotli encoder to further speed up patch generation.

There has been some prior research that is relevant: Fast and efficient recompression using previous compression artifacts

Conclusions

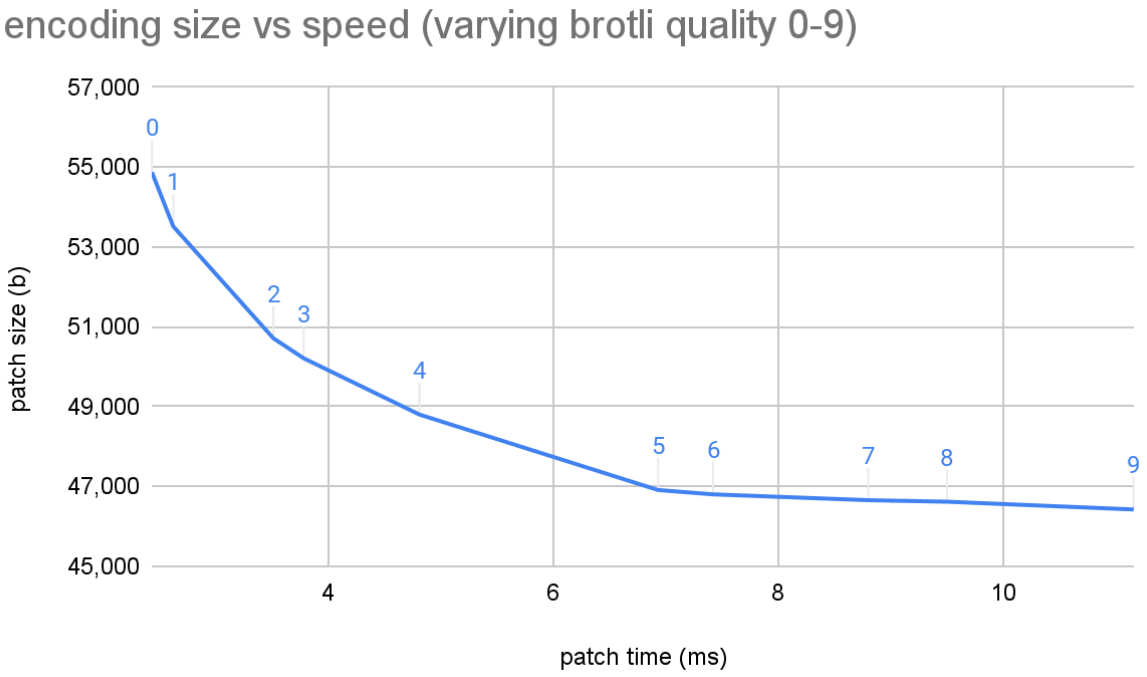
The experiments found:

1. For dynamic usage brotli quality 5 offers a good tradeoff between encoding time and encoded size.
2. Precompressing immutable tables is a viable approach and can be used to speed up the generation of the initial response.
3. A custom brotli encoder that is aware of the font format provides massive speedups in patch generation times, particularly with large bases (in one case times were reduced from 130ms to 3.6ms).

Effect of Brotli Quality

The easiest way to improve performance of the patch generation is to tune the brotli encoder quality level. This trades off increased compressed size in exchange for faster compression times.

To better understand how brotli quality effects the encoding time vs size tradeoff I tested encoding of a 750 codepoint japanese font across varying levels of quality:



From this it appears that brotli quality 5 is a sweet spot where further quality increases yield only marginal gains while still increasing overall encoding time.

Custom Encoder

The incremental font specification only calls for the produced patch to be a valid brotli stream. This leaves room for a custom encoder to be built that improves performance in this particular use case vs the existing general purpose brotli encoder.

Broadly there are two techniques which could be used to speed up patch generation:

1. Precompress portions of the font and cache them. Re-use these precompressed chunks to generate the final patch.
2. Use knowledge of the font format and patch use case to guide the encoder, for example:
    a. Instructing the encoder which segments of the font are unchanged to allow for fast backwards reference encoding.
    b. Instructing the encoder which parts of the shared dictionary are relevant. For example when encoding a table, only consider the equivalent table in the shared dictionary for use in generating backwards references.
    c. Instructing the encoder when tables (or portions of tables) are significantly different and to encode without considering the shared dictionary.

