A Developers Overview of the HDF5 Metadata Cache

John Mainzer

Last Modified on January 16, 2015

Table of Contents

1. Introduction

This paper attempts to provide the developer with a conceptual overview of the HDF5 metadata cache and the early design decisions that shaped it. It does not attempt to introduce the basic concepts of caching. If the principle of locality, the notions of working set size and hit rate, or the LRU (Least Recently Used) replacement policy are unfamiliar, the reader is advised to review the subject of caching before proceeding. Most any introductory Operating Systems Design text should cover this material in adequate depth.

It may also be useful to read the description of the UNIX buffer cache found in chapter three of M. J. Bach's "The Design of the UNIX Operating System", as the basic structure of the current version of the metadata cache is loosely based on this design. However, the reader with a good working knowledge of caching and data structures will likely find this unnecessary.

1.1. Outline of the remainder of this Paper

Section 2 of this paper is a review of the history of the metadata cache in HDF5. This is included as echos of the first metadata cache remain in the current cache code, and may be incomprehensible without a historical background.

Section 3 introduces the general structure and layout of the cache. Note that everything in this section is equally applicable to both the version 2 and version 3 metadata caches.

Section 4 discusses metadata I/O, and is only applicable to the version 3 metadata cache. Prior versions delegated metadata I/O to the cache clients.

Section 5 discusses the implementation of adaptive cache resizing. This feature was introduced early in the version 2 metadata cache, and remains essentially unchanged in the version 3 cache.

Section 6 discusses the implementation of flush dependencies. This feature was originally implemented to support SWMR (Single Writer Multiple Readers), but has also be used in the conversion of the fractal heap to the version 3 metadata cache client API.

Section 7 introduces the extensions to the metadata cache required to support PHDF5. This code was introduced early in the development of the version 2 metadata cache, and remains largely unchanged.

Section 8 discusses configuring the metadata cache from the client program at run time. The facilities discussed in this section were implemented early in the development of the version 2 metadata cache, and remain largely unchanged.

Section 9 discusses metadata cache debugging facilities.

Section 10 contains a checklist for writing a metadata cache client, along with a discussion of some of the design issues that may arise.

Finally, the declarations of the major structures used in the metadata cache with their header comments are included in several appendicies at the end of this paper. Please note the length and detail of these header comments. They are a major resource for the developer, which should be read and maintained carefully.

2. History of the HDF5 Metadata Cache

2.1. The Version 1 Metadata Cache

The version 1 metadata cache was implemented before I joined the HDF5 project, so my guesses as to the motivations behind the design decisions incorporated in it are just that. Sadly, while I once met the gentleman who wrote it, he was unwilling to discuss his design decisions with me. Thus I fear my guesses may be the best we can do.

The version 1 metadata cache contained three major departures from typical cache design practice – specifically:

1. Use of a hash table with no provision for collisions for indexing the cache, which thus baked the “Evict on Collision” replacement policy into the design.
2. Disk I/O delegated to cache clients.
3. No fixed cache entry size.

As I was not there at the time, I can only speculate, but at a guess these design decisions were made for ease of implementation (item 1), and to facilitate retrofitting the metadata cache onto the HDF5 library (items 2 & 3). Once can view the version 2 and version 3 metadata caches as reversals of the first two of these decisions. The third remains to this day. While it has required some epicycles discussed later in this paper, it has not proved particularly troublesome. In any case, it is so deeply embedded in the HDF5 library that it would be almost impossible to reverse.

2.1.1. The Version 1 Metadata Cache Indexing Method and Replacement Policy

The version 1 metadata cache indexed the cache with a hash table with no provision for collisions. Instead, collisions were handled by evicting the existing entry to make room for the new entry. Aside from flushes, there was no other mechanism for evicting entries, so the replacement policy could best be described as "Evict on Collision". Note that this design is generally similar to the cache architecture used in early hardware caches for CPUs. Hence it seems possible that these designs were used as a starting point for the version 1 metadata cache.

As a result of this decision, if two frequently used entries hashed to the same location, they would evict each other regularly. To decrease the likelihood of this situation, the default hash table size was set fairly large -- slightly more than 10,000. This worked well, but since the size of metadata entries is not bounded, and since entries were only evicted on collision, the large hash table size allowed the cache size to explode when working with HDF5 files with complex structure. This explosion resulted in reports of machines locking up when running HDF5 – repairing this was the primary impetus behind the development of the version 2 metadata cache.

The "Evict on Collision" replacement policy also caused problems with the parallel version of the HDF5 library, as a collision with a dirty entry could force a write in response to a metadata read. Since all metadata writes must be collective in the parallel case while reads need not be, this could cause the library to hang if only some of the processes participated in a metadata read that forced a write. Prior to the implementation of the version 2 metadata cache, this was dealt with by maintaining a shadow cache for dirty entries evicted by a read. Writes of these entries were then piggybacked on the next collective write

2.1.2. The Version 1 Metadata Cache I/O

The second unusual feature of the version 1 metadata cache was delegating all I/O to the cache clients. While I suspect that this decision greatly simplified the initial integration of the metadata cache in to the HDF5 library, it meant that the cache didn't have a complete picture of the metadata under its control, and also allowed the cache clients to hide the exact on disk structure of entries in the cache from the cache. As long as the metadata cache was only required to support caching proper, this design decision was unusual but workable. However, for both metadata journaling and SWMR, it is necessary for the metadata cache to have complete control over I/O. Reversing this design decision to permit metadata journaling and SWMR was the primary impetus behind the development of the version 3 metadata cache.

2.1.3. The Version 1 Metadata Cache Code

For those interested in spelunking in old versions of the HDF5 library, note that the version 1 metadata cache was located in the H5AC* files. Subsequent versions of the metadata cache have been located largely in the H5C* files, although the H5AC* files still exists, and all calls to the metadata cache are still made to functions with the H5AC prefix.

2.2. The Version 2 Metadata Cache

The version 2 metadata cache was designed to address the above cache size explosion issue discussed in 2.1.1 above, and was first introduced in the 1.6.4 release. After implementation, it became evident that the working set size for HDF5 files varies widely depending on both structure and access pattern. Thus it was necessary to add support for cache size adjustment under either automatic or user program control (discussed in section 5 below). This code was introduced in the 1.8 release. Note that the version 2 metadata cache retained the version 1 cache design feature of delegating all file I/O to the cache clients.

Later in its development, the version 2 metadata cache was further modified to allow the user to disable evictions. This was useful in machines with large amounts of memory when it was necessary to maximize raw data throughput. Needless to say, use of this option forces the cache RAM footprint to grow without bound until evictions are enabled again.

Structurally, the version 2 (and version 3) metadata cache can be thought of as a heavily modified version of the UNIX buffer cache as described in chapter three of M. J. Bach's "The Design of the UNIX Operating System" In essence, the UNIX buffer cache uses a hash table with chaining to index a pool of fixed size buffers. It uses the LRU (Least Recently Used) replacement policy to select candidates for eviction. See any introductory Operating Systems text for a discussion of caching and the LRU replacement policy. In passing, note that the HDF5 metadata cache uses the non-standard terminology protect / unprotect and pin / unpin for two slightly different flavors of the more typical lock / unlock operations used to hold an entry in cache so that it can be read or modified, and to release the entry back into the cache.

The protect / unprotect operations are largely identical to the lock / unlock operation discussed in any tutorial on caching basics, albeit with some interesting twists. The pin / unpin operation was introduced to avoid the overhead of the protect / unprotect operation on frequently accessed entries. Once an entry is pinned, it may not be evicted or flushed except as part of a user initiated flush. In the version 2 cache, a pinned entry may be read or written at any time. This will have to change in the version 3 metadata cache, as journaling requires that the metadata cache be informed whenever an entry is modified.

Since HDF5 metadata entries are not of fixed size, and may grow arbitrarily large, the size of the version 2 (and version 3) metadata cache cannot be controlled by setting a maximum number of entries. Instead the cache keeps a running sum of the sizes of all entries, and attempts to evict entries as necessary to stay within a user specified maximum size. Note the use of the word "attempts" here -- as will be seen, it is possible for the cache to exceed its currently specified maximum size. At present, the LRU replacement policy is the only option for selecting candidates for eviction.

Per the standard unix buffer cache, dirty entries are given two passes through the LRU list before being evicted. The first time they reach the end of the LRU list, they are flushed, marked as clean, and moved to the head of the LRU list. When a clean entry reaches the end of the LRU list, it is simply evicted if space is needed.

The cache cannot evict entries that are protected or pinned, and thus it will temporarily grow beyond its maximum size if there are insufficient unprotected and unpinned entries available for eviction.

In the parallel version of the library, all metadata writes are done done either by or at the direction of the cache running under process 0 of the file communicator. All the other caches must retain dirty metadata cache entries until the process 0 cache tells them that the metadata is clean.

Since all operations modifying metadata must be collective, all caches see the same stream of dirty metadata. This fact is used to allow them to synchronize every n bytes of dirty metadata, where n is a user configurable value that defaults to 256 KB.

To avoid sending the other caches messages from the future, process 0 must not write any dirty entries until it reaches a synchronization point. When it reaches a synchronization point, it writes entries or directs other caches to write entries as needed, and then broadcasts the list of flushed entries to the other caches. The caches on the other processes use this list to mark entries clean before they leave the synchronization point, allowing them to evict those entries as needed.

The caches will also synchronize on a user initiated flush.

To minimize overhead when running in parallel, the cache maintains "clean" and “dirty” LRU lists in addition to the regular LRU list. The “clean” list contains only clean entries, and is used as a source of candidates for eviction when flushing dirty entries is not allowed.

Since flushing entries is forbidden most of the time when running in parallel, the caches can be forced to exceed their maximum sizes if they run out of clean entries to evict.

To decrease the likelihood of this event, the version 2 (and version 3) cache allows the user to specify a minimum clean size -- which is a minimum total size of all the entries on the clean LRU plus all unused space in the cache. At each sync point, the cache attempts to flush sufficient dirty metadata to meet the minimum clean size on sync point exit.

While the clean and dirty LRU lists are only maintained in the parallel version of the HDF5 library, the notion of a minimum clean size still applies in the serial case. Here it is used to force a mix of clean and dirty entries in the cache even in the write only case.

This in turn reduces the number of redundant flushes by avoiding the case in which the cache fills with dirty metadata and all entries must be flushed before a clean entry can be evicted to make room for a new entry.

Observe that in both the serial and parallel cases, the maintenance of a minimum clean size modifies the replacement policy, as dirty entries may be flushed earlier than would otherwise be the case so as to maintain the desired amount of clean and/or empty space in the cache.

