RamCache

Ram Cache

New Ram Cache Algorithm (CLFUS)

The new Ram Cache (TS-120) uses ideas from a number of cache replacement policies and algorithms, including LRU, LFU, CLOCK, GDFS and 2Q. I call it CLFUS (Clocked Least Frequently Used by Size). It avoids any patented algorithms (naughty IBM). And includes the following features:

- Balances Recentness, Frequency and Size to maximize hit rate (not byte hit rate).
- Is Scan Resistant and extracts robust hit rates even when the working set does not fit in the Ram Cache.
- Supports compression at 3 levels fastlz, gzip(libz), and xz(liblzma). Compression can be moved to another thread.
- Has very low CPU overhead, only little more than a basic LRU. Rather than using an O(lg n) heap, it uses a probabilistic replacement policy for O (1) cost with low C.
- Has relatively low memory overhead of approximately 200 bytes per object in memory.

The rational for emphasizing hit rate over byte hit rate is that the overhead of pulling more bytes from secondary storage is low compared to the cost of a request.

The Ram Cache consists of an object hash fronting 2 LRU/CLOCK lists and a "Seen" hash table. The first "Cached" list contains objects in memory while the second contains a "History" of objects which have either recently been in memory or are being considered for keeping in memory. The "Seen" hash table is used to make the algorithm scan resistant.

The list entries record the following information:

- key - 16 byte unique object identifier
- auxkeys - 8 bytes worth of version number (in our system the block in the partition). When the version of an object changes old entries are purged from the cache.
- hits - number of hits within this clock period
- size - the size of the object in the cache
- len - the actual length of the object (differs from size because of compression and padding)
- compressed_len - the compressed length of the object
- compressed (none, fastlz, libz, liblzma)
- uncompressible (flag)
- copy - whether or not this object should be copied in and copied out (e.g. HTTP HDR)
- LRU link
- HASH link
- IOBufferData (smart point to the data buffer)

The interface to the cache is Get and Put operations. Get operations check if an object is in the cache and are called on a read attempt. The Put operation decides whether or not to cache the provided object in memory. It is called after a read from secondary storage.

Seen Hash

The Seen List becomes active after the Cached and History lists become full after a cold start. The purpose is to make the cache scan resistant which means that the cache state must not be effected at all by a long sequence Get and Put operations on objects which are seen only once. This is essential, without it not only would the cache be polluted, but it could lose critical information about the objects that it cares about. It is therefore essential that the Cache and History lists are not effected by Get or Put operations on objects seen the first time. The Seen Hash maintains a set of 16 bit hash tags, and requests which do not hit in the object cache (are in the Cache List or History List) and do not match the hash tag result in the hash tag begin updated but are otherwise ignored. The Seen Hash is sized to approximately the number of objects in the cache in order to match the number that are passed through it with the CLOCK rate of the Cached and History Lists.

Cached List

The Cached list contains objects actually in memory. The basic operation is LRU with new entries inserted into a FIFO (queue) and hits causing objects to be reinserted. The interesting bit comes when an object is being considered for insertion. First we check if the Object Hash to see if the object is in the Cached List or History. Hits result in updating the "hit" field and reinsertion. History hits result in the "hit" field being updated and a comparison to see if this object should be kept in memory. The comparison is against the least recently used members of the Cache List, and is based on a weighted frequency:

CACHE_VALUE = hits / (size + overhead)

A new object must beat enough bytes worth of currently cached objects to cover itself. Each time an object is considered for replacement the CLOCK moves forward. If the History object has a greater value then it is inserted into the Cached List and the replaced objects are removed from memory and their list entries are inserted into the History List. If the History object has a lesser value it is reinserted into the History List. Objects considered for replacement (at least one) but not replaced have their "hits" field set to zero and are reinserted into the Cached List. This is the CLOCK operation on the Cached List.

History List

Each CLOCK the least recently used entry in the History List is dequeued and if the "hits" field is not greater than 1 (it was hit at least once in the History or Cached List) it is deleted, otherwise the "hits" is set to zero and it is requeued on the History List.

Compression/Decompression

Compression is performed by a background operation (currently called as part of Put) which maintains a pointer into the Cached List and runs toward the head compressing entries. Decompression occurs on demand during a Get. In the case of objects tagged "copy" the compressed version is reinserted in the LRU since we need to make a copy anyway. Those not tagged "copy" are inserted uncompressed in the hope that they can be reused in uncompressed form. This is a compile time option and may be something we want to change.

There are 3 algorithms and levels of compression (speed on 1 thread i7 920) :

- fastlz: 173 MB/sec compression, 442 MB/sec decompression : basically free since disk or network will limit first, ~53% final size
- libz: 55 MB/sec compression, 234 MB/sec decompression : almost free, particularly decompression, ~37% final size
- liblzma: 3 MB/sec compression, 50 MB/sec decompression : expensive, ~27% final size

These are ballpark numbers, and your millage will vary enormously. JPEG for example will not compress with any of these. The RamCache does detect compression level and will declare something "incompressible" if it doesn't get below 90% of the original size. This value is cached so that the RamCache will not attempt to compress it again (at least as long as it is in the history).

ram cache and memory issues and roadmap

- how to control the memory usage by hostname etc
- can we use share memory to host ram cache? which will help us get high memory hit ratio even after a restart
- can we unbind the ram cache and vol? which may make ram cache a better performance and easy to use, for example in ET_CLUSTER we can read from ram cache directly if hit, otherwise we can shcedule an cache read.
- can we setup an ram cache space for origin side cache before it is synced to disk, and we can set for better checking for example we can track xxM objects in 2G ram, and only write to disk when that object is hit by 2 times.