Customizing Progressive JPEG for Efficient Image Storage

Eddie Yan†, Kaiyuan Zhang†, Xi Wang†, Karin Strauss‡†, and Luis Ceze†
‡*Microsoft Research and* †*University of Washington*

Abstract

Modern image storage services, especially those associated with social media services, host massive collections of images. These images are often replicated at many different resolutions to support different devices and contexts, incurring substantial capacity overheads. One approach to alleviate these overheads is to resize them at request time. However, this approach can be inefficient, as reading full-size source images for resizing uses more bandwidth than reading pre-resized images. We propose repurposing the progressive JPEG standard and customizing the organization of image data to reduce the bandwidth overheads of dynamic resizing. We show that at a PSNR of 32 dB, dynamic resizing with progressive JPEG provides 2.5× read data savings over baseline JPEG, and that progressive JPEG with customized encode parameters can further improve these savings (up to 5.8× over the baseline). Finally, we characterize the decode overheads of progressive JPEG to assess the feasibility of directly decoding progressive JPEG images on energy-limited devices. Our approach does not require modifications to current JPEG software stacks.

1 Introduction

Images are ubiquitous on the modern web. With the rapid expansion of social media services, the largest social media networks now host billions of images [8]. Image hosts face the challenge of handling the massive rates at which users upload images, especially as scaling of cost per gigabyte slows [7]. This issue is compounded by the need to store each image at multiple resolutions to support different contexts or devices. In 2010, Facebook stored up to 4 different versions of each image [4], later reporting that *dynamic resizing* was also performed [8]. *dynamic resizing* saves capacity by generating low resolution copies of images on the fly without committing them to storage.

Faced with a similar problem, Flickr [1] switched to dynamic resizing and reported that doing so helped to eliminate the need for storage capacity upgrades for an entire year. While dynamic resizing is an attractive method for reducing storage overheads, it introduces two main trade-offs. First, computation is traded for capacity: when an uncached image is requested, the image must be decoded and resized. Second and perhaps more importantly, bandwidth is traded for capacity: reading the entire source image for resizing can waste bandwidth. Bandwidth can be precious in cold storage scenarios that sacrifice performance for cost and density [5] or when an access misses in the cache.

This paper proposes repurposing *progressive JPEG* to reduce both read bandwidth and storage overheads. The progressive JPEG standard specifies a variant of JPEG images originally designed for bandwidth-constrained networks. In a progressive JPEG image, image data is partitioned and arranged by frequency content instead of by vertical position in an image (scanline), allowing for a lossy preview before the entire image has been downloaded. We demonstrate that by repurposing progressive JPEG, a significant portion of read bandwidth can be saved by reading only the necessary image data for resizing. Additionally, we show that tuning encode-time parameters to match predefined image sizes can further reduce read bandwidth.

Finally, we characterize the cost of decoding custom progressive JPEG directly on the client relative to decoding resized baseline images. We find that the computation–bandwidth trade-off favors transcoding images on the server side, where the computational costs are comparable to a baseline dynamic resizing scheme.

2 Background: Progressive JPEG

The progressive JPEG standard was originally designed to allow partially transmitted images to be previewed [17]. Progressive JPEG works by exploiting the fact that partitioning image data in the frequency domain from low to high frequency roughly corresponds to partitioning image data from coarse to fine details. By ini-

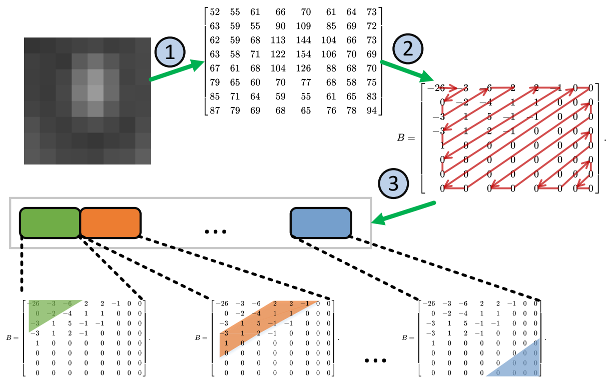

Figure 1: Sketch of Progressive JPEG Encoding: 1. Images are divided into 8×8 macroblocks. 2. Intensity values are transformed to the frequency-domain using a DCT. Red arrows indicate the low to high frequency order of coefficients. 3. Highlighted regions denote scans.

tially decoding only low frequency data, a preview can be rendered with an incomplete image file.