While the version 2 (and version 3) metadata cache only supports the LRU replacement policy at present, that may change. Support for multiple replacement policies was very much in mind when the cache was designed, as was the ability to switch replacement policies at run time. The situation has been complicated by the later addition of the adaptive cache resizing requirement, as two of the resizing algorithms piggyback on the LRU list. However, if there is need for additional replacement policies, it shouldn't be too hard to implement them.

Finally, late in the development of the version 2 cache, support for flush dependencies was added. The objective of this modification was to allow the cache to enforce the flush ordering relationships required for the lock free on disk data structure modifications – which are in turn required to implement the SWMR (Single Writer, Multiple Reader) feature requested by some users of the HDF5 library. Observe that the use of flush dependencies also modifies the LRU replacement policy, and will likely modify any other replacement policy that may be implemented.

2.3. The Version 3 Metadata Cache

While the version 3 metadata cache does not contain metadata journaling support at present, support for metadata journaling was the reason it was developed. The primary change required to implement journaling was moving metadata I/O into the metadata cache so that the cache could see the actual on disk images of the various on disk data structures used by HDF5, and thus enable it to construct journal entries. This in turn required a complete re-design and re-implementation of the interface between the metadata cache and the cache clients.

The cost of converting the cache clients to use the new interface was badly underestimated in the initial cost estimates (~ 50 hours estimated, ~1000+ hours actual), and thus the completed version 3 cache with journaling support spent several years sitting unused on a development branch. It was finally brought into the trunk without journaling support as the same direct control over metadata I/O is required for SWMR. At this time, it is unclear if and when journaling support will be moved into the trunk. Note that it will be necessary to either verify that pinned entries are not modified by cache clients or to detect all such changes before journaling can be supported.

The remainder of this paper refers to the version 3 metadata cache, although with the exception of the section on metadata I/O, all of it is directly applicable to the version 2 cache as well.

3. Basic Structure of the HDF5 Metadata Cache

This section contains an overview of the low level functions of the metadata cache, and the associated data structures. The primary emphasis is on the peculiarities of the HDF5 implementation and the location location of the supporting code and data structures. A knowledge of basic cache design and the associated data structures is assumed.

3.1. Indexing

As discussed earlier, the metadata cache uses a hash table with chaining to index entries by base address in the metadata cache. The most unusual features of this hash table are its size (which must be a power of 2) and its hash function, which is much simpler that is typical – specifically

```
(int)(((x) & H5C__HASH_MASK) >> 3),
```

where

```
H5C__HASH_MASK = ((size_t)(H5C__HASH_TABLE_LEN - 1) << 3).
```

This works because of the lack of any fixed size for HDF5 metadata. This in turn seems to add sufficient noise to the to placement of metadata in the HDF5 file to allow the hash table to function well with its very simplistic hash function.

The hash table also maintains a running total of the size of all entries in the index, as well as the total sizes of all clean and dirty entries.

The chaining is implemented with a DLL (Doubly Linked List). If a hash bucket contains more than one cache entry, entries not at the head of the DLL are moved to the head of the DLL on lookup.

The code managing the index and its associated DLLs is in a collection of macros located in H5Cpkg.h. While this code was originally in functions, it was moved to macros as profiling indicated that the function call overhead of this and other frequently called functions accounted for roughly half of the metadata cache processing. When compiled with sanity checking enabled, these macros contain extensive sanity checking code that should detect most errors in the use of the index macros.

The declaration of the hash table used by the index, and other fields used in the maintenance of the index may be found in H5C_t, declared in H5Cpkg.h. Each cache entry also contains fields required to support chaining in the hash table. These fields may be found in H5C_cache_entry_t, which is declared in H5Cprivate.h. See section 3.6 below for further discussion of these structures.

3.2. Replacement Policy

At present, the only replacement policy supported by the metadata cache is LRU – but LRU modified by support for a minimum clean size, and by flush dependencies.

To facilitate implementation of other replacement policies, all replacement policy operations (i.e. update for insert, protect/unprotect, flush, etc. are implemented in macros defined in H5Cpkg.h. As with the index macros discussed above, these were originally functions. They were converted to macros because of excessive function call overhead for these frequently used operations.

The LRU replacement policy is implemented with a DLL (Doubly Linked List). In the parallel version of the metadata cache, this list is augmented with clean and dirty LRUs to facilitate eviction of clean entries when metadata writes are forbidden. In all cases, these lists are augmented with running tallies of the number of entries resident and the sum of their sizes.

The cache also maintains lists of currently protected entries (the protected list) and currently pinned entries (the pinned entry list). Entries are removed from the LRU list(s) on protect or pin and inserted in the protected list or the pinned entry list as appropriate to prevent them from participating in the LRU algorithm while protected and/or pinned. Entries that are unpinned and unprotected are inserted at the head of the LRU list(s) as per the usual LRU algorithm.

The LRU algorithm used in the metadata cache differs from the basic algorithm in the following particulars:

1. As per the UNIX buffer cache, dirty entries are given two passes through the LRU. When they reach the bottom of the list, they are flushed and then moved to the head of the list. Clean entries that reach the bottom of the list are simply evicted as necessary to free up space for newly inserted or loaded entries.
2. Support for a minimum reserve of clean entries in both the serial and parallel versions of the metadata cache distorts the LRU algorithm by flushing dirty entries sooner than would otherwise be the case. This probably results in earlier eviction of cold dirty entries, and extra flushes of cool dirty entries. It likely has no effect on hot entries.
3. Support for flush dependencies almost certainly distorts the LRU algorithm, although the exact nature of this distortion is hard to predict. A study of this would be useful.

Moving entries to the head of the LRU whenever they are unprotected is equivalent to the usual practice of moving them to the head of the list whenever they are accessed.

The pinning facility is slightly different, as there is no guarantee that a pinned entry will be accessed frequently while it is pinned. That said, in practice, it usually is. Thus placing unpinned entries at the head of the LRU likely causes little if any distortion of the algorithm.

Note that it is possible to protect and unprotect a pinned entry. In this case, the entry is moved from the pinned entry list and back again. While entries may be unpinned at any time, they may only be pinned while protected, or as part of the unprotect operation.

The declaration of the fields used by the replacement policy may be found in H5C_t, declared in H5Cpkg.h. Each cache entry also contains fields required to support the LRU and the clean and dirty LRUs. These fields may be found in H5C_cache_entry_t, which is declared in H5Cprivate.h. See section 3.6 below for further discussion of these structures.

Note that the LRU list is employed by the default adaptive cache resizing algorithm (discussed in Section 5 below). This will complicate implementation of alternate replacement policies should this ever be required.

3.3. Inserting, Loading, Flushing, and Evicting Entries

Entries may enter the metadata cache either by being inserted by a cache client, or as the result of a protect operation on an entry that is not currently in the cache. In either case, if there is not sufficient space in the cache for the new entry, the cache will attempt to make space by evicting entries until there is sufficient space. Selection of entries to be evicted is the province of the replacement policy – only the modified LRU policy discussed above for now.

Entries are inserted into the cache via the H5C_insert_entry() call in H5C.c. Entries inserted into the cache in this manner are dirty by definition. Note that it is the cache clients responsibility to allocate file space for the new entry. H5C_insert_entry() accepts flags indicating that:

1. The entry should be inserted pinned.
2. The entry should have its flush marker flag set (see discussion of flushing the cache below).
3. The entry should be flushed last (used for superblock and related on disk data structure).

4. The entry should be flushed collectively (parallel case only).

Entry loads are triggered by the protect call (discussed below). The actual load is performed by the H5C_load_entry() function in H5C.c. In addition to the I/O discussed in section 4 below, the function initializes the instance of H5C_cache_entry_t which must be the first field of the in core representation of all cache entries. Note that it does not either insert the entry into the cache data structures, or arrange for sufficient space for the entry – this is done by the caller.

Flushing and evicting entries is handled by H5C_flush_single_entry() in H5C.c. In addition to the I/O discussed in Section 4 below, it updates the cache index, slist (discussed in section 3.5 below), and the replacement policy for the flush and/or eviction.

FLAGS	Operations:
None	Flush entry to disk (must be dirty). Mark clean. Remove from slist. Update index for entry clean. Update replacement policy for flush.
H5C__FLUSH_CLEAR_ONLY_FLAG	Mark clean (entry must be dirty). Call clear callback if defined. Remove from slist. Update index for entry clean. Update replacement policy for flush.
H5C__FLUSH_INVALIDATE_FLAG	Flush entry if dirty. Remove from slist if dirty. Remove from index. Update replacement policy for eviction. Discard in core representation of entry.
H5C__FLUSH_CLEAR_ONLY_FLAG H5C__FLUSH_INVALIDATE_FLAG	Remove from slist if dirty. Remove from index. Update replacement policy for eviction. Discard in core representation of entry.
H5C__FLUSH_CLEAR_ONLY_FLAG H5C__FLUSH_INVALIDATE_FLAG H5C__FREE_FILE_SPACE_FLAG	Remove from slist if dirty. Remove from index. Update replacement policy for eviction. Discard on disk file space of entry. Discard in core representation of entry.
H5C__FLUSH_INVALIDATE_FLAG H5C__TAKE_OWNERSHIP_FLAG	Flush entry if dirty. Remove from slist if dirty. Remove from index. Update replacement policy for eviction. Do not discard in core representation of entry – entry is now managed by cache client.

Table 1: Flags accepted by H5C_flush_single_entry()

H5C_flush_single_entry() accepts flags indicating that the target entry should be:

1. Marked clean but not written to disk (H5C__FLUSH_CLEAR_ONLY_FLAG).
2. Removed from the cache (H5C__FLUSH_INVALIDATE_FLAG).
3. Have its on disk space allocation released (H5C__FREE_FILE_SPACE_FLAG).
4. Removed from the cache and turned over to the cache client (H5C__TAKE_OWNERSHIP_FLAG).

The above flags may appear in the combinations indicated in table 1. Note that the list of operations for each flag combination is somewhat simplified – see the code for the full details. In particular, flags specific to test code are omitted, as are some callbacks.

Entries may also be removed from the cache via the H5C_expunge_entry() call – however this is only a wrapper which calls H5C_flush_single_entry() with the H5C__FLUSH_CLEAR_ONLY_FLAG and H5C__FLUSH_INVALIDATE_FLAG flags set. The user may also pass in the H5C__FREE_FILE_SPACE_FLAG, which is forwarded to H5C_flush_single entry().

Finally, as discussed below, entries may be removed from the cache via the unprotect call if the appropriate flags are set.

3.4. Protecting, Unprotecting, Pinning, and Unpinning Entries

The protect (or more conventionally the lock) operation is implemented by the H5C_protect() function in H5C.c. This function loads the indicated entry into the cache if it is not already present, and attempts to make space for it if necessary via a call to H5C_make_space_in_cache() (also in H5C.c). It then removes the target entry from the replacement policy data structures, and transfers it to the protected list so as to lock it into the cache.

