RFC: Metadata Cache Image

John Mainzer

HDF5 metadata is typically small, and scattered throughout the HDF5 file. While small, widely scattered I/Os are not a significant issue on small machines with local file systems, they are a major performance concern on large HPC systems. The metadata cache does a reasonably good job of minimizing such I/Os during normal operation. However, the cache must still be populated at file open, and flushed at file close. Thus the metadata I/O overhead of simply opening and closing HDF5 files on such systems is a concern.

This RFC proposes writing the contents of the metadata cache to file in a single block on file close, and then populating the cache with the contents of this block on file open – thus avoiding the many small I/Os that would otherwise be required on file open and close.

1. Introduction

For historical reasons, elements of metadata in HDF5 are of no fixed size, and may be arbitrarily large. Most entries are small, and thus HDF5 generates numerous small metadata reads and writes. The metadata cache minimizes this number, but enough are still issued to cause problems on large HPC systems. In particular, such reads and writes are currently unavoidable at file open and close, as the metadata cache must be populated on file open, and (if the file has been modified) flushed on file close.

This RFC explores the notion of avoiding small metadata writes on file close by writing the contents of the metadata cache to file in a single block. On file open, this block would be read immediately, and used to populate the metadata cache before any metadata access requests are received from the library. If the access pattern after the file open is similar to that just before the file close, this approach could avoid the majority of metadata I/O on file open as well.

The immediate impetus for this RFC is a use case in which many processes in an HPC environment access a single HDF5 file in a round robin. Specifically, each process opens the file, writes to it, closes it, and then passes control of the file to the next process. As the HDF5 file is opened and closed many times during processing, reduction of file open/close overhead is a major concern.

2. Cycle of Operation

To clarify the proposed enhancement (now largely implemented), consider the following cycle of operation.

If a metadata cache image is desired, the HDF5 file is opened (or created) with a new FAPL (File Access Property List) property indicating that the contents of the metadata cache should be written to an image on file close, instead of the usual processing in which dirty entries are written back to their assigned locations in the file and clean entries are simply discarded.

This new FAPL entry has no effect until file close, at which point processing proceeds as follows:

1. The metadata cache serializes all entries in the cache so as to fix all entry file locations and sizes.
2. The metadata cache scans each entry in the cache, and determines which entries will be included in the metadata cache image. For each such entry, it makes note of the following information:
    - Its position in the LRU list if it is on that list.
    - Whether it is dirty.
    - If the entry is a child in a flush dependency relationship, the address(s) of the parent(s).
    - If the entry is a parent in a flush dependency relationship, how many children it has.

    At a minimum, the superblock and the metadata supporting superblock extension messages must be excluded from the cache image, as the HDF5 file cannot be opened if these pieces of metadata are not in the expected locations. In principle, all other metadata can be included in the cache image.
3. The metadata cache allocates a buffer large enough for serialized representations of all entries in the cache that have been selected for inclusion in the metadata cache image, along with additional information indicating the address, length, assigned ring, and type of each entry, and also the information collected in item 2 above. This buffer must also be large enough to contain the current adaptive cache resizing configuration and status if desired.
4. The metadata cache creates a super block extension message indicating that the contents of the metadata cache has been written to a cache image. Note that at this point, the message will not contain the correct base address and length of the metadata cache image.
5. The metadata cache allocates space for the metadata cache image at the end of the HDF5 file. This space is the same size as the buffer allocated in 3 above.
6. The metadata cache updates the super block extension message created in 4 above to contain the base address and length of the metadata cache image.
7. The metadata cache is then flushed as usual, with the proviso that all entries selected for inclusion in the metadata cache image are written to the buffer (annotated with base address, length, type, etc.). Super block related entries (and all other entries excluded from the cache image) must be written to file in their usual places, as they will needed for file open,
8. After the flush and if so directed, the metadata cache writes the current adaptive cache resizing status to the buffer (this is not implemented at present).
9. Finally, the metadata cache writes the cache image buffer to its allocated space in the HDF5 file, and frees the buffer.
10. File close then proceeds as normal.

Note that the above cycle of operation is simplified conceptual overview. Deltas of particular interest are discussed in the implementation details section.

File open proceeds as usual up to the point at which the super block extensions are read.

If the version of the library that is used to open the file does not understand the metadata cache image super block extension, it must refuse to open the file.

If the library does understand the metadata cache image super block extension, it must advise the metadata cache of the existence, base address, and size of the cache image, and then delete the metadata cache image superblock extension message if the file has been opened read/write.

Once so advised, the metadata cache must proceed as follows prior to the first entry protect (or just prior to file close, if the file is closed without any further activity):

1. Allocate a buffer for the cache image, and load the cache image from file.
2. Scan the metadata cache image, and create a "prefetched" cache entry for each entry in the image. Note that these entries are be different from metadata cache entries in the existing cache, in that they contain only the on disk image of the entry, not the in core representation that is created when an entry is loaded from disk at the request of a cache client. Call these entries prefetched entries. Mark each new entry with the address, length, ring, type, dirty flag, order in the LRU (if defined), flush dependency parents (if any), and number of flush dependency children (if any) recorded in the metadata cache image block. Place all the serialized entries in a linked list for ease of scanning. Call this list the prefetched entries list.
3. Scan the prefetched entries list, insert all entries in the index, and insert all dirty entries in the slist. Recall that the metadata cache uses a skip list to maintain a list of all dirty entries in increasing address order. On cache flush, it uses this list to write entries in increasing address order to the extent permitted by flush dependencies.
4. Scan the prefetched entries list to set up the flush dependencies specified. Pin all entries that are parents in flush dependency relationships, moving these entries from the prefetched entries list to the pinned entries list. Note that when this operation is complete, all entries remaining in the prefetched entries list that were not manually pinned should be annotated with their order in the LRU. Note also that the flush dependencies created will be slightly different that the usual flush dependencies, in that the metadata cache must decide when to create and destroy them, instead of delegating this issue to the clients. For clarity, call these flush dependencies "reloaded flush dependencies", to distinguish them from the flush dependencies created and managed by cache clients.
5. Scan the remaining entries in the prefetched entries list, and insert them in the LRU in the indicated order. At this point the prefetched entries list should be empty.
6. If it is included, read the adaptive cache resizing data from the cache image buffer, and configure the metadata cache to recreate the configuration and status recorded. This is not implemented at present.
7. Free the buffer containing the metadata cache image, and release the file space it resided in.