As with baseline JPEG images, progressive JPEG encoding involves transforming image data to the frequency domain with a discrete cosine transform (DCT). In the frequency domain, intensity values become frequency coefficients; in the case of JPEG, there are 64 coefficients for each 8×8 pixel region. Progressive JPEG partitions frequency coefficients into groups called scans. Figure 1 shows a sketch of the progressive JPEG encode process and partitioning of scans. A single scan can contain a single coefficient, an approximation of a single coefficient, multiple coefficients, or approximations of multiple coefficients; the fundamental property is that scans contain refinements of image data. Only the first scan is necessary to display a low quality image preview.

3 Approach

In order to resize a baseline JPEG image to a reduced resolution, the full image must be read. We repurpose progressive JPEG, reading only the scans necessary for a specific image quality for the resized image. To further reduce the amount of data that must be read, we tune progressive JPEG parameters to match predefined image resolutions. We specify resolutions relative to source images (e.g. 10% of a 500×500 image is a 50×50 image).

3.1 Defining Image Quality

Using progressive JPEG and dynamic resizing in place of static baseline images requires an image quality metric and quality threshold to determine when have we read enough image data. For each resolution, we define image quality using the peak signal-to-noise ratio (PSNR). To compute the PSNR, the reduced resolution image (which may be lossier) is compared against a source image scaled to the same resolution. For our experiments, we choose a PSNR threshold of 32 dB as the cutoff where no additional image data (or scans) of a progressive JPEG image needs to be read. Our technique of customizing progressive JPEG is orthogonal to the choice of quality metric and threshold, but higher quality thresholds will reduce savings.

3.2 Tuning Progressive JPEG Encoding

We used the `jpegtran` [11] transcoder, which allows the groupings of frequency coefficients and their successive approximations in scans to be customized. We implement a greedy algorithm that enumerates groupings of coefficients (scan configurations) and chooses a configuration based on the resulting PSNR value. Configurations are enumerated by adding coefficient approximations until the PSNR target is met; this process is repeated for all predefined resolutions.

The algorithm is characterized by the following pseudocode which finds the next coefficient approximation to include; some details such as color channels are omitted.

```
best_psnr = 0;
best_coeff = None;
for coeff ∈ $(0, \text{max\_coeff(config)} + \text{c\_depth})$ do
    if approx[coeff] = 0 then
        continue;
    end if
    //approx[] is initialized to a_depth
    temp_approx = approx[coeff] - 1;
    temp_config = config + (coeff, temp_approx);
    psnr = calc_psnr(source_image, temp_config);
    if psnr > best_psnr then
        best_coeff = (coeff, temp_approx);
        best_psnr = psnr;
    end if
end for
return best_coeff;
```

We tune the *coefficient depth* (`c_depth`) parameter used by the greedy algorithm, as it can reduce the search time by pruning the enumerated configurations. We find that reducing this parameter leads to more space-efficient configurations, perhaps by pruning configurations that are locally optimal (in terms of PSNR) but inefficient.

An artifact of the `jpegtran` encoder is that it is limited to at most 100 scans in a given image. This limit also effectively constrains the maximum *approximation depth* (`a_depth`) parameter for images where more scans are needed for approximation refinements. However, to our benefit, the encoder also supports specifying multiple frequency coefficients (within the same color channel) that share the same approximation level in a single scan. This feature allows us to work around the 100 scan limit in many cases; we implement a simple algorithm that identifies the longest intervals of coefficients that share approximation levels and merges these coefficients into

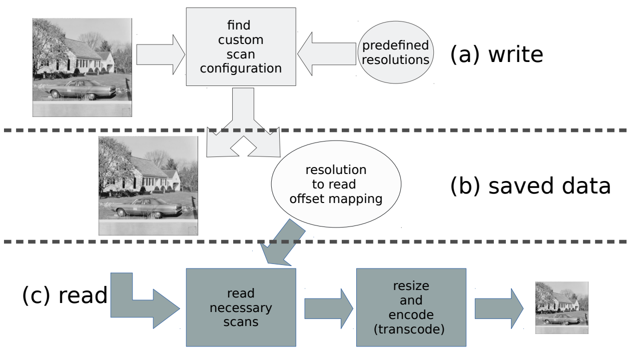
find custom scan configuration
predefined resolutions
(a) write
resolution to read offset mapping
(b) saved data
(c) read
read necessary scans
resize and encode (transcode)