However, H5C_protect() also performs several other less standard functions.

1. The function accepts the H5C__READ_ONLY_FLAG. When this flag is set, the target entry is protected read only, and may be protected read only multiple times.
2. The function also accepts the H5C__FLUSH_LAST_FLAG (and, in the parallel version of the cache, the H5C__FLUSH_COLLECTIVELY_FLAG). These flags are only applied to entries that are loaded as a result of the protect operation. As they are only applied to the super block and related data structures, and as these structures are pinned in the cache until the file is closed, this limited functionality is sufficient to the task.
3. H5C_protect() invokes the adaptive cache resizing code at the end of each epoch if this feature is enabled (see Section 5 below for a discussion of adaptive cache resizing).
4. The H5C_protect() call may be invoked on pinned (but currently unprotected) entries. In this case, the entry is moved from the pinned entry list to the protected entry list.

The unprotect (or more conventionally unlock) operation is implemented by the H5C_unprotect() function in H5C.c. The function removes the target entry from the protected list, and returns it to the replacement replacement policy data structures where it can be flushed and evicted as indicated by the replacement policy.

However, like H5C_protect(), it performs several other less standard functions.

1. If the target entry was protected read only, it decrements the read only reference count, and only releases the entry if the count drops to zero.
2. If the target entry was pinned during this or a previous protect, the entry is moved to the pinned entry list instead of being returned to the replacement policy data structures.
3. If the H5C__DELETED_FLAG is set, the entry is passed to the H5C_flush_single_entry() function (discussed above) with both the H5C__FLUSH_CLEAR_ONLY_FLAG and H5C__FLUSH_INVALIDATE_FLAG set. Note that either the H5C__FREE_FILE_SPACE_FLAG or the H5C__TAKE_OWNERSHIP_FLAG (but not both) may appear with the H5C__DELETED_FLAG, and will be passed to H5C__flush_single_entry() if they appear.
4. If the H5C__DIRTIED_FLAG is set, the entry will be marked dirty, and the metadata cache data structures (both regular and replacement policy) are updated accordingly if the entry was not already dirty. Obviously, this flag may not appear if the entry was protected read only.
5. If the H5C__PIN_ENTRY_FLAG is set, the entry will be marked as being pinned and transferred to the pinned entry list instead of being released to the replacement policy data structures. Note that the entry may not be already pinned.
6. If the H5C__UNPIN_ENTRY_FLAG is set, the entry will be marked as not being pinned, and released to the replacement policy data structures. Note that the entry must be pinned to begin with.
7. If the H5C__SET_FLUSH_MARKER_FLAG, the entry's flush marker flag is set (see section 3.5 below for an explanation of what this entails).

Observe that not all flags can be set under all circumstances – for instance the H5C__PIN_ENTRY_FLAG and the H5C__UNPIN_ENTRY_FLAG may not appear together, and the target entry must not be already pinned (or unpinned). Similarly, the H5C__DIRTIED_FLAG may not be applied to an entry that was protected read only. See the code for further details.

As indicated above, the pin and unpin operations are an attempt at avoiding the overhead of repeated protect and unprotect operations on frequently accessed entries.

Currently protected entries may be pinned via the H5C_pin_protected_entry() function in H5C.c. This operation simply sets the is_pinned flag in the entry so that it will be moved to the pinned entry list on unprotect.

Pinned entries may be unpinned via the H5C_unpin_entry() function in H5C.c. If the entry is protected at the time, the entry's is_pinned flag is simply reset, so that it will be released to the replacement policy data structures on unprotect. If the entry is not protected at the time of the call, it is immediately marked as unpinned, and released to the replacement policy data structures where it may be flushed or evicted as directed by the replacement policy.

3.5. Flushing the Cache

The metadata cache is flushed either on user command, or to flush and evict all entries in preparation for file close (and possibly library shutdown).

In either case, to maximize disk performance, the cache attempts to flush all dirty entries in increasing address order. To support this, all dirty entries are inserted in a skip list maintained by the metadata cache. This in turn ensures that a list of all dirty entries in increasing address order is always available.

Conceptually, we scan the slist in increasing address order, flushing each entry as we come to if via calls to H5C_flush_single_entry(). Unfortunately, there are several issues that may prevent such a well ordered flush of all dirty entries:

1. As discussed in Section 4 below, the pre-serialize and serialize callbacks used to obtain on disk images of metadata cache entries may resize and/or relocate the entry being flushed. Further, they may dirty, relocate, resize, or unpin other entries in the cache. In addition to the obvious possible disarrangement of cache entry flush order, such operation may modify the slist out from under the flush code. This in turn requires the flush to re-start from the beginning of the slist.
2. Entries marked as flush last (i.e. the super block and related entries) are not flushed until no other dirty entries remain.
3. Flush dependencies (discussed in section 6) may also alter flush order, as they require that certain entries be flushed before others, regardless of their addresses in cache.
4. If the flush call is made with the H5C__FLUSH_MARKED_ENTRIES_FLAG set, only those entries whose flush_marker boolean fields in H5C_cache_entry_t are set will be flushed. This feature exists to allow the library to flush only a well defined subset of the dirty metadata – for example, all dirty metadata associated with a particular on disk data structure.

The flush_marker boolean is set when the target entry is inserted or unprotected with the H5C__SET_FLUSH_MARKER_FLAG set. The flush_marker fields of an entries is cleared when it is flushed – regardless of the reason.

All metadata cache flush requests are handled via a call to H5C_flush_cache() in H5C.c.

The flush and eviction of all entries prior to file close is requested by passing in the H5C__FLUSH_INVALIDATE_FLAG, which causes the flush to be handled by H5C_flush_invalidate_cache(), also in H5C.c

If the H5C__FLUSH_INVALIDATE_FLAG is not supplied, the flush is handled by H5C_flush_cache() proper. In the typical case, the function simply flushes all dirty entries to disk via calls to H5C_flush_single_entry() with the flags passed into it. However, when the H5C__FLUSH_MARKED_ENTRIES_FLAG is set, H5C_flush_cache() flushes only those dirty entries whose flush_marker boolean fields are set.

3.6. Code Organization and Major Data Structures

All the functionality discussed in Section 3 is implemented in H5C.c, with data structures defined in H5Cpkg.h and H5Cprivate.h.

The version 1 metadata cache code used be in H5AC.c and the related header files. When the version 2 meatadata cache was first implemented, H5AC became little more than a wrapper for the new version of the cache. However, with changes to the code supporting PHDF5 (see Section 7), H5AC.c and its associated header files became home to much of the synchronization code supporting PHDF5.

The main structure used by the metadata cache is H5C_t, defined in H5Cpkg.h. Associated with this definition is an extensive header comment outlining the use of each of the fields in the structure. While the version of this structure and header comment found in H5Cpkg.h should be viewed as the master copy, I have reproduced both in Appendix 1 as a convenience.

Each metadata cache entry must contain an instance of H5C_cache_entry_t as its first element. This instance is used by the cache to store all entry specific data, and to support membership in the various linked lists used to store cache entries. H5C_cache_entry_t is defined in H5Cprivate.h, and is accompanied by an extensive header comment outlining the use of each of the fields in the structure. While the version of this structure and header comment in H5Cprivate.h should be viewed as the master copy, I have reproduced both in Appendix 2 as a convenience.

Please review both of these structures, and note the size and detail of their associated header comments. These comments are a valuable resource for the developer, and should be read and maintained carefully.

4. Metadata I/O

Metadata I/O in the HDF5 metadata cache follows the following general plan.

On entry load, the cache first asks the client for the load size of the entry via a callback. The cache then allocates a buffer of this size, loads it with data from the file staring at the supplied start address, and passes the buffer to the client to be deserialized. The client allocates an in core representation of the entry, loads it with data from the supplied buffer, and passes the in core representation to the cache. The cache then initializes the cache specific fields of the in core representation and inserts it into the cache.

In some cases, the client may decide that it reported an incorrect load size upon examination of the buffer supplied by the metadata cache. To simplify greatly, in this case it informs the cache of the error and reports the correct size. The load procedure is then repeated with the correct load size.

On entry flush, the cache first calls the clients pre-serialize callback if it is supplied. This callback allows the client to resize and/or relocate the entry. Clients may also use the call to place the entry in correct format for writing to disk. For example, this may require relocating entries pointed to by the target entry from temporary to real file space so that the target entry contains real file space addresses of the referenced entries.

Pre-serializeation complete, the cache allocates a buffer large enough for the on disk image of the target entry, and passes it to the client to be loaded with the on disk representation. The client loads the buffer with the on disk representation of the cache entry, and passes the buffer back to the cache. The cache then writes the buffer to file at the base address of the cache entry.

Pointers to the client callback functions are stored in instance of H5C_class_t, which is defined in H5Cprivate.h. This structure is accompanied by a header comment that documents the callbacks to be provided by the client. While the version in H5Cprivate.h should be viewed as the canonical version, the definition of H5C_class_t, the associated pointer types, and the header comment are reproduced in Appendix 3 as a convenience. Please read the associated header comment, and note the detail of the descriptions of the callback functions to be provided by the client. If you modify this structure, or the functionality of the callbacks, be sure to update the header comment.

For details of the use of the callbacks, see the code of H5C_load_entry() and H5C_flush_single_entry().

5. Adaptive Cache Resizing

As mentioned earlier, the metadata working set size for a HDF5 file varies wildly depending on the structure of the file and the access pattern. For example, a 2MB limit on metadata cache size is excessive for an H5repack of almost all HDF5 files I have tested. However, I have a file submitted by one of our users that that will run a 13% hit rate with this cache size, and (in 2004) would lock up one of the HDF Group linux boxes using the version 1 metadata cache. Increase the metadata cache size to 4 MB, and the hit rate exceeds 99%.

In this case the main culprit was a root group with more than 20,000 entries in it. As a result, the root group heap exceeds 1 MB, which tends to crowd out the rest of the metadata in a 2 MB cache

This case and a number of synthetic tests convinced us that we needed to modify the new metadata cache to expand and contract to match working set size within user specified bounds.

I was unable to find any previous work on this problem, so I invented solutions as I went along. If you are aware of prior work, please send me references. The closest I was able to come was a group of embedded CPU designers who were turning off sections of their cache to conserve power.

5.1. Increasing the Cache Size

In the context of the HDF5 library, the problem of increasing the cache size as necessary to contain the current working set turns out to involve two rather different issues.

The first of these, which was recognized immediately, is the problem of recognizing long term changes in working set size, and increasing the cache size accordingly, while not reacting to transients.