Brotli Concepts

Brotli File Organization

For a more complete overview see: Compressed Representation Overview

At a high level a brotli file consists of a header (specifies window length) followed by one or more “meta-blocks”. Each meta-block uncompresses to 0 - 16mb of data.

Data in brotli is compressed using two main methods:

1. LZ77 style compression that uses back references (ie. copy data seen previously in the uncompressed stream into the output).
2. Prefix encoding.

A meta-block is semi-indepedent of other meta-blocks. Prefix encoding is fully self contained, but meta-blocks can have backwards references to data from a previous meta-block.

Static/Shared Dictionary

Standard brotli has a static dictionary that is predefined by the specification. Backwards references that exceed the current window are interpreted to be references into that static dictionary.

Shared brotli adds a shared dictionary in addition to the static dictionary with a dictionary file that can be provided to the encoder/decoder. Backwards references outside of the window refer to this file.

Making Meta-Block Independent

By default meta-blocks are not fully independent from each other, however with a few small encoding behaviour changes they can be made to be fully independent. The specification specifically documents how this is done: Creating Self-Contained Parts within the Compressed Data

Notably a self-contained meta-block:

- Must use the same window length as the other meta-blocks in the stream.
- Cannot reference data from a previous metablock.
- Cannot reference the static or shared dictionary.
- Should probably be byte aligned following: Aligning Compressed Meta-Blocks to Byte Boundaries to allow for easy concatenation.

Incorporating a Custom Encoder

A custom encoder can be easily incorporated via the concatenation of independent meta-blocks. Where existing encoder behaviour is desired the existing encoder implementation can be used to produce the meta-blocks and where custom behaviour is needed meta-blocks can be produced by an alternate encoder and concatenated into the stream.

Case 1: Initial Request

To speed up the initial request we should find ways to precompress and cache portions of the font. Parts of the font which don't change with respect to the subset definition could be compressed into self-contained meta-blocks which could be cached and re-used when assembling the fully compressed subset. A more detailed algorithm follows:

Offline setup:

- Divide the tables of the font into two categories:
    a. Immutable: these tables don't change when the subset definition is changed. Note that some table which would normally change with the subset can be made immutable if retain gids is set and the table is added to the subsetter's passthrough list. For example GSUB/GPOS and COLRv1 may benefit from being immutable in certain cases where the offset based nature of the table render them difficult to patch efficiently.
    b. Mutable: these tables do change when the subset definition is changed.
- Re-order the tables in the font. Place all the mutable tables first and the immutable tables last.
- Compress, at the highest quality, the immutable segment into a collection of self-contained meta-blocks, and cache the resulting stream of meta-blocks.

At request time:

1. Compute the font subset, dropping all immutable tables.
2. Using the computed subset and the list of immutable tables from the original font compute a new table directory. Start a brotli stream and add the table directory to it.
3. Dynamically compress the tables from the subset result into the brotli stream.
4. Lastly, append the precompressed immutable segment to the stream and mark the last metablock as the last.

Caching Glyph Blocks

For most fonts glyph data makes up the bulk of the font file. So if we can find a way to precompress the glyph data that should yield significant performance improvements. Precompressing glyph data should be possible by dividing the glyph set into blocks:

Offline setup:

- Divide the glyphs into sets that are commonly needed together.
- Re-order the glyphs in the font such that grouped glyphs are sequential.
- Compress and cache each glyph block into a self contained meta-block.