Figure 2: Sketch of a dynamic resizing scheme using custom progressive JPEG. Given an input image and predefined target resolutions (a), a suitable scan configuration is found. This process produces a mapping (b) of resolutions to scans (file offsets in bytes). Given a requested image and resolution (c), the necessary scans are read and the image transcoded.

scheme	**stored data**
baseline (static)	stores source, pre-resized images
baseline (dynamic)	stores source images
progressive (dynamic) (ours, naïve)	stores source images in progressive format
custom progressive (dynamic) (ours, preferred)	stores source images in tuned progressive format

Table 1: Description of each evaluated scheme. The first two schemes represent baselines used by current systems.

single scans. "Merging" can also be done with the first (DC) coefficients across channels. Merging allows us to encode images using configurations that would otherwise exceed the 100 scan limit of the encoder. Still, when this limit is exceeded, we reduce the maximum *approximation depth* used by the greedy algorithm.

3.3 Proposed Read-Write Process

We envision a read/write scheme (Figure 2) where, at write time, files are losslessly transcoded using a custom scan configuration as they are added to the system. At read time, only the necessary scans are read before the resulting image is transcoded to baseline JPEG. The mapping between the requested resolution and how many scans to read (offset within the file) is a result of the custom progressive JPEG configuration process.

4 Evaluation

We evaluate the storage overheads (capacity, bandwidth) required for the four storage schemes shown in Table 1. For each approach, we evaluate the storage overheads when three image resolutions in addition to the original may be requested: 10%, 25%, and 50%. For our custom progressive JPEG scheme, we also evaluate the compute overheads relative to dynamic resizing with baseline JPEG images. This comparison attempts to answer the question of whether it is beneficial to offload resizing from the image host to the requesting client—an option not possible with dynamic resizing on baseline images.

4.1 Compute Overheads

Many existing JPEG decoders support progressive JPEG and custom progressive JPEG, raising the question of whether progressive JPEG images should be served directly to clients without transcoding to baseline JPEG. However, decoding progressive JPEG images is more computationally expensive than decoding (equivalent) baseline images [12]. We therefore consider the computational overheads of two schemes: (1) the preferred scheme where custom progressive JPEG images are transcoded to baseline images on the server side, and (2) where custom progressive JPEG images are served directly to the client, offloading computation from the server. Offloading transcode (2) is not possible when using baseline images as it would be equivalent to sending the entire source image. For server side transcode (1), we calculate the overhead by measuring the time to decode a custom progressive JPEG image versus a full baseline source image for resizing. For client side decode (2), we calculate overhead by measuring the time it takes to decode a custom progressive JPEG versus a previously resized baseline image.

4.2 Dataset and Encoder

We perform our evaluation with the MIRFLICKR [9] dataset, using 24,988 JPEG images with an average resolution of 462×399. We use the original images as the source baseline JPEG images and the ImageMagick `convert` [10] tool to generate resized baseline images. For progressive JPEG images, we use `jpegtran` with the `-optimize` and `-progressive` flags. For progressive JPEG images with custom scan configurations, we use `jpegtran` with the `-progressive` and `-scans [file]` flags. In all cases, `jpegtran` performs transcoding losslessly. We also iteratively reduce the quality level parameter until the PSNR drops below 32 dB to avoid inflating the capacity usage of static baseline images. Still, the quality level of the resized baseline images is not strictly equivalent; we compute PSNR on progressive images before they have been re-encoded to baseline images. For many (static baseline) images resized to 10% at a quality setting of 100, we observed that the PSNR was below 32 dB despite acceptable visual quality.

5 Results

Overall, we find that dynamic resizing dramatically reduces storage overheads significantly (by 41%). Additionally, using custom progressive JPEG for dynamic resizing yields the most efficient use of storage bandwidth.