As before, the above is a simplified conceptual overview of the actual processing. See the implementation details section for expansions on points of interest.

Note that the existence in the metadata cache of prefetched entries modifies the behavior of the cache as described below:

1. If a cache client requests a prefetched entry, the cache skips the usual read of the serialized version of the entry from file, and instead passes the prefetched entry image to the client deserialize callback, and replaces the prefetched entry with the regular entry returned by that callback. If the prefetched entry is a child in a reloaded flush dependency, that dependency is destroyed before the call to the deseriailize callback. If the prefetched entry is a parent in one

or more reloaded flush dependencies, those relationships are transferred to the regular entry returned by the deserialize callback.

2. If a prefetched entry is flushed prior to any request by a cache client, the image of the entry is simply written to file and marked clean without any call to any client callback.
3. If a prefetched entry is evicted prior to any request for it by a cache client, the eviction is performed without any call to any client callback. If the entry is a child in a reloaded flush dependency, this dependency is destroyed just prior to the eviction. Note that the prefetched entry cannot be a parent in a reloaded flush dependency, as parents in flush dependencies cannot be evicted until all of their children have been evicted – at which point the entry is no longer a parent in a flush dependency.

Note that we have not discussed any provision for controlling the size of the metadata cache image. Such a facility is superfluous, as the size of the metadata cache image is implied by the metadata cache size, and there are already facilities to control the size of the metadata cache.

A more important issue is how to prevent entries that haven't been used in many open/close cycles from accumulating in the cache image, and increasing the image store / load overhead. This can be done by allowing specification of a maximum number of file open/close cycles during which a given prefetched entry may appear in subsequent cache images. As shall be seen, we have included API and file format changes required to implement this limit. However the supporting code is not implemented as of this writing.

In the parallel case, the cache image is created and written by process 0 only, and contains the contents and (if so directed) adaptive cache resizing status of that cache. This image is read by process 0 only on file open, and then broadcast to all other processes. With these exceptions, changes to processing are the same as outlined above.

As the metadata cache image enhancement observes flush dependencies, it should be transparent to SWMR.

Finally, note the proposed store and restore of metadata cache adaptive resize status. When implemented, this will have the effect of allowing the metadata cache to adapt to the stream of cache accesses across the sequence of processes that open and close the file. Assuming that the pattern of cache accesses is relatively homogeneous across processes, this should allow the metadata cache (and the metadata cache image) to adapt in size to hold the current working set – with the implied reduction in metadata I/O.

3. Additions to the API

3.1 Property List Operations

If a metadata cache image is desired, it must be requested at file open or file create in the FAPL (File Access Property List).

The signatures for the calls for getting and setting this property are:

```
herr_t H5Pset_mdc_image_config(hid_t plist_id,
                    H5AC_cache_image_config_t * config_ptr);

herr_t H5Pget_mdc_image_config(hid_t plist_id,
                    H5AC_cache_image_config_t * config_ptr);
```

Where H5AC_cache_image_config_t is defined as follows:

```
typedef struct H5AC_cache_image_config_t {
   int32_t                    version;
   hbool_t                    generate_image;
   hbool_t                    save_resize_status;
   int32_t                    entry_ageout;
} H5AC_cache_image_config_t;
```

The version field should be set to H5AC__CURR_CACHE_IMAGE_CONFIG_VERSION, and the generate_image field should be set to either TRUE or FALSE depending on whether a cache image is desired. The save_resize_status and entry_ageout fields are ignored at present.

When implemented, the save_resize_status field will control whether the adaptive resize configuration and status are stored in the cache image, and restored when the file is opened. The default value is FALSE.

The entry_ageout field will allow the user to specify the number of times a cache entry can appear in subsequent cache images (created in subsequent file closes) without being accessed. The default value is -1, which indicates that the entry may appear in an indefinitely long sequence of cache images. When implemented, this feature should allow the user avoid the case in which the cache image fills up with infrequently used entries over a long sequence of close / open cycles. The maximum value of this field is 100.

While it is an obvious error to request a cache image when opening the file read only, it is not in general possible to test for this error in the H5Pset_mdc_image_config() call. Rather than fail the subsequent file open, we have elected to resolve the issue by silently ignoring the file image request in this case.

It is also an error to request a cache image on a file that does not support superblock extension messages (i.e. superblock version less than 2). As above, it is not always possible to detect this error in the H5Pset_mdc_image_config() call, and thus we fail silently in this case as well.

Finally, at present, creation of a cache image is not supported in the parallel case. In this case as well, requests for a cache image will fail silently.

3.2 Cache Image Information Call

As some tools need to know whether a cache image exists, we have also added the following call to provide the needed information:

```
herr_t H5Fget_mdc_image_info(hid_t file_id, haddr_t *image_addr, hsize_t *image_len);
```

Where file_id is the id of the open HDF5 file. On return, *image_addr and *image_len should contain the offset and length of the cache image if it exists, or HADDR_UNDEF and zero if no file image exists.