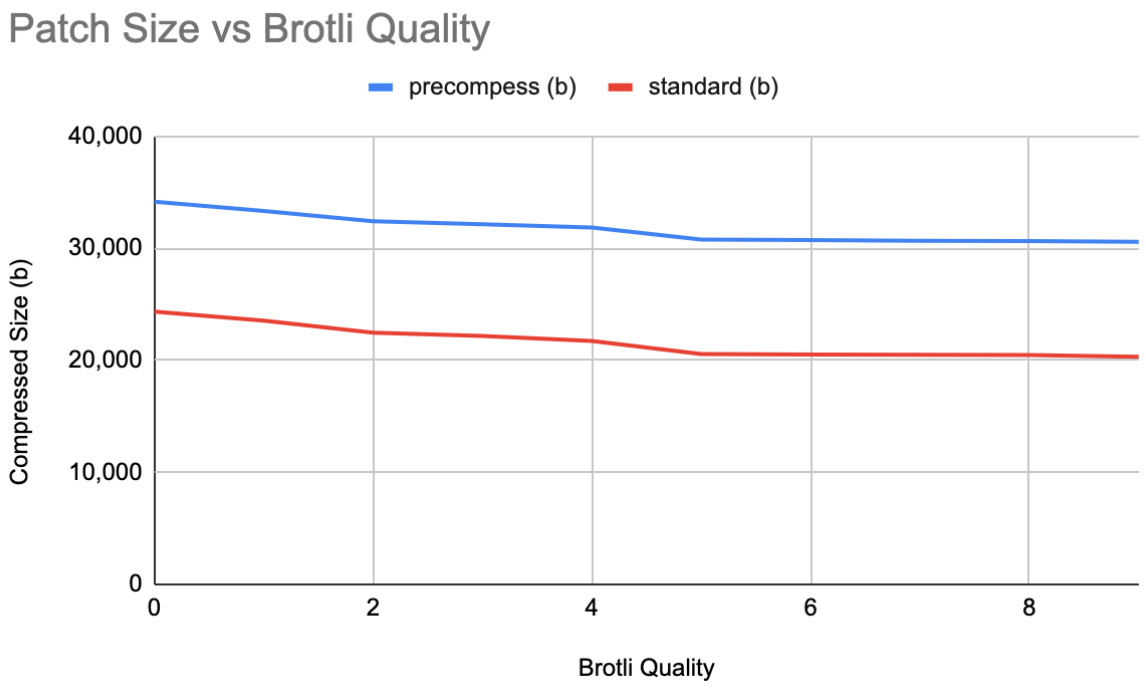
At request time:

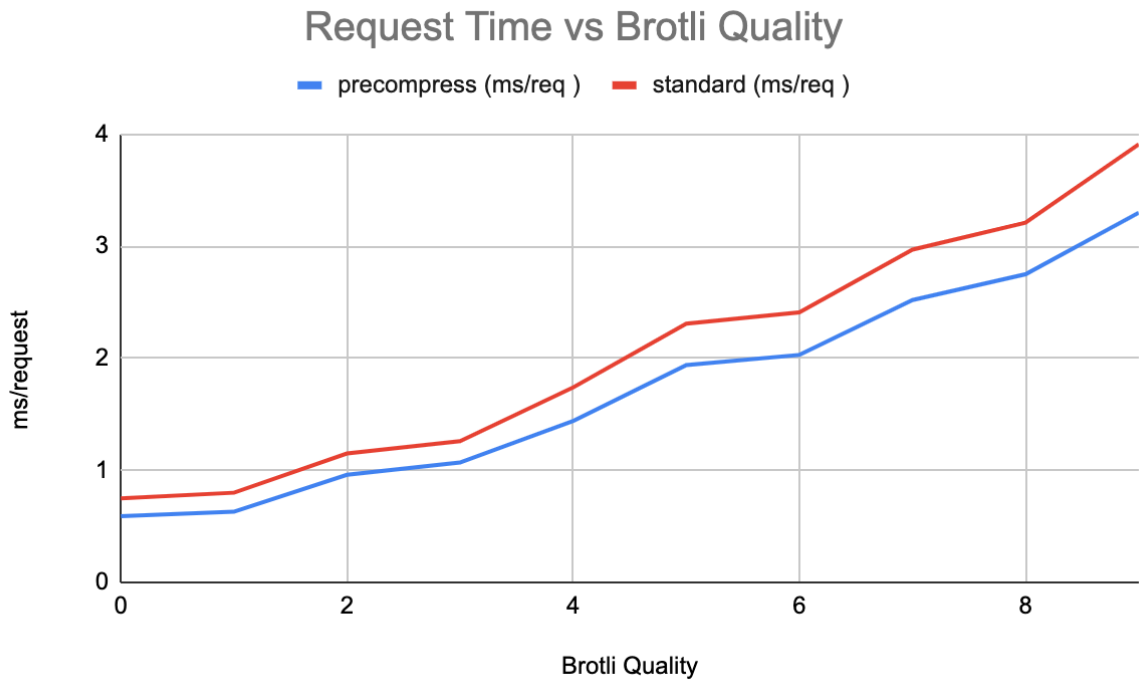
- Identify the set of glyph blocks that overlap the users requested subset.
- Assemble the glyf/CFF table using the cached precompressed blocks.
- Glyph data that fall outside of these glyph blocks can still be added dynamically (eg. in the case where only a small number of glyphs are needed from a block, then you could choose to dynamically generate instead of using the block).