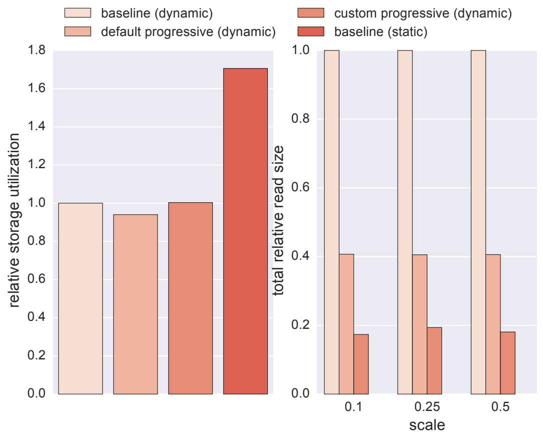

Figure 3: Storage utilization (left) and read sizes measured by the amount of data read to achieve a PSNR of at least 32 dB (right). Note that the PSNR of the resulting images with each scheme can be different despite this lower-bound: default progressive JPEG overshoots the quality target. Overall, dynamic resizing schemes provide similar and substantial storage savings over static resizing. Custom progressive JPEG provides the most bandwidth savings (up to 5.8× *vs. baseline).*

5.1 Storage Capacity

Unsurprisingly, storing baseline images along with resized images uses the most storage capacity (Figure 3). Progressive JPEG is slightly more space-efficient than baseline JPEG [15], though all dynamic resizing approaches are similar in storage utilization. Normalized to dynamic baseline JPEG, dynamic custom progressive JPEG incurs 0.3% storage overhead while dynamic progressive JPEG provides 6.0% storage savings. Custom progressive JPEG likely suffers the small additional overhead due to the increased number of scans.

5.2 Read Bandwidth

We consider the case where the requested resolutions of images are not cached[1]. We estimate the read bandwidth requirements of each method using the amount of data read necessary to achieve a satisfactory PSNR for all 24,988 images. Here, baseline pre-resized images are omitted because their PSNR values were not comparable; pre-resized images should offer competitive if not better bandwidth savings relative to custom progressive JPEG. A substantial portion of read bandwidth can be saved just by using progressive JPEG for dynamic resizing: 59% for 10% resolution, with similar improvements for other scales. Customizing progressive JPEG improves savings to 83% for the 10% resolution case. The savings in read bandwidth (Figure 3) from custom progressive JPEG can largely be explained as a PSNR–read size trade-off. This trade-off is evident for default progressive JPEG, which overshoots the quality target (reading enough scans to meet the quality target results in an average PSNR of around 37-38 dB). Interestingly, the progressive schemes seem to require roughly the same amount of read data for all three image scales; this may be a limitation of using PSNR to define image quality.

[1]If the requested resolutions were already cached, we would expect performance to be identical under each scheme.

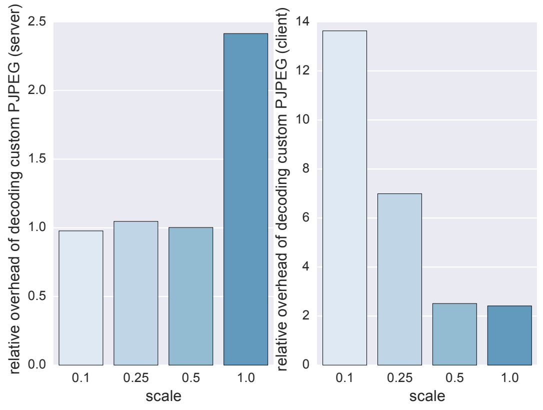

Figure 4: Relative overhead of transcoding on the server (left) and relative overhead of decoding custom progressive JPEG on the client (right).

5.3 Compute Overheads

For lower resolutions, the decode overheads (Figure 4) of custom progressive JPEG may be prohibitive (up to 13.6× slower than baseline JPEG) on the client side. However, the computational cost of decoding a baseline source image is comparable (1.0-2.4×) to that of decoding a custom progressive JPEG image. Given this compute–bandwidth trade-off, it makes more sense to transcode custom progressive JPEG images on the server than to decode custom progressive JPEG on the client.