When a hdf5 file is opened R/W, any metadata cache image will be read and deleted from the file on the first metadata cache access (or, if persistent free space managers are enabled, on the first file space allocation / de-allocation, or read of free space manager status, whichever comes first).

Thus, if the file is opened R/W, this function should be called immediately after file open and before any other operation. If H5Fget_mdc_image_info() is called after the cache image is loaded, the function will correctly report that no cache image exists, as image will have already been read and deleted from the file.

In the R/O case, the function may be called at any time, as any cache image will not be deleted from the file.

4. Implementation Details

While the above "Cycle of Operation" provides a good conceptual outline of the proposed Metadata Cache Image enhancement, some implementation details are glossed over in that section. These details are addressed in this section. Note that as implementation is not fully complete as of this writing, some details are still not fully developed.

4.1 Metadata Cache Image Super Block Extension Message

The metadata cache image super block extension message indicates the presence of a cache image by its existence – thus it need only contain the base address and length of the image. The file format is as follows:

Name: Metadata Cache Image Message

Header Message Type: 0x0017

Length: Fixed

Status: Optional, may not be repeated.

Description: This message indicates the existence, location, and size of a metadata cache image. It is *only* found in the super block extension. Versions of the library that do not understand this message **must** refuse to open files in which it appears. Thus bits 3 (fail if unknown and opened for write) and 7 (fail if unknown always) in the Header Message Flags for this message **must** be set.

Format of Data:

Metadata Cache Image Message:

byte	byte	byte	byte
Version	No space allocated – table alignment only		
Offset^{O}			
Length^{L}			

4.2 Metadata Cache Image File Format

As currently implemented, the metadata cache image is a single block of memory typically allocated at the end of the file.

As the metadata cache image must contain a representation not only of the contents of the metadata cache, but also its current adaptive resizing configuration and status (if requested), the proposed format of the image is somewhat complex.

In an attempt to make this format more readable, it is presented in hierarchical format, with the top level showing the overall format of the image, and with two sub-formats showing the formats of cache entries and the adaptive cache resizing configuration and status respectively.

The top level format follows:

Metadata Cache Image:

byte	byte	byte	byte
Signature			
Version	Flags	No space allocated – table alignment only	
Image Data Length^L			
num_entries			
Entry image 0			
.			
.			
.			
Entry image n			
Resize status (if present)			
checksum			

The fields of the top level format described in the following table. Recall that the “Entry image” and “Resize status” fields are sub-formats embedded in the Metadata Cache Image format.

Field Name:	Description:
Signature	Magic number indicating that this is a metadata cache image. Must be set to 'MDCI'.
Version	Version of the Metadata Cache Image. At present, only version 0 is defined.
Flags	Flags indicating various properties of the entry: bit 0 Set if and only if image contains resize status. All other bits reserved.
Image Data Length^L	Length of metadata cache image in bytes. Typically, this value will be the same as the cache image block length recorded in the cache image message. However it may be smaller if the cache image does not fill the entire cache image block.
num_entries	The number of metadata cache entries whose images are stored in the

	metadata cache image.
Entry image n	Image of the n'th entry image stored in the metadata cache image. See “Metadata Cache Entry Image” below for the details of these fields.
Resize status	Configuration and status of the adaptive metadata cache resize algorithms on the imaged metadata cache. Note that this field is present if and only if bit 0 of the flags field is set. See “Metadata Cache Adaptive Resize Status Image” below for the details of this field.
checksum	Checksum of the contents of the Metadata Cache Image.

The Metadata Cache Entry Image is a variable length format, each instance of which contains the serialized image of an entry, along with other data required to reconstruct the entry when the cache image is reloaded. Note that the variable length parts are the list of dependency parent offsets and the serialized entry image. The lengths of these fields are indicated by the Dependency Parent Count and Length fields respectively.

Metadata Cache Entry Image:

byte	byte	byte	byte
Signature			
Type	Flags	Ring	Age
Dependency Child Count		Dirty Dependency Child Count	
Dependency Parent Count		No space allocated	
Index in LRU			
Offset^{O}			
Length^{L}			
Dependency Parent Offsets^{O}			
Entry Image			

Field Name:	Description:
Signature	Magic number indicating that this is a metadata cache entry image. Must be set to 'MCEI'.
Type	Value of the id field of the instance of H5C_class_t associated with the entry. This field is stored primarily for sanity checking.
Flags	Flags indicating various properties of the entry:

	bit 0 If set, entry is dirty. bit 1 if set, entry is in LRU bit 2 If set, entry is a flush dependency parent. bit 3 If set, entry is a flush dependency child. All other bits reserved.
Ring	Integer indicating the flush ordering ring to which this entry is assigned.
Age	Number of times that a prefetched entry has appeared in subsequent cache images, or 0 if the entry was a created from a regular entry.
Dependency Child Count	If bit 2 above is set, the number of flush dependency children of the entry. Otherwise 0.
Dirty Dependency Child Count	If bit 2 above is set, the number of dirty flush dependency children of the entry. Otherwise 0.
Dependency Parent Count	If bit 3 above is set, the number of flush dependency parents of this entry. Otherwise 0.
Index in LRU	If bit 1 above is set, the index of the entry in the LRU. Otherwise 0
Offset	Address of the metadata cache entry in the HDF5 file.
Length	Length of the metadata cache entry image in bytes. Also the length of the space allocated for the entry in the HDF5 file.
Dependency Parent Offsets	If bit 3 above is set, an array containing the offsets of the flush dependency parents in the HDF5 file of length equal to the Dependency Parent Count above. Otherwise no space is allocated to this field.
Entry Image	Serialized image of the metadata cache entry.