The second, which was recognized the hard way, is to adjust the cache size for sudden, dramatic increases in working set size caused by requests for large pieces of metadata which may be larger than the current metadata cache size.

The algorithms for handling these situations are discussed below. These problems are largely orthogonal to each other, so both algorithms may be used simultaneously.

5.1.1. Hit Rate Threshold Cache Size Increment

Perhaps the most obvious heuristic for identifying cases in which the cache is too small involves monitoring the hit rate. If the hit rate is low for a while, and the cache is at its current maximum size, the current maximum cache size is probably too small.

The hit rate threshold algorithm for increasing cache size applies this intuition directly.

Hit rate statistics are collected over a user specified number of cache accesses. This period is known as an epoch.

At the end of each epoch, the hit rate is computed, and the counters are reset. If the hit rate is below a user specified threshold and the cache is at its current maximum size, the maximum size of the cache is increased by a user specified multiple. If required, the new cache maximum size is clipped to stay within the user specified upper bound on the maximum cache size, and optionally, within a user specified maximum increment.

My tests indicate that this algorithm works well in most cases. However, in a synthetic test in which hit rate increased slowly with cache size, and load remained steady for many epochs, I observed a case in which cache size increased until hit rate just exceeded the specified minimum and then stalled. This is a problem, as to avoid volatility, it is necessary to set the minimum hit rate threshold well below the desired hit rate. Thus we may find ourselves with a cache running with a 91% hit rate when we really want it to increase its size until the hit rate is about 99%.

If this case occurs frequently in actual use, we will have to come up with an improved algorithm. Please report this behavior if you see it in the wild. However, I had to work rather hard to create it in my synthetic tests, so I would expect it to be uncommon.

5.1.2. Flash Cache Size Increment

A fundamental problem with the above algorithm is that contains the hidden assumption that cache entries are relatively small in comparison to the cache itself. While I knew this assumption was not generally true when I developed the algorithm, I thought that cases where it failed would be so rare as to not be worth considering, as even if they did occur, the above algorithm would rectify the situation within an epoch or two.

While it is true that such occurrences are rare, and it is true that the hit rate threshold cache size increment algorithm will rectify the situation eventually, the performance degradation experienced by users while waiting for the epoch to end was so extreme that some way of accelerating response to such situations was essential.

To understand the problem, consider the following use case:

Suppose we create a group, and then repeatedly create a new data set in the group, write some data to it and then close it.

In some versions of the HDF5 file format, the names of the datasets will be stored in a local heap associated with the group, and the space for that heap will be allocated in a single, contiguous chunk. When this local heap is full, we allocate a new chunk twice the size of the old, copy the data from the old local heap into the new, and discard the old local heap.

By default, the minimum metadata cache size is set to 2 MB. Thus in this use case, our hit rate will be fine as long as the local heap is no larger than a little less than 2 MB, as the group related metadata is accessed frequently and never evicted, and the data set related metadata is never accessed once the data set is closed, and thus is evicted smoothly to make room for new data sets.

All this changes abruptly when the local heap finally doubles in size to a value above the slightly less than 2 MB limit. All of a sudden, the local heap is the size of the metadata cache, and the cache must constantly swap it in to access it, and then swap it out to make room for other metadata.

The hit rate threshold based algorithm for increasing the cache size will fix this problem eventually, but performance will be very bad until it does, as the metadata cache will largely ineffective until its size is increased.

An obvious heuristic for addressing this "big rock in a small pond" issue is to watch for large "incoming rocks", and increase the size of the "pond" if the rock is so big that it will force most of the "water" out of the "pond".

The add space flash cache size increment algorithm algorithm applies this intuition directly:

Let x be either the size of a newly inserted entry, a newly loaded entry, or the number of bytes by which the size of an existing entry has been increased (i.e. the size of the "rock").

If x is greater than some user specified fraction of the current maximum cache size, increase the current maximum cache size by x times some user specified multiple, less any free space that was in the cache to begin with. Further, to avoid confusing the other cache size increment/decrement code, start a new epoch.

At present, this algorithm pays no attention to any user specified limit on the maximum size of any single cache size increase, but it DOES stay within the user specified upper bound on the maximum cache size.

While it should be easy to see how this algorithm could be fooled into inactivity by large number of entries that were not quite large enough to cross the threshold, in practice it seems to work reasonably well.

Needless to say, the issue will have to be revisited should this cease to be the case.

5.2. Decreasing the Cache Size

Identifying cases in which the maximum cache size is larger than necessary turned out to be more difficult.

5.2.1. Hit Rate Threshold Cache Size Reduction

One obvious heuristic is to monitor the hit rate and guess that we can safely decrease cache size if hit rate exceeds some user supplied threshold (say .99995).

The hit rate threshold size decrement algorithm implemented in the new metadata cache implements this intuition as follows:

At the end of each epoch (this is the same epoch that is used in the cache size increment algorithm), the hit rate is compared with the user specified threshold. If the hit rate exceeds that threshold, the current maximum cache size is decreased by a user specified factor. If required, the size of the reduction is clipped to stay within a user specified lower bound on the maximum cache size, and optionally, within a user specified maximum decrement.

In my synthetic tests, this algorithm works poorly. Even with a very high threshold and a small maximum reduction, it results in cache size oscillations. The size increment code typically increments maximum cache size above the working set size. This results in a high hit rate, which causes the threshold size decrement code to reduce the maximum cache size below the working set size, which causes hit rate to crash causing the cycle to repeat. The resulting average hit rate is poor.

It remains to be seen if this behavior will be seen in the field. The algorithm is available for use, but it wouldn't be my first choice.

5.2.2. Ageout Cache Size Reduction

Another heuristic for dealing with oversized cache conditions is to look for entries that haven't been accessed for a long time, evict them, and reduce the cache size accordingly.

The age out cache size reduction applies this intuition as follows: At the end of each epoch (again the same epoch as used in the cache size increment algorithm), all entries that haven't been accessed for a user configurable number of epochs (1 - 10 at present) are evicted. The maximum cache size is then reduced to equal the sum of the sizes of the remaining entries. The size of the reduction is clipped to stay within a user specified lower bound on maximum cache size, and optionally, within a user specified maximum decrement.

In addition, the user may specify a minimum fraction of the cache which must be empty before the cache size is reduced. Thus if an empty reserve of 0.1 was specified on a 10 MB cache, there would be no cache size reduction unless the eviction of aged out entries resulted in more than 1 MB of empty space. Further, even after the reduction, the cache would be one tenth empty.

In my synthetic tests, the age out algorithm works rather well, although it is somewhat sensitive to the epoch length and age out period selection.

5.2.3. Ageout With Hit Rate Threshold Cache Size Reduction

To address these issues, I combined the hit rate threshold and age out heuristics.

Age out with threshold works just like age out, except that the algorithm is not run unless the hit rate exceeded a user specified threshold in the previous epoch.

In my synthetic tests, age out with threshold seems to work nicely, with no observed oscillation. Thus I have selected it as the default cache size reduction algorithm.

The age out algorithm is implemented by inserting a marker entry at the head of the LRU list at the beginning of each epoch. Entries that haven't been accessed for at least n epochs are simply entries that appear in the LRU list after the n-th marker at the end of an epoch.

5.3. Supporting Data Structures and Code

The functions supporting adaptive cache resizing are located in H5C.c. Look for functions whose names start with "H5C__auto" and "H5C__flash".

The fields supporting adaptive cache resizing are declared in struct H5C_t, located in H5Cpkg.h. In particular, note the array of epoch marker entries, the associated instance of H5C_class_t, and callback function declarations. None of these callbacks should ever be invoked.

Configuration of the the adaptive cache resizing code is discussed in Section 8 below.

6. Flush Dependencies

<<<This section needs work – assigned to Vailin >>>

7. Parallel Operation

PHDF5 requires all operations that can modify metadata to be executed collectively. Thus, the metadata caches on all processes in an MPI computation see the same stream of dirty metadata.   The method used to maintain maintain coherency between metadata caches in such computations rests on this observation.

In contrast, operations that read metadata need not be executed collectively – this implies that even if the caches are coherent, caches on different MPI processes can contain different sets of clean entries, and that while they must contain the same set of dirty entries, those entries may appear in different orders on the local LRU list.

7.1. Cache Coherency

For purposes of metadata caching in HDF5, cache coherency means that if two caches on two different MPI processes have seen the same same sequence of collective operations modifying metadata, then all entries that appear in both caches must be identical. While this definition does not require that both caches contain the same set of dirty entries, in the current implementation that will be the case.

Since all operations modifying metadata are performed collectively, there are only two ways that caches can lose coherency – messages from the future, and messages from the past.

Messages from the future occur when one MPI process performs an operation that modifies metadata and writes that modified metadata to disk before a second process loads some or all of this modified metadata to perform the same operation.

Messages from the past occur when a piece of metadata is modified, written to disk, and evicted by one process, overwritten with an earlier version of the cache entry by a second process, and then reloaded back into cache by the first process.

In the HDF5 metadata cache, we avoid messages from the past and future by synchronizing the caches from time to time, and not allowing metadata writes (and thus evictions of dirty metadata cache entries) outside of sync points.

Since all caches see the same stream of dirty metadata, we use this as our clock. Every n bytes of dirty metadata (n is user configurable, but is set to 256 KB by default), all caches enter a sync point, which starts with a barrier. Once past the barrier, the process 0 metadata cache uses the replacement policy to pick entries to flush to reduce its load of dirty metadata sufficiently to meet the min clean constraint, and, if necessary, to evict sufficient entries to meet its target size.

In the initial version of this algorithm (process 0 metadata write), the process 0 cache would flush the selected entries, broadcast the list of flushed entries to all other metadata caches in the computation, and then exit the sync point. All other processes would receive the list of flushed entries, mark these entries clean in their internal structures, and then leave the sync point. Observe that no exiting barrier is required, since all flushed entries have been written to disk before the list of flushed entries is broadcast.

In the second version of this algorithm (distributed metadata write), process 0 broadcasts the list of entries to be flushed. All processes then run the same algorithm to divide responsibility for flushing the entries between them. Each process flushes its entries, and marks the remainder clean. In this version of the algorithm, a barrier on exit is required.

Sync points may also be triggered by flush calls – which must be collective. Processing here is conceptually the same as that detailed above, save that all dirty entries are flushed. Note that MPI applications that create all data sets early in the computations and then perform no further metadata operations are encouraged to flush the cache after completing the data set creation phase. If this is not done, dirty metadata will be held in the cache until file close, regardless of how frequently or infrequently it is accessed.

Note that there is one hidden assumption in the above algorithm – specifically that entries will neither change size nor location as a result of being flushed. This is handled by the simple expedient of disabling all features that could cause this to happen in parallel builds.