Results

In this experiment a 250 codepoint subset of Roboto was compressed with brotli in two ways:

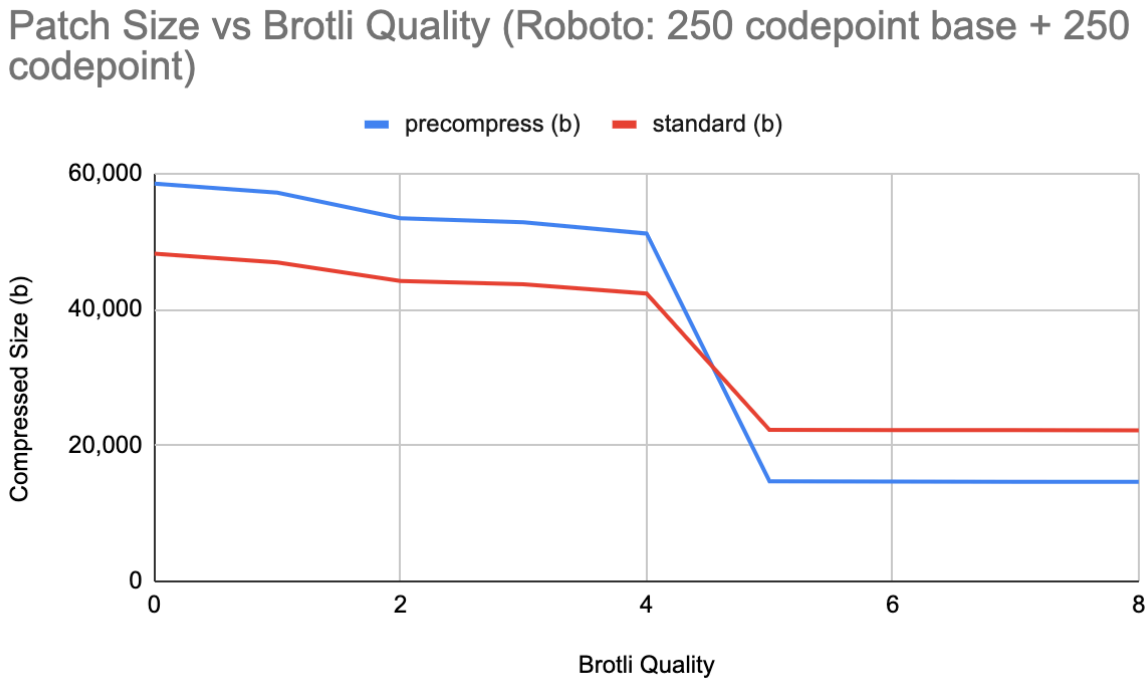
1. The layout tables (GSUB, GPOS, GDEF) were made immutable and precompressed at quality 11. The remaining portions where compressed at a varying quality level.
2. The layout tables were left mutable and the entire subset was compressed at varying quality levels.





From these results we can see that precompressing reduces encoding times as expected, as a tradeoff patch size is increased since the entire layout table is being included in the precompressed case.

In the precompressed case future patches will be smaller due to not having to patch against the layout tables:

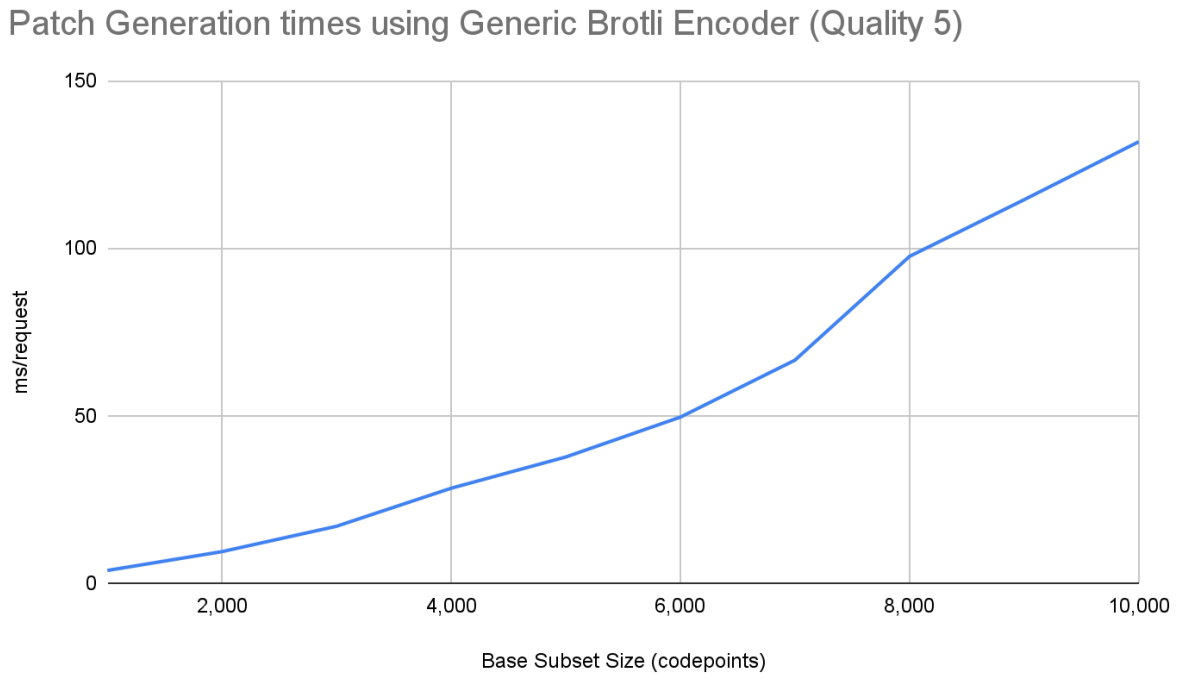


Case 2: Patch Request

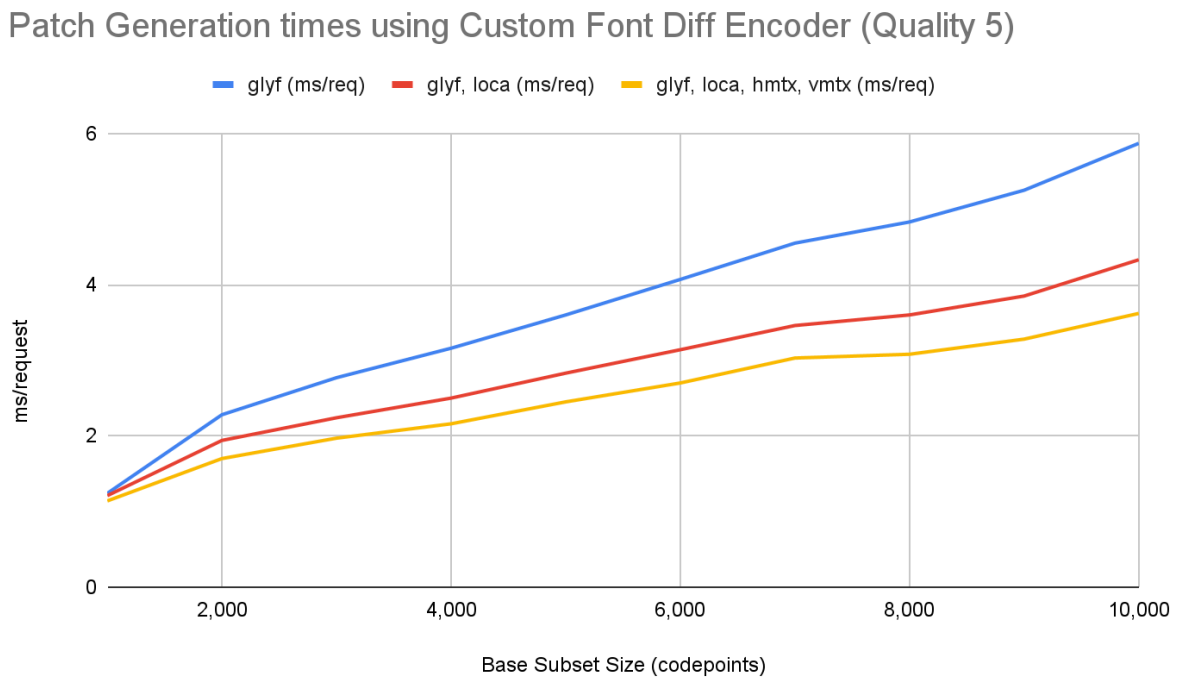
The caching of glyph blocks from the previous section can also be used in the generation of patch responses. The process is the same, identify the set of new glyphs and then choose precompressed blocks to be added to the subset. Compile the final brotli stream as a concatenation of dynamically generated and precompressed glyph data.