Conceptually, the Metadata Cache Adaptive Resize Status Image contains the configuration and current status of the adaptive metadata cache resizing algorithms that attempts to estimate the current size of the metadata working set, and adjust the metadata cache size accordingly. This data is used to reconstruct this configuration and status when the metadata cache image is reloaded on file open.

As this feature is not yet implemented, and as the code in question is fairly involved, this format will almost certainly change as oversights and unnecessary fields become apparent. There may also be changes in general organization.

Notes:

- The Metadata Cache Adaptive Resize Status Image employs a number of doubles, which are not currently used in the file format spec. On discussion with Quincey, I learned that macros for serializing and deserializing doubles in this context have recently been developed. See Quincey for pointers when you get close to this element of the implementation.

Metadata Cache Adaptive Resize Status Image:

byte	byte	byte	byte

Signature			
Version	incr_mode	flash_incr_mode	decr_mode
flags		epoch_mkrs_active	
epoch_length (8 bytes)			
cache_hits (8 bytes)			
cache_accesses (8 bytes)			
min_size^{L}			
max_size^{L}			
max_cache_size^{L}			
min_clean_size^{L}			
index_len			
index_size^{L}			
clean_index_size^{L}			
dirty_index_size^{L}			
lower_hr_threshold (double)			
Increment (double)			
max_increment^{L}			
flash_multiple (double)			
flash_threshold (double)			
flash_size_increase_threshold^{L}			
upper_hr_threshold (double)			
decrement (double)			
max_decrement^{L}			
epochs_before_eviction			

empty_reserve (double)

The following description of the fields in the "Metadata Cache Adaptive Resize Status Image" consists mostly of references to fields in the metadata cache data structures from which the fields are copied and restored. These fields are well documented in the source code, and (in many cases) in the user level documentation as well. While this is certainly good enough for the current version of this document, we need to decide if it is sufficient for the final version.

Field Name:	Description:
Signature	Magic number indicating that this is a metadata cache adaptive resize status image. Must be set to 'ARSI'.
Version	Version of the Metadata Cache Adaptive Resize Status Image. At present, only version 0 is defined.
incr_mode	Value of the incr_mode field of the metadata cache's instance of H5C_auto_size_ctl_t. (cache_ptr->resize_ctl->incr_mode)
flash_incr_mode	Value of the flash_incr_mode field of the metadata cache's instance of H5C_auto_size_ctl_t. (cache_ptr->resize_ctl->flash_incr_mode)
decr_mode	Value of the decr_mode field of the metadata cache's instance of H5C_auto_size_ctl_t. (cache_ptr->resize_ctl->decr_mode)
Flags	Flags indicating the values of boolean fields in H5C_t (the main structure for the metadata cache), and in the instance of H5C_auto_size_ctl_t that appears in H5C_t: bit 0 cache_ptr->size_increase_possible bit 1 cache_ptr->flash_size_increase_possible bit 2 cache_ptr->size_decrease_possible bit 3 cache_ptr->resize_enabled bit 4 cache_ptr->cache_full bit 5 cache_ptr->size_decreased bit 6 cache_ptr->resize_ctl->apply_max_incr bit 7 cache_ptr->resize_ctl->apply_max_decr bit 8 cache_ptr->resize_ctl->apply_empty_reserve
epoch_mkrs_active	Value of the epoch_markers_active field in H5C_t. (cache_ptr->epoch_markers_active)

epoch_length	Value of the epoch_length field of the metadata cache's instance of H5C_auto_size_ctl_t. (cache_ptr->resize_ctl->epoch_length)
cache_hits	Value of the cache_hits field in H5C_t. (cache_ptr->cache_hits)
cache_accesses	Value of the cache_accesses field in H5C_t. (cache_ptr->cache_accesses)
min_size	Value of the min_size field of the metadata cache's instance of H5C_auto_size_ctl_t. (cache_ptr->resize_ctl->min_size)
max_size	Value of the max_size field of the metadata cache's instance of H5C_auto_size_ctl_t. (cache_ptr->resize_ctl->max_size)
max_cache_size	Value of the max_cache_size field in H5C_t. (cache_ptr->max_cache_size)
min_clean_size	Value of the min_clean_size field in H5C_t. (cache_ptr->min_clean_size)
index_len	Value of the index_len field in H5C_t. (cache_ptr->index_len)
index_size	Value of the index_size field in H5C_t. (cache_ptr->index_size)
clean_index_size	Value of the clean_index_size field in H5C_t. (cache_ptr->clean_index_size)
dirty_index_size	Value of the dirty_index_size field in H5C_t. (cache_ptr->dirty_index_size)
lower_hr_threshold	Value of the lower_hr_threshold field of the metadata cache's instance of H5C_auto_size_ctl_t. (cache_ptr->resize_ctl->lower_hr_threshold)
increment	Value of the increment field of the metadata cache's instance of H5C_auto_size_ctl_t. (cache_ptr->resize_ctl->increment)
max_increment	Value of the max_increment field of the metadata cache's instance of H5C_auto_size_ctl_t. (cache_ptr->resize_ctl->max_increment)
flash_multiple	Value of the flash_multiple field of the metadata cache's instance of H5C_auto_size_ctl_t.