6 Discussion

We find that customizing progressive JPEG provides a substantial advantage in terms of read size over default progressive JPEG for our quality target. One caveat is that customizing progressive JPEG relies on trading image quality for read size; there is no inherent improvement to the JPEG standard. Rather, custom progressive JPEG facilitates partitioning images at a fine granularity so that this partitioning matches quality specifications closely. In this sense, default progressive JPEG can be viewed as an lower-bound on the bandwidth savings of custom progressive JPEG: 2.5× at a 37-38 dB threshold. Decoding progressive JPEG images is also more computationally expensive (by up to 13.6×) than decoding their baseline counterparts, enough so that it does not make sense to push decoding to the client. Still, decoding progressive JPEG images partially for transcoding on the server is comparable in terms of compute to decoding full baseline images, so transcoding on the server with custom progressive JPEG remains a reasonable approach.

7 Related Work

Efficient image storage is an active field of research. Recent work has aimed to reduce overheads due to metadata for small files [4] as well as develop SSD friendly caching algorithms [16]. Related work has also investigated the quality–density trade-off for approximate storage, showing that matching the importance of image data with the reliability of storage can improve storage efficiency [6]. Using custom progressive JPEG limits metadata overheads when only storing a single version of each image and can improve caching behavior as different versions of an image share data. Grouping scans of progressive JPEG is related to ordering image data from most to least important, but the binary format used here is not amenable to storage on approximate storage media.

Progressive JPEG images can also be partially deleted gracefully by discarding high frequency data first—improving storage elasticity. The concept of motifs: descriptions of computation needed to reconstruct a file discussed in [13, 14] is implicitly implemented by a dynamic resizing storage scheme as only the highest quality version of an image is stored while lower quality versions are implicitly defined by motifs.

Dynamic resizing has precursors in image processing systems such as zimg [2] that allow clients to upload and request images with added operations such as cropping and scaling. To the best of our knowledge, these systems do not vary the amount of data read based on quality via a progressive frequency domain encoding. Dynamic resizing has also been used by Flickr [1] and Facebook [8]: in addition to storing multiple versions of each photo, Facebook incorporates "Resizers" when the requested version requires additional processing. Finally, progressive JPEG has been recently used by Facebook [3] to reduce data consumption and speed up the apparent loading of images on the client side; the latter is achieved by rendering an acceptable quality scan before all scans have been transmitted. However, this approach does not involve dynamic resizing or customizing progressive JPEG.

8 Limitations and Future Work

A limitation of our current implementation is the cost of evaluating custom scan configurations. Our naïve implementation takes days to process 24,988 images on 8 cluster nodes (12 cores/12 threads per node) with Westmere-class CPUs. We suspect that this time can drastically reduced without sacrificing significant bandwidth savings by aggressively pruning the search space or applying machine learning techniques to choose scan configurations. Note that while the customization process for progressive JPEG is currently expensive, it only needs to be performed once, at write time.

The PSNR metric is limited in its relevance to perceived visual quality [18]. We often found that the PSNR of higher resolution resizes was higher than that of lower resolution resizes even with less image data read—this issue may be mitigated with conservative PSNR thresholds. To the best of our knowledge, there is no standard, widely used method of computing the image quality of a resized image derived from a source image.

Finally, an issue when using progressive JPEG for dynamic resizing is the *minimum* resolution of the resized images. Progressive JPEG is less efficient in terms of read bandwidth for resizes smaller than 10% of the source image, which may limit savings when the source images are much higher in resolution than their resized versions. This threshold is due to JPEG's use of 8×8 macroblocks: even a single frequency coefficient represents at least $\frac{1}{64}$ of the total image data. Even when approximations are used, this approach may require more read bandwidth than pre-resized images. Still, using progressive JPEG should be more space-efficient than baseline JPEG for dynamic resizing.

Future Work We expect that a solution to reduce the cost of enumerating custom JPEG scan configurations will be to prune the search space to a much smaller subset of likely "good" configurations. It may be possible to obtain comparable results by only trying a few custom scan configurations per image—with this subset being determined by identifying the best configurations when naïvely re-encoding a larger dataset of images. Along these lines, even choosing from a larger pool of configurations may be tractable if a machine learning model is applied to each image to choose the best configuration.

