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. ost 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, these 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, 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 property has no effect until file close. On file close, processing proceeds as follows:

1. The metadata cache serializes all entries in the cache so as to fix their on disk 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 of the parent.
    - 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. At present, the file free space managers are also excluded, although with modifications to improve their behavior on file close, they could and should be included in the metadata cache image. All other metadata is 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.
4. The metadata cache creates a superblock 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 2 above.
6. The metadata cache updates the superblock extension message created in 3 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.). Superblock 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. At present, file free space managers are also omitted from the image, and are thus written to file as per the superblock and its associated metadata.
8. After the flush, 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 somewhat simplified. In particular, due to the peculiar behavior of some cache clients, it may be necessary to repeat steps 1 through 6 several times.

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

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

If the library does understand the metadata cache image superblock 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.

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 different from metadata cache entries in the existing cache, in that they contain only the ondisk image of the entry, not the incore 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 parent (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. 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. 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.

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 deserialize 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. Arguably, 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. That said, we have included such a in the metadata cache image control API – although that facility is not yet implemented.

In the parallel case, the cache image is created by process 0, and contains the contents and 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 store and restore of metadata cache adaptive resize status. This has 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

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;
    size_t                  max_image_size;
} 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 max_image_size field is ignored at present.

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.

As discussed in the "Cycle of Operation" section above, the cache image is read automatically if present.

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 as of this writing, some details not fully developed.

4.1 Metadata Cache Image Superlock Extension Message

The metadata cache image superblock 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 superblock 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$_L$			

Field Name:	Description:
Version	Version of the Metadata Cache Image Message. At present, only version 0 is defined.
Offset	Address in the file of the metadata cache image.
Length	Length in bytes of the metadata cache image.

4.1 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, 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 toplevel format follows:

Metadata Cache Image:

byte	byte	byte	byte
Signature			
Version	No space allocated – table alignment only		
num_entries			

Entry image 0
.
.
.
Entry image n
Resize status
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.
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. 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 part is the serialized entry image, and that the length of this image is stored in the Length field.

Metadata Cache Entry Image:

byte	byte	byte	byte
Signature			
Type	Flags	Ring	No space allocated
Dependency Child Count		No space allocated	
Index in LRU			

Dependency Parent Offset$_O$
Offset$_O$
Length$_L$
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.
Ring	Integer indicating the flush ordering ring to which this entry is assigned.
Dependency child count	If bit 2 above is set, the number of flush dependency children of the entry. Otherwise 0.
Index in LRU	If bit 1 above is set, the index of the entry in the LRU. Otherwise 0
Dependency Parent Offset	If bit 3 above is set, the address of the flush dependency parent in the HDF5 file. 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.
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 attempt 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 over sites and unnecessary fields become apparent. There may also be changes in general organization.

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)

4.2 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 version 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 straightforward, 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 current load entry routine, with the following deltas:

- No file I/O
- Destruction of any (reloaded) flush dependency relationship 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 well know).
- 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.
- Omission of any callbacks to the underlying class of the prefetched entry.
-

4.3 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, I 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.

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 me to avoid making copies of the on disk images of entries, and to minimize changes to the flush routines. I was able to order the entries in the image so as to ensure that flush dependency parents appeared prior to flush dependency children much earlier at file close warning time.

Also as expected, it proved necessary to serialize all entries in the cache prior to flushing any entries on file close, and prior to any computation of the size of the metadata cache image. Note that in some cases it is necessary to serialize the cache repeatedly to allow for odd behavior of some of the cache clients.

Note: the H5C_image_entry_t was created to facilitate the construction of the cache image, and is also used in reloading the image. There is some question as to whether this type should be retained. If it is, this section should be updated to discuss this type and its use. More detailed discussion of the metadata cache image construction process is delayed pending this decision.

4.4 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 princip 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.

4.5 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 of a 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 superblock 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 if no protect call occurs first.

4.6 Metadata Cache Image in the Parallel Case

As discussed in the Cycle of Operation section above, we proposed to handle the parallel case with the following deltas from the serial case:

1. On file close, the metadata cache image will be created by the process 0 metadata cache. All other caches will be informed that their dirty entries are now clean as per the current option of metadata writes from process 0 only, and thus be able to discard their contents on close.
2. On file open, process 0 will read the metadata cache image and broadcast it to all processes, where it will be decoded and used to populate each cache in the computation as per the serial case.

As of this writing, the modifications required for the parallel case are partially implemented at present, and completely untested.

TODO: *Update this section with developer level details after full implementation.*

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 will be trivial, as all that will be needed is to open the target file R/W and without the metadata cache image FAPL entry, and then close it.

TODO: Update this section after implementation.

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.

As of this writing, regression test code for the metadata cache image facility consists of a sequence of smoke checks, reasonably rigorous control flow tests, and tests to verify correct handling of a cache image request on a file that is opened read only.

While these tests have likely exposed the vast majority of errors in the code, more focused testing is advisable. The exact extent of this testing remains to be determined, so for now this section contents itself with a check list of points to be examined and verified.

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 as follows:

- Does control work correctly – specifically:
    - 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?
    - Is the metadata cache notified on file open that a metadata cache image will be created on file close? (may not be needed)
    - Is the call to generate a metadata cache image issued on file close?
    - Do versions of the library that don't understand metadata cache images refuse to open files that contain one?
    - 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?
- Is the metadata cache image created correctly?
    - Are individual entries correctly serialized?
    - Are all entries in the cache serialize with the appropriate annotations (flush dependencies, dirty, LRU index, etc)?
    - Is the adaptive cache resizing configuration and status recorded correctly?
    - Is the calculation of image size correct?
    - Does the image have the expected structure?
- Is the image written to file correctly?
- Is the image read from file correctly?
  - Is the image interpreted correctly?
    - Are individual entries correctly read and represented as serialized entries?
    - Is the adaptive cache resizing configuration and status restored correctly?
    - Are flush dependencies restored correctly?
- Are serialized entries handled correctly?
    - On protect?
    - On flush?

        - On eviction?
- Are reloaded flush dependencies on serialized entries managed correctly?
        - On protect?
        - On flush?
        - On eviction?
    - Is parallel handled correctly?
    - 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.

TODO: *Update this section with developer level details as implementation continues.*

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 superblock regardless. Thus I don't see much room for gain here.

2. Given that the immediate used 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 serialized (AKA prefetched) entries facilitates broadcasts of metadata entries, thus allowing us to avoid the scenario in which each process reads the same piece of metadata from file simultaneously in collective operations.

Acknowledgements

TBD

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.