	(cache_ptr->resize_ctl->flash_multiple)
flash_threshold	Value of the flash_thresholdfield of the metadata cache's instance of H5C_auto_size_ctl_t. (cache_ptr->resize_ctl->flash_threshold)
flash_size_increase_threshold	Value of the flash_size_increase_threshold field H5C_t. (cache_ptr->flash_size_increase_threshold).
upper_hr_threshold	Value of the upper_hr_threshold field of the metadata cache's instance of H5C_auto_size_ctl_t. (cache_ptr->resize_ctl->upper_hr_threshold).
decrement	Value of the decrement field of the metadata cache's instance of H5C_auto_size_ctl_t. (cache_ptr->resize_ctl->decrement)
max_decrement	Value of the max_decrement field of the metadata cache's instance of H5C_auto_size_ctl_t. (cache_ptr->resize_ctl->max_decrement)
epochs_before_eviction	Value of the epochs_before_eviction field of the metadata cache's instance of H5C_auto_size_ctl_t. (cache_ptr->resize_ctl->epochs_before_eviction)
empty_reserve	Value of the empty_reserve field of the metadata cache's instance of H5C_auto_size_ctl_t. (cache_ptr->resize_ctl->empty_reserve)

4.3 Prefetched Metadata Cache Entries

For current purposes, a prefetched metadata cache entry is simply an entry that appeared in a metadata cache image, that was loaded into the cache, but has not yet been used (i.e. protected) by the library, and which therefore contains only the on disk serialized image of the entry. Earlier versions of this document referred to these entries as “serialized metadata cache entries”, however, on implementation it was observed that these entries had to be treated exactly as prefetched entries would be. As there were already tentative plans to support prefetched entries, it seemed appropriate to change the name so as to facilitate reuse of the code with minimal confusion.

The ideal way of implementing prefetched metadata cache entries would be to alter our entry load processing so that every entry would be a prefetched entry when it is first loaded into the cache. While this would simplify the management of prefetched entries greatly, sadly it is not practical as the correct size of the serialized image of an entry may be unknown until after the entry has been partially deserialized.

As it is impractical to make the prefetched entry part of the normal cycle of entry load, the metadata cache has been modified to support prefetched entries as a new type of entry, that is converted into a normal entry the first time it is protected (or locked, to use the more standard notation).

Fortunately, this has been relatively straight forward, requiring little more than the addition of a boolean flag to H5C_cache_entry_t to indicate whether the entry is prefetched, the creation of a class of prefetched entries with the associated callbacks, and the creation of a routine to handle the details of converting a prefetched entry into a regular one. This routine is a simplified version of the load entry routine, with the following deltas:

- No file I/O
- Destruction of any (reloaded) flush dependency relationships in which the target entry is a child prior to calling the deserialize callback.
- No deserialize retries allowed regardless of entry type (since the size of serialized entry is known).
- Replacement of the prefetched entry with the regular entry returned by the deserialize callback.
- Transfer of any (reloaded) flush dependency relationships in which the target entry is the parent from the prefetched entry to the new regular entry.
- Discard of the old prefetched entry, with the serialized image of the entry being transferred to the new regular entry.

As discussed in the Cycle of Operation section above, the code to write entries to disk also had to be modified to handle prefetched entries. This has been handled through a combination of modifications to H5C_flush_single_entry(), and the above mentioned creation of a client class for prefetched entries.

Finally, there was the matter of evicting a serialized entry. Again, H5C_flush_single_entry() has been modified to support this, and again, the deltas from regular processing are small – specifically:

- Destruction of any (reloaded) flush dependency relations in which the target entry is a child prior to eviction. This is handled via the notify callback in the prefetched entry class. Recall that by the time any entry is evicted, it may not be a flush dependency parent.
- Omission of any callbacks to the underlying class of the prefetched entry.

4.3.1 Prefetched Metadata Cache Entries – code overview

H5C_deserialize_prefetched_entry() is the main routine for converting prefetched entries into regular metadata cache entries. It is called from H5C_protect() whenever a prefetched entry is found to be the target of a protect call. When invoked, it proceeds as follows:

1. Destroy all flush dependencies in which the target prefetched entry is a flush dependency child. The client code will reconstruct these relationships if necessary.
2. Destroy all flush dependencies in which the target prefetched entry is a flush dependency parent. Note that in such cases, the children must be prefetched entries as well. Make note of these flush dependencies so that they can be reconstructed with the deserialized version of the prefetched entry.
3. Pass the entry image buffer from the prefetched entry to the deserialize function appropriate to the underlying type of the prefetched entry. This will result in the creation of a new cache entry containing a deserialized version of the prefetched entry. Initialize the cache entry fields of this new entry, marking it dirty if the prefetched entry is dirty or if the deserialize processes dirties it. Transfer the image buffer to the new entry.
4. Remove the prefetched entry from the cache and discard it via a call to H5C__flush_single_entry() with the H5C__FLUSH_INVALIDATE_FLAG and the

H5C__FLUSH_CLEAR_ONLY_FLAG flags set. If the prefetched entry is dirty, also pass in the H5C__DEL_FROM_SLIST_ON_DESTROY_FLAG.

5. Insert the deserialized version of the prefetched entry into the cache.
6. Reconstruct any flush dependencies in which the prefetched entry was parent, but with the deseriailized entry replacing the prefetched entry in that role.

At this point, the prefetched entry has been replaced with a regular entry, and processing proceeds as usual.

Most of the unusual details of flushing and evicting prefetched entries are handled by the callbacks for the prefetched entry class – in particular, the H5C__prefetched_entry_notify() routine takes down any flush dependency relationships in which a prefetched entry is a child just before eviction (recall that an entry about to be evicted cannot have flush dependency children).

Similarly, H5C__prefetched_entry_free_icr() frees the flush dependency parent addresses array if exists, and verifies that the image buffer is not longer attached to the entry before it frees its space.

All other prefetched entry callbacks should never be called.

4.4 Constructing the Metadata Cache Image

The basic outline of the construction of the metadata cache image is given in the Cycle of Operation section above, and modulo some minor deltas, the actual implementation is quite close to this outline.

After consultation with Mark Miller, we went ahead with the optimization of retaining the on disk images of clean entries so that they need not be serialized again on file close. Given recent concerns about HDF5 library footprint, we should consider making this user configurable.