7.2. History and Implementation Details

When I originally developed the code supporting PHDF5 with the version 2 (and subsequently version 3) cache, I planned to allow the process 0 metadata cache to write entries between sync points, and then update the rest of the caches at the sync point.

As initially implemented, this allowed occasional message from the future to reach the other metadata caches. To fix this, I required all metadata writes to take place during the sync point.

After mulling on the matter for six months or so, it occurred to me that if I restricted the process 0 metadata cache to writing only entries that had been dirty in all caches at the end of the last sync point, I could avoid the messages from the future issue I encountered on my first attempt. This optimization has never been implemented. If it ever is, note that allowing process 0 to write entries between sync points requires extra code to maintain the count of dirty bytes as seen by all other caches. This code should still be extant, although I am sure it has suffered bit rot.

The main routine for running sync points is H5AC_run_sync_point() in H5AC.c. The function is called by H5AC_insert_entry(), H5AC_move_entry(), and H5AC_unprotect() whenever the dirty bytes threshold is exceeded. It is also called by H5AC_flush_entries(),

Management of cache coherency in PHDF5 also requires maintenance of a number of variables. These variables are stored in an instance of H5AC_aux_t, and associated with the metadata cache's instance of H5C_t via the aux_ptr field in that structure. While the version of H5AC_aux_t in H5Cpkg.h should be viewed as the canonical one, the declaration of the structure and its associated header comment have been reproduced in Appendix 4 for convenience. As usual, please read and maintain the header comment carefully.

8. Configuring the Metadata Cache

All the metadata cache configuration data for a given file is contained in an instance of the H5AC_cache_config_t structure -- the definition of which is given below:

```
typedef struct H5AC_cache_config_t
{
   /* general configuration fields: */
   int                    version;
```

```
        hbool_t                  rpt_fcn_enabled;

        hbool_t                  open_trace_file;
        hbool_t                  close_trace_file;
        char                    trace_file_name
                                [H5AC__MAX_TRACE_FILE_NAME_LEN + 1];

        hbool_t                  evictions_enabled;

        hbool_t                  set_initial_size;
        size_t                  initial_size;

        double                   min_clean_fraction;

        size_t                  max_size;
        size_t                  min_size;

        long int                 epoch_length;


        /* size increase control fields: */
        enum H5C_cache_incr_mode    incr_mode;

        double                   lower_hr_threshold;

        double                   increment;

        hbool_t                  apply_max_increment;
        size_t                  max_increment;

        enum H5C_cache_flash_incr_mode flash_incr_mode;
        double                   flash_multiple;
        double                   flash_threshold;


        /* size decrease control fields: */
        enum H5C_cache_decr_mode    decr_mode;

        double                   upper_hr_threshold;

        double                   decrement;

        hbool_t                  apply_max_decrement;
        size_t                  max_decrement;

        int                     epochs_before_eviction;

        hbool_t                  apply_empty_reserve;
        double                   empty_reserve;


        /* parallel configuration fields: */
        int                     dirty_bytes_threshold;
        int                     metadata_write_strategy;

      } H5AC_cache_config_t;
```

This structure is defined in H5ACpublic.h. Each field is discussed below and in the associated header comment.

The C API allows you get and set this structure directly. Unfortunately the Fortran API has to do this with individual parameters for each of the fields (with the exception of version).

While the API calls are discussed individually in the reference manual, the following high level discussion of what fields to change for different purposes should be useful.

8.1. General Configuration

The version field is intended to allow THG to change the H5AC_cache_config_t structure without breaking old code. For now, this field should always be set to H5AC__CURR_CACHE_CONFIG_VERSION, even when you are getting the current configuration data from the cache. The library needs the version number to know where fields are located with reference to the supplied base address.

The rpt_fcn_enabled field is a boolean flag that allows you to turn on and off the resize reporting function that reports the activities of the adaptive cache resize code at the end of each epoch -- assuming that it is enabled.

The report function is unsupported, so you are on your own if you use it. Since it dumps status data to stdout, you should not attempt to use it with Windows unless you modify the source. You may find it useful if you want to experiment with different adaptive resize configurations. It is also a convenient way of diagnosing poor cache configuration. Finally, if you do lots of runs with identical behavior, you can use it to determine the metadata cache size needed in each phase of your program so you can set the required cache sizes manually.

The trace file fields are also unsupported. They allow one to open and close a trace file in which all calls to the metadata cache are logged in a user specified file for later analysis. The feature is intended primarily for THG use in debugging or optimizing the metadata cache in cases where users in the field observe obscure failures or poor performance that we cannot re-create in the lab. The trace file will allow us to re-create the exact sequence of cache operations that are triggering the problem.

At present we do not have a play back utility for trace files.

To enable the trace file, you load the full path of the desired trace file into trace_file_name, and set open_trace_file to TRUE. In the parallel case, an ASCII representation of the rank of each process is appended to the supplied trace file name to create a unique trace file name for that process.

To close an open trace file, set close_trace_file to TRUE.

It must be emphasized that you are on your own if you play with the trace file feature. Needless to say, the trace file feature is disabled by default. If you enable it, you will take a large performance hit and generate huge trace files.

The evictions_enabled field is a boolean flag allowing the user to disable the eviction of entries from the metadata cache. Under normal operation conditions, this field will always be set to TRUE.

In rare circumstances, the raw data throughput requirements may be so high that the user wishes to postpone metadata writes so as to reserve I/O throughput for raw data. The evictions_enabled field exists to allow this -- although the user is to be warned that the metadata cache will grow without bound while evictions are disabled. Thus evictions should be re-enabled as soon as possible, and it may be wise to monitor cache size and statistics (to see how to enable statistics, see the debugging facilities section below).

Evictions may only be disabled when the automatic cache resize code is disabled as well. Thus to disable evictions, not only must one set the evictions_enabled field to FALSE, but also set incr_mode to H5C_incr__off, set flash_incr_mode to H5C_flash_incr__off, and set decr_mode to H5C_decr__off.

To re-enable evictions, just set evictions_enabled back to TRUE.

Before passing on to other subjects, it is worth re-iterating that disabling evictions is an extreme step. Before attempting it, you might consider setting a large cache size manually, and flushing the cache just before high raw data throughput is required. This may yield the desired results without the risks inherent in disabling evictions.

The set_initial_size and initial_size fields allow you to specify an initial maximum cache size. If set_initial_size is TRUE, initial_size must lie in the interval [min_size, max_size] (see below for a discussion of the min_size and max_size fields).

If you disable the adaptive cache resizing code (done by setting incr_mode to H5C_incr__off, flash_incr_mode to H5C_flash_incr__off, and decr_mode to H5C_decr__off), you can use these fields to control maximum cache size manually, as the maximum cache size will remain at the initial size.

Note, that the maximum cache size is only modified when set_initial_size is TRUE. This allows the use of configurations specified at compile time to change resize configuration without altering the current maximum size of the cache. Without this feature, an additional call would be required to get the current maximum cache size so as to set the initial_size to the current maximum cache size, and thereby avoid changing it.

The min_clean_fraction sets the current minimum clean size as a fraction of the current max cache size. While this field was originally used only in the parallel version of the library, it now applies to the serial version as well. Its value must lie in the range [0.0, 1.0]. 0.01 is reasonable in the serial case, and 0.3 in the parallel.

A potential interaction, discovered at release 1.8.3, between the enforcement of the min_clean_fraction and the adaptive cache resize code can severely degrade performance. While this interaction is easily dealt in the serial case by setting min_clean_fraction to 0.01, the problem is more difficult in the parallel case. Please see Section 8.5 below for further details.

The max_size and min_size fields specify the range of maximum sizes that may be set for the cache by the automatic resize code. min_size must be less than or equal to max_size, and both must lie in the range [H5C__MIN_MAX_CACHE_SIZE, H5C__MAX_MAX_CACHE_SIZE] -- currently [1 KB, 128 MB]. If you routinely run a cache size in the top half of this range, you should increase the hash table size. To do this, modify the H5C__HASH_TABLE_LEN #define in H5Cpkg.h and re-compile. At present, H5C__HASH_TABLE_LEN must be a power of two.

The epoch_length is the number of cache accesses between runs of the adaptive cache size control algorithms. It is ignored if these algorithms are turned off. It must lie in the range [H5C__MIN_AR_EPOCH_LENGTH, H5C__MAX_AR_EPOCH_LENGTH] -- currently [100, 1000000]. The above constants are defined in H5Cprivate.h. 50000 is a reasonable value.

8.2. Increment Configuration

The incr_mode field specifies the cache size increment algorithm used. Its value must be a member of the H5C_cache_incr_mode enum type -- currently either H5C_incr__off or H5C_incr__threshold (note the double underscores after "incr"). This type is defined in H5Cpublic.h.

If incr_mode is set to H5C_incr__off, regular automatic cache size increases are disabled, and the lower_hr_threshold, increment, apply_max_increment, and max_increment fields are ignored.

The flash_incr_mode field specifies the flash cache size increment algorithm used. Its value must be a member of the H5C_cache_flash_incr_mode enum type -- currently either H5C_flash_incr__off or H5C_flash_incr__add_space (note the double underscores after "incr"). This type is defined in H5Cpublic.h.

If flash_incr_mode is set to H5C_flash_incr__off, flash cache size increases are disabled, and the flash_multiple, and flash_threshold, fields are ignored.

8.2.1. Hit Rate Threshold Cache Size Increase Configuration

If incr_mode is H5C_incr__threshold, the cache size is increased via the hit rate threshold algorithm. The remaining fields in the section are then used as follows:

lower_hr_threshold is the threshold below which the hit rate must fall to trigger an increase. The value must lie in the range [0.0 - 1.0]. In my tests, a relatively high value seems to work best -- 0.9 for example.

increment is the factor by which the old maximum cache size is multiplied to obtain an initial new maximum cache size when an increment is needed. The actual change in size may be smaller as required by max_size (above) and max_increment (discussed below). increment must be greater than or equal to 1.0. If you set it to 1.0, you will effectively turn off the increment code. 2.0 is a reasonable value.

apply_max_increment and max_increment allow the user to specify a maximum increment. If apply_max_increment is TRUE, the cache size will never be increased by more than the number of bytes specified in max_increment in any single increase.

8.2.2. Flash Cache Size Increase Configuration

If flash_incr_mode is set to H5C_flash_incr__add_space, flash cache size increases are enabled. The size of the cache will be increased under the following circumstances:

Let t be the current maximum cache size times the value of the flash_threshold field.

Let x be either the size of the newly inserted entry, the size of the newly loaded entry, or the number of bytes added to the size of the entry under consideration for triggering a flash cache size increase.

If t < x, the basic condition for a flash cache size increase is met, and we proceed as follows:

Let space_needed equal x less the amount of free space in the cache.