9 Conclusion

Faced with growing demand for storage, image hosting services are increasingly turning to dynamic image resizing to improve the efficiency of image storage. We showed that progressive encodings can dramatically reduce the amount of data that needs to be read for resizing images—potentially saving over 80% of read bandwidth when tuned encode-time parameters used. Finally, we give an estimate of progressive JPEG decode overheads which suggests that while serving custom progressive images directly to energy-constrained devices is difficult to justify, transcoding custom progressive JPEG on the server side incurs acceptable overheads.

Acknowledgements

We thank the members of the Sampa group and the anonymous reviewers for their feedback. This work was supported in part by a gift from Intel Corporation and by DARPA under contract FA8750-16-2-0032.

References

[1] A year without a byte. `https://code.flickr.net/2017/01/05/a-year-without-a-byte/`.

[2] zimg - a lightweight and high performance image storage and processing system. `http://zimg.buaa.us/`.

[3] BAR, T. Faster photos in Facebook for iOS. `https://code.facebook.com/posts/857662304298232/faster-photos-in-facebook-for-ios/`.

[4] BEAVER, D., KUMAR, S., LI, H. C., SOBEL, J., VAJGEL, P., ET AL. Finding a needle in Haystack: Facebook's photo storage. In *OSDI* (2010), vol. 10, pp. 1–8.

[5] BLACK, R., DONNELLY, A., HARPER, D., OGUS, A., AND ROWSTRON, A. Feeding the pelican: using archival hard drives for cold storage racks. In *8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 16)* (2016), USENIX Association.

[6] GUO, Q., STRAUSS, K., CEZE, L., AND MALVAR, H. S. High-density image storage using approximate memory cells. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems* (2016), ACM, pp. 413–426.

[7] GUPTA, P., WILDANI, A., MILLER, E. L., ROSENTHAL, D., ADAMS, I. F., STRONG, C., AND HOSPODOR, A. An economic perspective of disk vs. flash media in archival storage. In *IEEE MASCOTS* (2014).

[8] HUANG, Q., BIRMAN, K., VAN RENESSE, R., LLOYD, W., KUMAR, S., AND LI, H. C. An analysis of Facebook photo caching. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (2013), ACM, pp. 167–181.

[9] HUISKES, M. J., AND LEW, M. S. The MIR Flickr retrieval evaluation. In *MIR '08: Proceedings of the 2008 ACM International Conference on Multimedia Information Retrieval* (New York, NY, USA, 2008), ACM.

[10] IMAGEMAGICK STUDIO, LLC. Imagemagick, 2008.

[11] INDEPENDENT JPEG GROUP AND OTHERS. Libjpeg, 2014.

[12] LANE, T. JPEG image compression faq, part 1/2. `http://www.faqs.org/faqs/jpeg-faq/part1/index.html`.

[13] SIGURBJARNARSON, H., RAGNARSSON, P. O., VIGFUSSON, Y., AND BALAKRISHNAN, M. Harmonium: Elastic cloud storage via file motifs. In *6th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 14)* (Philadelphia, PA, 2014), USENIX Association.

[14] SIGURBJARNARSON, H., RAGNARSSON, P. O., YANG, J., VIGFUSSON, Y., AND BALAKRISHNAN, M. Enabling space elasticity in storage systems. In *9th ACM International on Systems and Storage Conference (SYSTOR)* (Haifa, Israel, 2016).

[15] SOUDERS, S. *Even faster web sites: performance best practices for web developers*. O'Reilly Media, Inc., 2009.

[16] TANG, L., HUANG, Q., LLOYD, W., KUMAR, S., AND LI, K. RIPQ: Advanced photo caching on flash for Facebook. In *13th USENIX Conference on File and Storage Technologies (FAST 15)* (2015), pp. 373–386.

[17] WALLACE, G. K. The JPEG still picture compression standard. *IEEE transactions on consumer electronics 38*, 1 (1992), xviii–xxxiv.

[18] WANG, Z., SHEIKH, H. R., AND BOVIK, A. C. No-reference perceptual quality assessment of JPEG compressed images. In *IEEE International Conference on Image Processing* (2002), vol. 1, IEEE.