Partial Object Caching

Introduction

Partial object caching, or more accurately range caching, is the ability of Traffic Server to cache range responses. This has two parts:

1. Cache the data from a 206 (range) response.
2. Serve, if possible, a range request from cached range responses.

Issues

The primary problem for range caching is tracking the ranges for the overall object. There are many well known data structures for storing such range information but it is considered critical for the Traffic Server cache that the memory footprint of the data be bounded.

It is also desirable to limit fragmentation of the stored data. If an object is scattered in a large number of small disk segments then performance will be suboptimal.

Performance requirements also mean that it should take a small fixed number (preferably 1) of disk reads to determine if a request can be satisfied with the current object range data.

Implementations

Two basic implementations have been discussed, both of them designed with the idea of bounding the range tracking data at the possible expense of discarding data. Both of them are monotonically increasing in that the amount of data stored for an object never decreases, even if not every byte that passes through ATS is cached.

Plan A is to use torrent style storage. Each segment of an object is divided into chunks (fixed number of chunks per segment). Range data is stored in chunk units and only complete chunks. Data in partial chunks is not stored. A bitmap is kept along with the segment offset data indicating which chunks have data. This can then be easily checked against an incoming request to see if it can be served.

Plan B is to use a span per segment. Each segment stores a low and high water mark of valid data. If an incoming request overlaps or is adjacent to this span, it is added and the span marks adjusted. If there is no overlap, the larger of the existing span or the incoming data is kept and the other discarded.

For both plans, the extra data is a 64bit word per segment.

Which is better depends on how the range requests are expected to be distributed. If they are roughly random, plan A is probably better. If they are (roughly) sequential plan B would do better. It is possible, if unlikely, for plan A to fail to store any data for a full sequence of range requests for the object if every request spans two chunks but covers neither of them.

Radical Idea

Presuming the range information would be stored in the root segment, memory bounding may not be necessary. This data is rewritten frequently in normal operation, such as for a refresh or the addition of an alternate. If we are willing to do a rewrite for a range request then we can have much weaker space bounds on the memory consumed by the range tracking. This would require a read from disk on a cache hit but that is commonly done now. The key difference would be treating a cache "miss" differently, although the mechanism used for an alternate miss may work as well.

Plugin Implementation

An alternative to modifying the cache is to implement a partial object caching plugin on top of the existing cache. The fundamental idea here is that the plugin fetches fixed-size "chunks" from the origin and uses those to compose the range requested by the client. For each client request, the plugin

- determines the set of chunks that overlap the range
- issues a new set of origin requests (in parallel) to fetch the corresponding chunks
- rewrites the cache key for each chunk so they are all stored independently
- creates a transformed client response by composing the correct byte ranges from the chunks

This would be a fairly complex plugin, but it has the advantage of not changing the cache format and minimizes the risk of breaking other features.

Summit Proposed Implementation

Partial Object Caching

Partial object caching is really the caching of range requests for the same object as a single object. It is now common for large objects that should be cached to never be retrieved in full, only one range at a time.
This design should enable caching range requests as part of a single object with only minimal changes to the cache internals.

We want this design to have a few essential properties.

* Data requests to the origin server must not be much larger than the original client request.
* Handle range requests for an object in any order with any overlap.
* Serve ranges requests from cache if the range requested is already in the cache.
* Minimal additional overhead of data and computation.

Normally a large object is stored as an ordered list of fragments each of roughly the same size. For a partial object we fix this size for a particular object. Different objects may have different fragment sizes but we require all fragments of specific partial object to be identical in size, except possibly for the last fragment. This is normally the case for all multiple fragment objects although it is not an explicit requirement. The only change here is that we will need to store the fixed fragment size in the metadata of the first fragment per alternate for a partial object.

Given that fixed fragment size we then (conceptually) divide the object in to ranges of that size. We say such a fragment is "touched" by a range request if any byte of that request is in the fragment. Because we fix the fragment size we can skip untouched fragments without losing the ability to insert those fragments later.

..sidebar:
Because fragments are computationally chained it is not possible to insert a fragment. The ordering of fragment cache ids is fixed by the cache id of the earliest fragment.

Range requests are processed by adjusting the size seen by the origin server to always be an integral multiple of the fragment size with a offset that is also an integral multiple of the fragment size. This can mean expanding or shrinking the request dependent on how it intersects already cached fragments. The altered size is chosen to be the minimum size that covers all of the untouched fragments while satisfying the size and offset requirements. If this yields a size that cover the entire object then the object will not be partially cached but will be retrieved in full.

The client always receives the originally requested range.

With this change we then use the fragment offset table to store the fragment presence. Because we have adjusted the request to the origin server a fragment will always be either all present or not present at all in the cache. Because the fragment size is fixed, we can compute the fragment offsets without having to store them, therefore we can use them purely as a marker while still allowing normal range acceleration operation if the requested range is covered by cached fragments.

Another minor change is that we permit an empty earliest fragment if the first range request does not touch that fragment. This is necessary because the cache logic assumes the earliest fragment is also closest to the stripe write cursor and will therefore be overwritten before any other fragment. This allows validity checking before any cache data is served. We must preserve that and so will write an empty earliest fragment even if it is not retrieved from the origin server. The current logic always reads the earliest fragment (if distinct from the first fragment) so doing this keeps the same logic for checking a partial object. We will need to make a note in the first fragment metadata about this so that the fragment table offsets can be done correctly (in effect this will shift the indices by one). Note that the earliest fragment is entirely empty (no content), a full fragment (if there other fragments), or a partial fragment "only if" it contains the entire object (i.e. it is also the last fragment).

As range requests are processed for a partial object additional fragments are written and the fragment is updated. This keeps all of the content fragments bookmarked between the earliest fragment and the first fragment as for full objects. It also allows the fragment offset table to be extended as needed if fragments beyond the current end are retrieved.

..note:
An open question is the handling of 304 responses and object refresh in general. If the client makes a range request that touches previously untouched fragments we can't attach an "If-Modified-Since" field to it. For a range request that can be satisfied from data in the cache, I don't know if we can attach an "If-Modified-Since". If that is possible then I presume we would treat the entire object uniformly, that is all bytes are valid or stale, not just the fragments touched by the client request.

A related issue is if we get different expiration data for different requests to the origin server. Should the object be updated with the new expiration data or keep the original?

..note:
A serious issue with caching range requests is tracking the valid bytes. It is to avoid this problem that we adjust the size of the range request to the origin server. By limiting it to being more than 2 fragment sizes larger than the original request we avoid excessive network traffic beyond the client request while guaranteeing we need only track the fragments, as is already done for full objects.

..note:
Although we must rewrite the first fragment on every request that adds fragments to the partial object we still need to be a bit careful about the full size so that we don't write a partial fragment in the false belief that it is the last fragment.

..note:
It is easiest to use the target fragment size for the fragment size of a partial object but this is in no way required. Currently I don't think it is a useful feature to be able to tune this independently but it would require very little extra work to implement. The main concern would be the additional complexity of configuration as seen by the user, although we could default it to zero which would mean "use the target fragment size". Then it would need to be configured only by those who had an interest in doing so.