Further, let increment equal space_needed times the value of the flash_multiple field. If increment plus the current cache size is greater than max_size (discussed above), reduce increment so that increment plus the current cache size is equal to max_size.

If increment is greater than zero, increase the current cache size by increment. To avoid confusing the other cache size increment or decrement algorithms, start a new epoch. Note however, that we do not cycle the epoch markers if some variant of the age out algorithm is in use.

The use of the flash_threshold field is discussed above. It must be a floating point value in the range of [0.1, 1.0]. 0.25 is a reasonable value.

The use of the flash_multiple field is also discussed above. It must be a floating point value in the range of [0.1, 10.0]. 1.4 is a reasonable value.

8.3. Decrement Configuration

The decr_mode field specifies the cache size decrement algorithm used. Its value must be a member of the H5C_cache_decr_mode enum type -- currently either H5C_decr__off, H5C_decr__threshold, H5C_decr__age_out, or H5C_decr__age_out_with_threshold (note the double underscores after "decr"). This type is defined in H5Cpublic.h.

If decr_mode is set to H5C_decr__off, automatic cache size decreases are disabled, and the remaining fields in the cache size decrease control section are ignored.

8.3.1. Hit Rate Threshold Cache Size Decrease Configuration

If decr_mode is H5C_decr__threshold, the cache size is decreased by the threshold algorithm, and the remaining fields of the decrement section are used as follows:

upper_hr_threshold is the threshold above which the hit rate must rise to trigger cache size reduction. It must be in the range [0.0, 1.0]. In my synthetic tests, very high values like .9995 or .99995 seemed to work best.

decrement is the factor by which the current maximum cache size is multiplied to obtain a tentative new maximum cache size. It must lie in the range [0.0, 1.0]. Relatively large values like .9 seem to work best in my synthetic tests. Note that the actual size reduction may be smaller as required by min_size and max_decrement (discussed below).

apply_max_decrement and max_decrement allow the user to specify a maximum decrement. If apply_max_decrement is TRUE, cache size will never be reduced by more than max_decrement bytes in any single reduction.

With the hit rate threshold cache size decrement algorithm, the remaining fields in the section are ignored.

8.3.2. Ageout Cache Size Reduction

If decr_mode is H5C_decr__age_out the cache size is decreased by the ageout algorithm, and the remaining fields of the decrement section are used as follows:

epochs_before_eviction is the number of epochs an entry must reside unaccessed in the cache before it is evicted. This value must lie in the range [1, H5C__MAX_EPOCH_MARKERS]. H5C__MAX_EPOCH_MARKERS is defined in H5Cprivate.h, and is currently set to 10.

apply_max_decrement and max_decrement are used as in section 2.4.3.1.

apply_empty_reserve and empty_reserve allow the user to specify a minimum empty reserve as discussed in section 2.3.2.2. An empty reserve of 0.05 or 0.1 seems to work well.

The decrement and upper_hr_threshold fields are ignored in this case.

8.3.3. Ageout With Hit Rate Threshold Cache Size Reduction

If decr_mode is H5C_decr__age_out_with_threshold, the cache size is decreased by the ageout with hit rate threshold algorithm, and the fields of decrement section are used as per the Ageout algorithm (see 5.3.2) with the exception of upper_hr_threshold.

Here, upper_hr_threshold is the threshold above which the hit rate must rise to trigger cache size reduction. It must be in the range [0.0, 1.0]. In my synthetic tests, high values like .999 seemed to work well.

8.4. Parallel Configuration

This section is a catch-all for parallel specific configuration data. At present, it has only two fields – dirty_bytes_threshold and metadata_write_strategy.

In PHDF5, all operations that modify metadata must be executed collectively. We used to think that this was enough to ensure consistency across the metadata caches, but since we allow processes to read metadata individually, the order of dirty entries in the LRU list can vary across processes. This in turn can change the order in which dirty metadata cache entries reach the bottom of the LRU and are flushed to disk -- opening the door to messages from the past and messages from the future bugs.

To prevent this, metadata may only be written during a sync point – and only by the metadata cache on process 0 of the file communicator if metadata_write_strategy is set to H5AC_METADATA_WRITE_STRATEGY__PROCESS_0_ONLY. In this case, after the process zero metadata cache writes entries to file, it sends the base addresses of the now clean entries to the other caches, so they can mark these entries clean as well, and then leaves the sync point. The other caches mark the specified entries as clean before they leave the synch point as well. (Observe, that since all caches see the same stream of dirty metadata, they will all have the same set of dirty entries upon sync point entry and exit.)

The different caches know when to synchronize by counting the number of bytes of dirty metadata created by the collective operations modifying metadata. Whenever this count exceeds the value specified in the dirty_bytes_threshold, they all enter the sync point, and process 0 flushes down to its minimum clean size and sends the list of newly cleaned entries to the other caches.

Needless to say, the value of the dirty_bytes_threshold field must be consistent across all the caches operating on a given file.

All dirty metadata can also by flushed under programatic control via the H5Fflush() call. This call must be collective, and will reset the dirty data counts on each metadata cache.

Absent calls to H5Fflush(), dirty metadata will only be flushed when the dirty_bytes_threshold is exceeded, and then only down to the min_clean_fraction. Thus, if a program does all its metadata modifications in one phase, and then doesn't modify metadata thereafter, a residue of dirty metadata will be frozen in the metadata caches for the remainder of the computation -- effectively reducing the sizes of the caches.

In the default configuration, the caches will eventually resize themselves to maintain an acceptable hit rate. However, this will take time, and it will increase the applications footprint in memory.

If your application behaves in this manner, you can avoid this by a collective call to H5Fflush() immediately after the metadata modification phase.

Management of metadata writes when metadata_write_strategy is set to H5AC_METADATA_WRITE_STRATEGY__DISTRIBUTED differs from that under H5AC_METADATA_WRITE_STRATEGY__PROCESS_0_ONLY in that while the process 0 cache selects the entries to be written, the actual writes are divided between the metadata caches.

8.5. Interactions

Evictions may not be disabled unless the automatic cache resize code is disabled as well (by setting decr_mode to H5C_decr__off, flash_decr_mode to H5C_flash_incr__add_space, and incr_mode to H5C_incr__off) -- thus placing the cache size under the direct control of the user program.

There is no logical necessity for this restriction. It is imposed because it simplifies testing greatly, and because I can't see any reason why one would want to disable evictions while the automatic cache size adjustment code was enabled. This restriction can be relaxed if anyone can come up with a good reason to do so.

At present there are two interactions between the increment and decrement sections of the configuration.

If incr_mode is H5C_incr__threshold, and decr_mode is either H5C_decr__threshold or H5C_decr__age_out_with_threshold, then lower_hr_threshold must be strictly less than upper_hr_threshold.

Also, if the flash cache size increment code is enabled and is triggered, it will restart the current epoch without calling any other cache size increment or decrement code.

In both the serial and parallel cases, there is the potential for an interaction between the min_clean_fraction and the cache size increment code that can severely degrade performance. Specifically, if the min_clean_fraction is large enough, it is possible that keeping the specified fraction of the cache clean may generate enough flushes to seriously degrade performance even though the hit rate is excellent.

In the serial case, this is easily dealt with by selecting a very small min_clean_fraction -- 0.01 for example -- as this still avoids the "metadata blizzard" phenomenon that appears when the cache fills with dirty metadata and must then flush all of it before evicting an entry to make space for a new entry.

The problem is more difficult in the parallel case, as the min_clean_fraction is used ensure that the cache contains clean entries that can be evicted to make space for new entries when metadata writes are forbidden -- i.e. between sync points.

This issue was discovered shortly before release 1.8.3 and an automated solution has not been implemented. Should it become an issue for an application, try manually setting the cache size to ~1.5 times the maximum working set size for the application, and leave min_clean_fraction set to 0.3.

You can approximate the working set size of your application via repeated calls to H5Fget_mdc_size() and H5Fget_mdc_hit_rate() while running your program with the cache resize code enabled. The maximum value returned by H5Fget_mdc_size() should be a reasonable approximation -- particularly if the associated hit rate is good.

In the parallel case, there is also an interaction between min_clean_fraction and dirty_bytes_threshold. Absent calls to H5Fflush() (discussed above), the upper bound on the amount of dirty data in the metadata caches will oscillate between (1 - min_clean_fraction) times current maximum cache size, and that value plus the dirty_bytes_threshold. Needless to say, it will be best if the min_size, min_clean_fraction, and the dirty_bytes_threshold are chosen so that the cache can't fill with dirty data.

8.6. Default Metadata Cache Configuration

Starting with release 1.8.3, HDF5 provides different default metadata cache configurations depending on whether the library is compiled for serial or parallel.

The default configuration for the serial case is as follows:

```
{                                                          \
  /* int         version              = */ H5C__CURR_AUTO_SIZE_CTL_VER,     \
  /* hbool_t     rpt_fcn_enabled       = */ FALSE,                        \
  /* hbool_t     open_trace_file       = */ FALSE,                        \
  /* hbool_t     close_trace_file      = */ FALSE,                        \
  /* char        trace_file_name[]     = */ "",                           \
  /* hbool_t     evictions_enabled     = */ TRUE,                         \
  /* hbool_t     set_initial_size      = */ TRUE,                         \
  /* size_t      initial_size          = */ ( 2 * 1024 * 1024),           \
  /* double      min_clean_fraction    = */ 0.01f,                        \
  /* size_t      max_size              = */ (32 * 1024 * 1024),           \
  /* size_t      min_size              = */ ( 1 * 1024 * 1024),           \
  /* long int    epoch_length          = */ 50000,                        \
  /* enum H5C_cache_incr_mode incr_mode = */ H5C_incr__threshold,           \
  /* double      lower_hr_threshold    = */ 0.9f,                         \
  /* double      increment             = */ 2.0f,                         \
  /* hbool_t     apply_max_increment   = */ TRUE,                         \
  /* size_t      max_increment         = */ (4 * 1024 * 1024),            \
  /* enum H5C_cache_flash_incr_mode      */                               \
  /*                 flash_incr_mode = */ H5C_flash_incr__add_space,      \
  /* double      flash_multiple        = */ 1.4f,                         \
  /* double      flash_threshold       = */ 0.25f,                        \
  /* enum H5C_cache_decr_mode decr_mode = */ H5C_decr__age_out_with_threshold,\
  /* double      upper_hr_threshold    = */ 0.999f,                       \
  /* double      decrement             = */ 0.9f,                        \
  /* hbool_t     apply_max_decrement    = */ TRUE,                         \
  /* size_t      max_decrement         = */ (1 * 1024 * 1024),             \
  /* int         epochs_before_eviction = */ 3,                           \
  /* hbool_t     apply_empty_reserve    = */ TRUE,                         \
  /* double      empty_reserve         = */ 0.1f,                         \
  /* int         dirty_bytes_threshold  = */ (256 * 1024),                  \
  /* int        metadata_write_strategy = */                              \
                          H5AC_METADATA_WRITE_STRATEGY__DISTRIBUTED  \
}
```