An additional optimization that may yield performance improvements would be to manually write out the back references for the known unchanged portions of the font (eg. all of the glyph data in the clients existing subset) to avoid the encoder having to spend time to figure them out.

Using these ideas I built a custom brotli font differ. See the design doc for a more detailed description of how it works. Then the custom encoder was used to generate patches (adding 10 codepoints) against a varying base subset size. First, here is the encoding times using the generic brotli encoder:

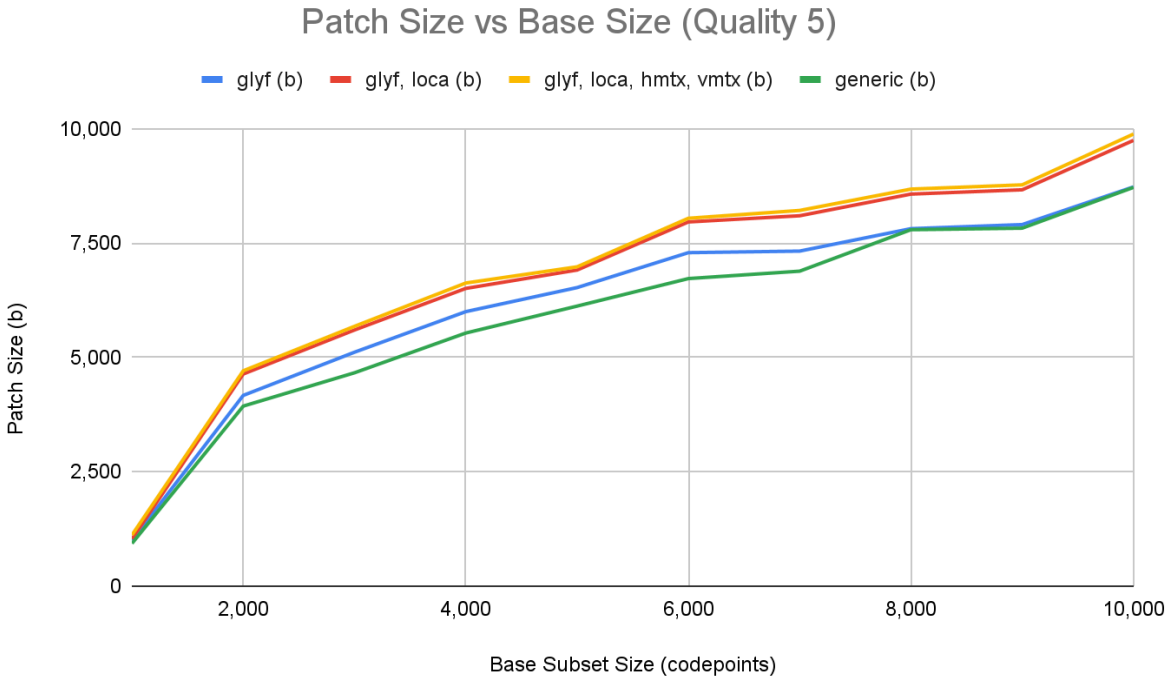


And here is the encoding times using the custom encoder:



The different lines capture the encoding times when custom encoding is used for the indicated list of tables (all other tables are encoded by the generic brotli encoder).

Here we see a massive speedup in using the custom encoder (from 130ms to 3.6ms at the largest base size). The downside is that custom generated patches are slightly larger then patches generated by the generic brotli encoder:



Experiment Code

The code used in these experiments can be found here.

Future Work

Fast and efficient recompression using previous compression artifacts provides a framework for using information from a previously compressed version of the original font to speed up compression of a subset of the original font. It should be possible to incorporate these techniques into the custom encoder developed here.

In particular we should be able to use a compressed copy of the full font file to identify where data is shared between data that is new to the derived subset (eg. new glyphs) and data that exists in the base subset, which is currently not supported in the existing implementation. This could both further speed up encoding times, and reduce the total patch size.

References

- Fast and efficient recompression using previous compression artifacts
- Brotli RFC
- Shared Brotli RFC
- Shared Brotli Diff Implementation in Incxfer