As expected, it proved most convenient to delay construction of the actual image until just before the final shutdown of the metadata cache. This allowed us to avoid making copies of the on disk images of entries, and to minimize changes to the flush routines.

While the necessary data is lost during the cache shutdown process, it is possible to order the entries in the image so as to ensure that flush dependency parents appear prior to flush dependency children. Thus this ordering is computed and stored at file close warning time. This used to be done by noting the flush dependency height of each entry, and then ordering entries in the image in decreasing flush dependency height order. As flush dependency height is no longer maintained in the metadata cache, it has become necessary to compute the flush dependency height of each entry at file close warning time so as to maintain this capability.

It has also been necessary to serialize all entries that will appear in the cache image at file close warning time so that the cache image size can be computed at this point, and inserted into the cache image superblock extension message.

Since flush dependency relationships, flush dependency heights, and locations in the LRU are likely to change during cache flush, it was convenient to define the H5C_image_entry_t structure (see H5Cprivate.h), and store all of this data in an array of same at file close warning time. This array is then used to construct the cache image proper as the metadata cache is being destroyed. As shall be seen, an array of H5C_image_entry_t is also used for temporary storage of the cache image data during image reload.

4.5.1 Constructing the Metadata Cache Image – code overview

Construction of the metadata cache image is performed in two phases.

The first phase is triggered by a call to H5AC_prep_for_file_close() from either H5F_flush() or H5F_dest() just before the first call to H5AC_flush(). H5AC_prep_for_file_close() simply calls H5C_prep_for_file_close(), which proceeds as follows:

- If a cache image exists, and has not yet been loaded, load it now via H5C_load_cache_image()
- If a cache image has been requested:
  1. Create the cache image superblock extension message. Note that this message will have invalid data at this point.
  2. Serialize the metadata cache via H5C_serialize_cache()
  3. Scan the contents of the metadata cache, and determine which entries will be included in the cache image. Mark these entries accordingly, and compute the size of the cache image. Do this via H5C_prep_for_file_close__scan_entries().

     Note that when entries marked as being contained in the cache image are flushed, writes of their images to disk is suppressed, and the buffers containing these images are not freed when the entry is evicted. This is not a memory leak, as pointers to these buffers are included in the array of H5C_image_entry_t discussed below.
  4. Allocate space for the cache image at the end of file. Do this allocation directly from the VFD, so as to avoid changing the contents of the free space managers.

     Note that if the alignment of the file is greater than 1, this may return a fragment that is not included in the cache image block. The current implementation drops this fragment on the floor. Eventually, we should fix this by increasing the size of the cache image block to be a multiple of the file alignment, and modifying the encode / decode algorithm to allow empty space at the end of the cache image block.

     The Image Data Length field in the cache image file format exists to allow this.
  5. Update the cache image superblock extension message to contain the correct base address and size.
  6. Construct an array of H5C_image_entry_t, one for each cache entry that will appear in the cache image. Load this array will all data required for creating the cache image including pointers to the buffers containing the on disk images of the entries. Do this via a call to H5C_prep_for_file_close__setup_image_entries_array().
  7. Sort the image entries array by flush dependency height and then index in the LRU. Do this via H5C_prep_for_file_close__qsort_image_entries_array().

The actual construction and write of the cache image to file is done in H5C_dest() – which is called from H5AC_dest(), which in turn is called by H5F_dest() as the final step in shutting down the metadata cache.

After calling H5C_flush_invalidate_cache() which flushes any dirty entries remaining in the cache, and then evicts all entries, H5C_dest() checks to see if a cache has been requested. If it has, the function proceeds as to:

1. Construct a buffer containing the image of the cache image block via H5C_construct_cache_image_buffer().
2. Free the image entries array via H5C_free_image_entries_array(). Note that the buffers containing the on disk images of the entries in the cache image are freed at this point.

3. Write the cache image buffer to file via a call to H5AC_write_cache_image(). In the serial case, this function simply writes the image to file at the appropriate location. In the parallel case, only process 0 writes the image, while all other processes do nothing.
4. Free the cache image buffer.

4.6 Loading the Metadata Cache Image

As discussed in the “Cycle of Operation” section, the metadata cache image is loaded into the cache on either the first protect call after it is informed of the existence of the image, or just before file close if there is no activity on the file after file open.

While in principal it should be possible to load the cache image as part of the file open process, in practice, a number of data structures are not fully setup at the point at which the file image is discovered. Hence the delayed open was selected to avoid technical risk.

The significant deltas from the “Cycle of Operation” section are:

- The use of an array of instance of H5C_image_entry_t to store the entry data until it can be used to construct prefetched entries, which are then inserted into the cache.
- The omission of the “prefetched entries list” discussed in the “Cycle of Operation” section. As entries in the metadata cache image are sorted so that flush dependency parents always appear before their associated flush dependency children, it was possible to insert prefetched entries into the cache as they are reconstructed.

Note also the discussion of the metadata cache image feature in the parallel case below.

4.6.1 Loading the Metadata Cache Image – Code Overview

The main routine for loading the metadata cache image is H5C_load_cache_image() which (if a cache image exists) is called either the first time a cache entry is protected, or if the file is closed without any operations on metadata, when the file is closed.

When H5C_load_cache_imaage() is called, it proceeds as follows:

1. Delete the cache image superblock extension message unless (for debugging purposes) instructed not to, and then dirty the superblock extension.
2. Allocate a buffer for the cache image block, and load the cache image into it from file via a call to H5C_read_cache_image().
3. Decode the cache image, and store its contents into an array of H5C_image_entry_t. Do this via a call to H5C_decode_cache_image_buffer().
4. Insert the entries from the cache image into the metadata cache, reconstructing flush dependencies as required, and maintaining the original order of entries in the LRU list. This is done via a call to H5C_reconstruct_cache_contents().