The default configuration for the parallel case is as follows:

```
{                                                                \
  /* int         version               = */ H5AC__CURR_CACHE_CONFIG_VERSION, \
  /* hbool_t     rpt_fcn_enabled        = */ FALSE,                        \
  /* hbool_t     open_trace_file        = */ FALSE,                        \
  /* hbool_t     close_trace_file       = */ FALSE,                        \
  /* char        trace_file_name[]      = */ "",                           \
  /* hbool_t     evictions_enabled      = */ TRUE,                         \
  /* hbool_t     set_initial_size       = */ TRUE,                         \
  /* size_t      initial_size          = */ ( 2 * 1024 * 1024),            \
  /* double      min_clean_fraction     = */ 0.3f,                         \
  /* size_t      max_size              = */ (32 * 1024 * 1024),            \
  /* size_t      min_size              = */ (1 * 1024 * 1024),             \
  /* long int    epoch_length          = */ 50000,                        \
  /* enum H5C_cache_incr_mode incr_mode = */ H5C_incr__threshold,            \
  /* double      lower_hr_threshold     = */ 0.9f,                         \
  /* double      increment             = */ 2.0f,                         \
  /* hbool_t     apply_max_increment    = */ TRUE,                         \
  /* size_t      max_increment         = */ (4 * 1024 * 1024),             \
  /* enum H5C_cache_flash_incr_mode       */                              \
  /*                 flash_incr_mode = */ H5C_flash_incr__add_space,       \
  /* double      flash_multiple        = */ 1.0f,                         \
  /* double      flash_threshold       = */ 0.25f,                        \
  /* enum H5C_cache_decr_mode decr_mode = */ H5C_decr__age_out_with_threshold, \
  /* double      upper_hr_threshold     = */ 0.999f,                       \
  /* double      decrement             = */ 0.9f,                         \
  /* hbool_t     apply_max_decrement    = */ TRUE,                         \
  /* size_t      max_decrement         = */ (1 * 1024 * 1024),             \
  /* int         epochs_before_eviction = */ 3,                           \
  /* hbool_t     apply_empty_reserve    = */ TRUE,                         \
  /* double      empty_reserve         = */ 0.1f,                         \
  /* int         dirty_bytes_threshold  = */ (256 * 1024),                  \
  /* int        metadata_write_strategy = */                              \
                              H5AC__DEFAULT_METADATA_WRITE_STRATEGY  \
}
```

The default serial configuration should be adequate for most purposes.

The same may not be true for the default parallel configuration due the interaction between the min_clean_fraction and the cache size increase code. See Section 8.6 above for further details.

Should you need to change the default configuration, it can be found in H5ACprivate.h. Look for the definition of H5AC__DEFAULT_CACHE_CONFIG.

8.7. Manual Metadata Cache Size Control

As discussed above, H5AC_cache_config_t has facilities that allow you to control the metadata cache size directly. Use H5Fget_mdc_config() and H5Fset_mdc_config() to get and set the metadata cache configuration on an open file. Use

H5Pget_mdc_config() and H5Pset_mdc_config() to get and set the initial metadata cache configuration in a file access property list. Recall that this list contains configuration data used when opening a file.

Use H5Fget_mdc_hit_rate() to get the average hit rate since the last time the hit rate stats were reset. This happens automatically at the beginning of each epoch if the adaptive cache resize code is enabled. You can also do it manually with H5Freset_mdc_hit_rate_stats(). Be careful about doing this if the adaptive cache resize code is enabled, as you may confuse it.

Use H5Fget_mdc_size() to get metadata cache size data on an open file.

Finally, note that cache size and cache footprint are two different things -- in my tests, the cache footprint (as inferred from the UNIX top command) is typically about three times the maximum cache size. I haven't tracked it down, but I would guess that most of this is due to the very small typical cache entry size combined with the rather large size of cache entry header structure. This should be investigated further, but there are other matters of higher priority.

9. Metadata Cache Debugging Facilities

The metadata cache has a variety of debugging facilities that may be of use. I doubt that any other than the report function and the trace file will ever be accessible via the API, but they are relatively easy to turn on in the source code.

As mentioned above, you can use the rpt_fcn_enabled field of the configuration structure to enable the default reporting function (H5C_def_auto_resize_rpt_fcn() in H5C.c). If this function doesn't work for you, you will have to write your own. In particular, remember that it uses stdout, so it will probably be unhappy under Windows.

There is also extensive statistics collection code. Use H5C_COLLECT_CACHE_STATS and H5C_COLLECT_CACHE_ENTRY_STATS in H5Cprivate.h to turn this on. If you also turn on H5AC_DUMP_STATS_ON_CLOSE in H5ACprivate.h, stats will be dumped when you close a file. Alternatively you can call H5C_stats() and H5C_stats__reset() within the library to dump and reset stats. Both of these functions are defined in H5C.c. Note that the stats code tends to suffer bit rot between uses, so you may have to work on it a bit to get it going.

Finally, the cache also contains extensive sanity checking code. Some of this is turned on when you compile in debug mode, but to enable more, turn on H5C_DO_SANITY_CHECKS in H5Cprivate.h. To enable the full suite, turn on H5C_DO_EXTREME_SANITY_CHECKS, also in H5Cprivate.h. Note that the overhead of the extreme sanity checks is quite large. Note also that there are sanity check for tagging and the slist – see control #defines in H5Cprivate.h.

10. Points to Consider When Writing a Metadata Cache Client

In terms of required data structures and callbacks, the check list for adding a new metadata cache client is short and simple:

1. Allocate an ID for each new client by adding identifiers to the H5AC_type_t enumerated type in H5ACprivate.h. As this is the canonical location for client IDs, observe that the definition of H5AC_type_t enum is in effect a list of all cache clients in the library. Note, however, that several test code clients share the same ID. Also the stand alone tests for the metadata cache use #defines for client IDs, which may be found in cache_common.h.
2. Implement the callback functions required for your new client(s). Note that the image_len, pre-serialize, notify, clear, and get_fsf_size callbacks may not be required for your client. All other callbacks are mandatory. Note that by custom, all your callback functions should be in a file named H5+cache.c, where "+" is replaced with the one or two capitalized letters that indicate your client.
3. Define a static instance of H5AC_class_t for each new client, and initialize it with the new ID, a name (for debugging purposes), flags as appropriate, and pointers to your callback functions. Use a pointer to this instance in all metadata cache calls that require a type and that deal with your new client.

The above check list makes integrating a client with the metadata look simple. However, there are a number of design issues that may complicate the task. The following subsections contain brief discussions of the issues encountered to date.

10.1. Metadata File Space Allocation / Deallocation

Before the version 3 metadata cache, allocation and deallocation of file space for metadata was handled by the client. At present, the client is still responsible for file space allocation. However, the client may choose to delegate file space deallocation to the metadata cache if desired.

To instruct the metadata cache to free the file space currently allocated to a metadata cache entry, include the H5AC__FREE_FILE_SPACE_FLAG with the H5AC__FLUSH_INVALIDATE_FLAG when unprotecting the entry, or include the H5AC__FREE_FILE_SPACE_FLAG in a call to H5AC_expunge_entry(). Note that the entry must be in real file space if the H5AC__FREE_FILE_SPACE_FLAG appears.

When freeing file space, the metadata cache normally presumes that the file space allocation has the same base address and length as the entry. While this is usually the case, it need not be so. For example, the extensible array allows some metadata entries to be paged into the cache. Since these pages cannot move or change size, space for the header and all pages is allocated at the same time, and freed only when the header is freed. Thus the header has once size for being loaded into or flushed from the cache, and another for purposes of freeing file space. To handle this issue, the cache supports the optional get_fsf_size() callback. If it is defined, it is called before freeing file space, and the size returned is used for purposes of the file space free operation.

10.2. Metadata Resizing and Relocation

It would be quite convenient if all pieces of metadata in HDF5 files were of fixed sizes that were known before the entries were read or written. Such is not the case, and thus the metadata cache is able to deal with entries whose size is not known at load time, and with entries which change size and/or location on flush.

10.2.1. Unknown Entry Size at Entry Load Time

If the size of cache entries used by your cache client is not known at load time, you will need to define the get_image_len callback and set either the H5AC__CLASS_SPECULATIVE_LOAD_FLAG or the H5AC__CLASS_COMPRESSED_FLAG.

The get_load_size callback must return a value that is large enough for your deserialize callback to determine the actual size of the entry (if H5AC__CLASS_SPECULATIVE_LOAD_FLAG is set) or a size that is an upper bound on the size of the entry (if H5AC__CLASS_COMPRESSED_FLAG is set).

The get_image_len callback must return the actual size of the entry, so that the cache can make a second try at loading it (if actual size is greater than the estimate returned by the get_load_size callback and H5AC__CLASS_SPECULATIVE_LOAD_FLAG is set), or simply reduce the size of the entry if the actual size is less than the value returned by your get_load_size callback.

For examples of clients which use these facilities, just grep for H5AC__CLASS_SPECULATIVE_LOAD_FLAG and/or H5AC__CLASS_COMPRESSED_FLAG in the src directory. I note that at present, the H5AC__CLASS_COMPRESSED_FLAG does not appear to be used.

10.2.2. Entry Size and/or Location Change at Flush Time

If the metadata entries maintained by your client can change size and/or location at flush time, you will have to implement the pre-serialize callback for your client, and have it perform the size and/or location changes if necessary, and report them back to the metadata cache.

The free space manager and the fractal heap are examples of cache clients which use these facilities, as they often put frequently modified entries in temporary file space beyond the end of file – thus avoiding frequent file space allocations / deallocations when the metadata entry size changes. However, before entries can be written to file, they must have real file space allocated for them, and be moved to their new locations.

Even if you don't use temporary file space, it is still possible for your metadata to change size, and therefore have to change location at flush time. For example, if you use compression filters, changes to the contents of your fixed size entry may increase its compressed size – forcing relocation within the HDF5 file.

10.3. Metadata Consistency Considerations

When your metadata is written to file, you must ensure that it contains consistent data. In particular, if it contains addresses of other pieces of metadata, you must ensure that these addresses are correct. The Fractal Heap and the Free Space Manager are examples of different ways of dealing with this issue.

To a first order approximation, the Fractal Heap may be thought of as a tree structure, with a header at the root, indirect blocks forming the internal nodes, and direct blocks at the leaves. With the exception of the header, all can reside in temporary file space, and since the direct blocks may be compressed, they may have to relocate even if they are already in real file space. Further, each node in the tree contains the address of the header, and of its immediate children.