    Very simply, H5C_reconstruct_cache_contents() scans through the entries in the array of H5C_image_entry_t, and performs the following processing on each entry:

    Call H5C_reconstruct_cache_entry() to construct a prefetched entry using the data in the target instance of H5C_image_entry_t.

    a. Insert the prefetched entry in the index.
    b. Update the replacement policy for the insertion.

c. If the cache entry is a flush dependency child, recreate the flush dependency relationships with its parent(s)

The prefetched entries are converted into regular entries when and if they are protected.

5. Free the buffer containing the cache image.
6. If directed, free the file space allocated for the cache image.

4.7 Overall Control of the Metadata Cache Image Feature

As discussed in the Cycle of Operation section, creation of a metadata cache image on file close is requested via a FAPL property on file open. Similarly, decoding the metadata cache image is automatic on file open if the version of the library used understands metadata cache images, and must prevent file open if the library doesn't understand them.

Much of this control uses existing facilities, albeit with extensions as follows:

- Definition of the new FAPL property.
- Code to create and manage the Metadata Cache Image super block extension message.
- Code to manage the high level details of the creation of the metadata cache image. This was implemented through a “prepare for file close” call to the metadata cache that is issued shortly before the first metadata cache flush in the file close process.
- Additions to the cache creation routine that checks the FAPL for a metadata cache image request, and makes note of it if it exists.
- Modifications to the superblock load code to detect the presence of a metadata cache image superblock extension message, and to pass the contents of the message onto the metadata cache if such a message exists.
- Modifications to the metadata cache to read the metadata cache image block prior to the first protect, or on close warning if no protect call occurs first.

4.8 Metadata Cache Image in the Parallel Case

The parallel case is complicated by the fact that while each cache in each process must contain the same dirty entries, there is so such requirement on clean entries. Further, the entries need not appear same order on the different LRUs.

As a result, there is no requirement that the different processes will construct the same cache images, or even cache images of the same size.

As long as all the cache images contain the same dirty entries (as they must until such time as the age out feature is implemented), this is not a problem as only process zero will write a cache image. Thus we need only broadcast the length of the process 0 cache image to all processes, so that this value will be used by all processes when allocating file space for the cache image block, and writing the cache image superblock extension message.

In the current implementation, all process construct a cache image, but only the process zero version is written to file.

On file open, the cache image is read by process 0 only, and then broadcast to all other processes.

NOTE: The collective metadata write code was recently merged into the cache image branch. Unfortunately, this code is currently incompatible with cache image, as it has no facility for suppressing writes of cache entries that are to be included in the cache image. This shouldn’t be a major problem, but pending resolution of this issue, the metadata cache image feature is disabled in the parallel case.

4.8.1 Metadata Cache Image in the Parallel Case – Code Overview

As discussed above, preparation for construction of the cache image is performed on receipt of the file close warning. The only change is the broadcast of the process 0 cache image size, which is used in allocation of file space for the cache image block, and in the metadata cache image superblock extension message. Note that at present, each process constructs its own cache image, event though all but the process 0 version is discarded.

At present, there is no attempt to force all processes to serialize dirty entries in the same order on different processes. As long as file space for metadata is allocated at creation time, and does not move or change size at flush time, this should not be an issue.

The actual reading and writing the cache image is handled by the

H5AC_read_cache_image(), and

H5AC_write_cache_image()

routines respectively. In the serial case, H5AC_read_cache_image() simply reads the image from file and loads it into the supplied buffer. H5AC_write_cache_image() does the converse. Note that these routines handle only actual I/O – encoding and decoding the cache image is handled elsewhere.

In the parallel case, both these routines perform as per the serial case if there is only one process.

With multiple processes, H5AC_read_cache_image() tests to see if it is process 0.

If it is, it reads the cache image block, broadcasts it to all other processes, and then returns the image to the caller.

If it isn't, it simply waits for the broadcast, and then return the broadcast image to the caller.

Similarly, with multiple processes, H5AC_write_cache_image() simply suppresses the cache image write to file for all processes other than process 0.

4.8.2 Metadata Cache Image in the Parallel Case – Known Issues

At present, the code to read cache images in the parallel case uses the same trigger as that in the serial case – typically, the cache image is loaded on the first metadata cache access after file open. Since all processes must participate in the loading the cache image, this creates the potential for a deadlock. For example, suppose process 1 accesses the HDF5 file, and sends a message to process 2, which does not access the file until after this message is received.

While this potential deadlock is easily avoided, the issue should be resolved.

5. Metadata Cache Image Removal Tool

The purpose of the metadata cache image removal tool is to open a HDF5 file with a metadata cache image, read that image into the metadata cache, discard the image, flush all dirty entries in the cache into the file proper, and then close the file.

From a code perspective, this is trivial, as all that is needed is to open the target file R/W and without the metadata cache image FAPL entry, and then close it.

Given the simplicity of the operation, we decided to augment an existing tool to perform this function, rather than write a new tool, with the associated overhead. As h5clear has as somewhat similar purpose (clearing file consistency flags for files created under SWMR and not closed properly), we elected to augment this tool to remove metadata cache images as well.

To this end, we added support for the "–m" and "–image" flags to h5clear. When either of theses flags are set, h5clear will open the supplied HDF5 file R/W, check to see if it contains a cache image, and then close it. If the file does not contain a cache image, h5clear will generate a warning message to that effect.

6. Testing

The metadata cache is central to the functioning of the HDF5 file, and thus any bugs in the metadata cache image facility will likely make themselves apparent quickly upon use of the facility.

The basic issue to be tested is whether the new feature saves and restores the contents and configuration of the metadata cache accurately. This can be broken down into the following check list:

1. Does control work correctly – specifically:
    - A. Is the new FAPL property recognized on file open, and does it result in a notation that a metadata cache image should be created on file close?
    - B. Is the metadata cache notified on file open that a metadata cache image will be created on file close? (may not be needed)
    - C. Is the call to generate a metadata cache image issued on file close?
    - D. Do versions of the library that don't understand metadata cache images refuse to open files that contain one?
    - E. Does the version of the library that does understand metadata cache images recognize the presence of same? Does it issue the necessary call to trigger load of the image into the metadata cache?
2. Is the metadata cache image created correctly?
    - A. Are individual entries correctly serialized?
    - B. Are all entries in the cache serialize with the appropriate annotations (flush dependencies, dirty, LRU index, etc)?
    - C. Is the adaptive cache resizing configuration and status recorded correctly?
    - D. Is the calculation of image size correct?
    - E. Does the image have the expected structure?
3. Is the image written to file correctly?
4. Is the image read from file correctly?
    - A. Is the image interpreted correctly?
        - I. Are individual entries correctly read and represented as serialized entries?
        - II. Is the adaptive cache resizing configuration and status restored correctly?
        - III. Are flush dependencies restored correctly?
5. Are prefetched entries handled correctly?
    - A. On protect?

   B. On flush?
   C. On eviction?
6. Are reloaded flush dependencies on prefetched entries managed correctly?
   A. On protect?
   B. On flush?
   C. On eviction?
7. Is parallel handled correctly?
8. Does SWMR work correctly with metadata cache images?

Aside from addressing the above questions, the test code should fit into the existing regression test framework, and should piggyback on existing test code to the extent reasonably practical.

6.1 Testing – Current Status

With the exception of 1D, section 1 is reasonably well tested with control flow tests and API error tests. 1D (rejection of files with cache images by versions of the library that do not support cache image) is completely un-tested at present.

Items 2 – 6 are tested indirectly through a combination of functional tests which put the library through its paces with cache image enabled, and directly through extensive assertions in the code.

Items 7 – 8 are completely untested.

6.2 Testing – Code Overview

At present, formal test code for the cache image feature resides in the cache_image test in test/cache_image.c. This test code can be divided into the following components:

1. Control flow tests: These test verify that the API calls controlling the creation of cache images perform as expected.
2. Smoke checks: In these tests, we create increasingly complex metadata cache contents through sequences of file opens and closes with cache images passed from one opening of the file to the next. While these tests do not test correct operation of the cache image explicitly, as they involve most on disk data structures used by HDF5, and as the operations on these data structures would likely fail if the cache image corrupted them, these tests are strongly indicative of the correctness of the cache image implementation.
3. API Error Tests: These test verify correct behavior of the cache image related API calls when passed invalid data.

In addition to these formal tests, there are also significant sanity checks in the cache image code which are directed at more explicit verification of correct behavior of cache image. As these tests are only compiled and run in debug builds, they should be converted to formal tests in test/cache_image.c.

At present, I believe the most pressing deficits of the cache image test code are:

1. No parallel testing
2. No verification that older versions of the library will refuse to read files with cache images.

7. Closing Comments and Observations

A number of comments and observations have come up in discussion of this work that should be recorded.

1. The notion has been raised of avoiding metadata cache image related writes to the superblock in the case in which a file with a metadata cache image is opened, and a metadata cache image is requested on file close.

   This is certainly possible, but it would require setting an overlarge cache image so that it would usually not be necessary to move or resize it.

   More to the point, at least in the use case under immediate consideration, there will be writes to the super block regardless. Thus I don't see much room for gain here.

2. Given that the immediate use case is a write only one, implementation of the alternate "Strict LRU" replacement policy in the metadata cache may be of value.
3. Implementation of prefetched entries facilitates broadcasts of metadata cache entries, thus allowing us to avoid the scenario in which each process reads the same piece of metadata from file simultaneously in collective operations.

5. Acknowledgements

TBD

6. Revision History

June 15, 2015:	First draft sent to Quincey for comment.
June 18, 2015:	Second draft sent to Quincey for comment.
June 23, 2015:	Minor cleanups, draft sent to Mark for comment.
Sept. 29, 2015:	Updated document to reflect design changes during implementation (which were minimal), and current state of implementation.
Oct. 19, 2016	Updated document to reflect design changes forced by free space manager and flush dependency design changes. Also expanded design documentation.
Nov. 8, 2016	Add save_resize_status and entry_timeout to, and delete max_image_size fields from H5AC_cache_image_config_t. Add Flags and Image Data Length fields to the top level cache image file format. Added the Age field to the metadata cache image file format. Further updates to reflect actual state of code and to expand design documentation. Made note of the disabling of cache image in the parallel case pending updates to the collective metadata write code.
March 1, 2017	Updated text to note that a request for creation of a cache image on file close (via a call to H5Pset_mdc_image_config()) will fail silently if the

	subsequent create or open does not refer to a file that uses a superblock that does not support superblock extension messages.
March 20, 2017	In section 3.1, added note indicating that cache image creation is disabled in parallel, and that requests for a cache image in this context will fail silently.
	Added section 3.2 describing the H5Fget_cache_image_info() call.
March 21, 2017	Updated to section 3.2 to clarify the behavior of the H5Fget_cache_info() in the R/W and R/O cases.
March 23, 2017	Updated section 5 to describe the functionality added to h5clear to allow it to function as the cache image removal tool.
April 7, 2017	Added section 4.8.2 outlining a potential deadlock in the current (quite limited) cache image support in parallel.