The Fractal Heap deals with the potential inconsistency issue by using flush dependencies to ensure that all descendents of any node n have been relocated to real file space if necessary and flushed to disk before n is flushed. Needless to say, changes in child addresses are noted in the parent as they are made.

This use of flush dependencies simplifies maintaining consistency, however it does require that flush dependencies be set up as entries are loaded or inserted into the cache, and taken down when child entries are evicted. The Fractal Heap uses the notify callback to implement this.

In contrast, the Free Space Manager metadata is much simpler, consisting of a header and possibly a section info entry for each free space manager. The header is of fixed size, and is always in real file space. The section info metadata entry may or may not exist, and if it does, it is frequently in temporary file space.

The Free Space Manager handles this issue with a pre-serialize callback that creates and/or moves the section info in real file space. This allows correct addresses and sizes to be stored in the header – which are then serialized in the subsequent serialize callback.

Finally, if your client maintains it own internal dirty bits, you will need to implement the clear callback. The primary use for this callback is in the parallel version of the library where entries are typically written by only one process. All other processes must be able to mark entries as clean when instructed to do so. Note that this means that the pre-serialize and serialize callback will not be called – which is a reason why we do not allow metadata to change size or relocate itself in PHDF5.

10.4. Test Code

In cache clients created for test purposes, it is frequently useful to avoid doing any file I/O. The metadata cache supports several flags for this purpose – see the discussion of the flags fields in the header comment for H5C_class_t in Appendix 3 for details. Make sure that you only use these flags in test code.

Appendix 1: H5C_t

The definition of struct H5C_t is reproduced below, along with its header comment. Please note the documentation of each field in the structure. Developers are encouraged to update and correct this header comment in H5Cpkg.h as appropriate. While the following was correct as of the preparation of this document, the version in H5Cpkg.h is the master copy, and should be referred to for up to date information.

Note the use of updates rather than re-writes to document changes that have not been fully propagated through the code. The objective is to provide explanations for quirks in the code that would otherwise be confusing.

```
/****************************************************************************
 *
 * structure H5C_t
 *
 * Catchall structure for all variables specific to an instance of the cache.
 *
 * While the individual fields of the structure are discussed below, the
 * following overview may be helpful.
 *
 * Entries in the cache are stored in an instance of H5TB_TREE, indexed on
 * the entry's disk address.  While the H5TB_TREE is less efficient than
 * hash table, it keeps the entries in address sorted order.  As flushes
 * in parallel mode are more efficient if they are issued in increasing
 * address order, this is a significant benefit.  Also the H5TB_TREE code
 * was readily available, which reduced development time.
 *
 * While the cache was designed with multiple replacement policies in mind,
 * at present only a modified form of LRU is supported.
 *
 *                                              JRM - 4/26/04
 *
 * Profiling has indicated that searches in the instance of H5TB_TREE are
 * too expensive.  To deal with this issue, I have augmented the cache
 * with a hash table in which all entries will be stored.  Given the
 * advantages of flushing entries in increasing address order, the TBBT
 * is retained, but only dirty entries are stored in it.  At least for
 * now, we will leave entries in the TBBT after they are flushed.
 *
 * Note that index_size and index_len now refer to the total size of
 * and number of entries in the hash table.
 *
 *                                              JRM - 7/19/04
 *
 * The TBBT has since been replaced with a skip list.  This change
 * greatly predates this note.
 *
 *                                              JRM - 9/26/05
 *
 * magic:       Unsigned 32 bit integer always set to H5C__H5C_T_MAGIC.
 *              This field is used to validate pointers to instances of
 *              H5C_t.
 *
 * flush_in_progress: Boolean flag indicating whether a flush is in
 *              progress.
 *
 * trace_file_ptr:  File pointer pointing to the trace file, which is used
 *              to record cache operations for use in simulations and design
 *              studies.  This field will usually be NULL, indicating that
 *              no trace file should be recorded.
 *
 *              Since much of the code supporting the parallel metadata
 *              cache is in H5AC, we don't write the trace file from
 *              H5C.  Instead, H5AC reads the trace_file_ptr as needed.
 *
 *              When we get to using H5C in other places, we may add
 *              code to write trace file data at the H5C level as well.
 *
 * aux_ptr:     Pointer to void used to allow wrapper code to associate
 *              its data with an instance of H5C_t.  The H5C cache code
 *              sets this field to NULL, and otherwise leaves it alone.
 *
 * max_type_id: Integer field containing the maximum type id number assigned
 *              to a type of entry in the cache.  All type ids from 0 to
 *              max_type_id inclusive must be defined.  The names of the
 *              types are stored in the type_name_table discussed below, and
 *              indexed by the ids.
 *
 * type_name_table_ptr: Pointer to an array of pointer to char of length
 *              max_type_id + 1.  The strings pointed to by the entries
 *              in the array are the names of the entry types associated
 *              with the indexing type IDs.
 *
 * max_cache_size:  Nominal maximum number of bytes that may be stored in the
```

```
*              cache.  This value should be viewed as a soft limit, as the
*              cache can exceed this value under the following circumstances:
*
*              a) All entries in the cache are protected, and the cache is
*                 asked to insert a new entry.  In this case the new entry
*                 will be created.  If this causes the cache to exceed
*                 max_cache_size, it will do so.  The cache will attempt
*                 to reduce its size as entries are unprotected.
*
*              b) When running in parallel mode, the cache may not be
*                 permitted to flush a dirty entry in response to a read.
*                 If there are no clean entries available to evict, the
*                 cache will exceed its maximum size.  Again the cache
*                 will attempt to reduce its size to the max_cache_size
*                 limit on the next cache write.
*
*              c) When an entry increases in size, the cache may exceed
*                 the max_cache_size limit until the next time the cache
*                 attempts to load or insert an entry.
*
*              d) When the evictions_enabled field is false (see below),
*                 the cache size will increase without limit until the
*                 field is set to true.
*
* min_clean_size: Nominal minimum number of clean bytes in the cache.
*              The cache attempts to maintain this number of bytes of
*              clean data so as to avoid case b) above.  Again, this is
*              a soft limit.
*
*
* In addition to the call back functions required for each entry, the
* cache requires the following call back functions for this instance of
* the cache as a whole:
*
* check_write_permitted:  In certain applications, the cache may not
*              be allowed to write to disk at certain time.  If specified,
*              the check_write_permitted function is used to determine if
*              a write is permissible at any given point in time.
*
*              If no such function is specified (i.e. this field is NULL),
*              the cache uses the following write_permitted field to
*              determine whether writes are permitted.
*
* write_permitted: If check_write_permitted is NULL, this boolean flag
*              indicates whether writes are permitted.
*
* log_flush:   If provided, this function is called whenever a dirty
*              entry is flushed to disk.
*
*
* In cases where memory is plentiful, and performance is an issue, it may
* be useful to disable all cache evictions, and thereby postpone metadata
* writes.  The following field is used to implement this.
*
* evictions_enabled:  Boolean flag that is initialized to TRUE.  When
*              this flag is set to FALSE, the metadata cache will not
*              attempt to evict entries to make space for newly protected
*              entries, and instead the will grow without limit.
*
```

```
*          Needless to say, this feature must be used with care.
*
*
* The cache requires an index to facilitate searching for entries.  The
* following fields support that index.
*
* index_len:   Number of entries currently in the hash table used to index
*          the cache.
*
* index_size:  Number of bytes of cache entries currently stored in the
*          hash table used to index the cache.
*
*          This value should not be mistaken for footprint of the
*          cache in memory.  The average cache entry is small, and
*          the cache has a considerable overhead.  Multiplying the
*          index_size by three should yield a conservative estimate
*          of the cache's memory footprint.
*
* clean_index_size: Number of bytes of clean entries currently stored in
*          the hash table.  Note that the index_size field (above)
*          is also the sum of the sizes of all entries in the cache.
*          Thus we should have the invarient that clean_index_size +
*          dirty_index_size == index_size.
*
*          WARNING:
*
*            The value of the clean_index_size must not be mistaken
*            for the current clean size of the cache.  Rather, the
*            clean size of the cache is the current value of
*            clean_index_size plus the amount of empty space (if any)
*            in the cache.
*
* dirty_index_size: Number of bytes of dirty entries currently stored in
*          the hash table.  Note that the index_size field (above)
*          is also the sum of the sizes of all entries in the cache.
*          Thus we should have the invarient that clean_index_size +
*          dirty_index_size == index_size.
*
* index:      Array of pointer to H5C_cache_entry_t of size
*          H5C__HASH_TABLE_LEN.  At present, this value is a power
*          of two, not the usual prime number.
*
*          I hope that the variable size of cache elements, the large
*          hash table size, and the way in which HDF5 allocates space
*          will combine to avoid problems with periodicity.  If so, we
*          can use a trivial hash function (a bit-and and a 3 bit left
*          shift) with some small savings.
*
*          If not, it will become evident in the statistics. Changing
*          to the usual prime number length hash table will require
*          changing the H5C__HASH_FCN macro and the deletion of the
*          H5C__HASH_MASK #define.  No other changes should be required.
*
*
* When we flush the cache, we need to write entries out in increasing
* address order.  An instance of a skip list is used to store dirty entries in
* sorted order.  Whether it is cheaper to sort the dirty entries as needed,
* or to maintain the list is an open question.  At a guess, it depends
* on how frequently the cache is flushed.  We will see how it goes.
```

```
*
* For now at least, I will not remove dirty entries from the list as they
* are flushed. (this has been changed -- dirty entries are now removed from
* the skip list as they are flushed.  JRM - 10/25/05)
*
* slist_changed: Boolean flag used to indicate whether the contents of
*             the slist has changed since the last time this flag was
*             reset.  This is used in the cache flush code to detect
*             conditions in which pre-serialize or serialize callbacks
*             have modified the slist -- which obliges us to restart
*             the scan of the slist from the beginning.
*
* slist_change_in_pre_serialize: Boolean flag used to indicate that
*             a pre_serialize call has modified the slist since the
*             last time this flag was reset.
*
* slist_change_in_serialize: Boolean flag used to indicate that
*             a serialize call has modified the slist since the
*             last time this flag was reset.
*
* slist_len:   Number of entries currently in the skip list
*             used to maintain a sorted list of dirty entries in the
*             cache.
*
* slist_size:  Number of bytes of cache entries currently stored in the
*             skip list used to maintain a sorted list of
*             dirty entries in the cache.
*
* slist_ptr:   pointer to the instance of H5SL_t used maintain a sorted
*             list of dirty entries in the cache.  This sorted list has
*             two uses:
*
*             a) It allows us to flush dirty entries in increasing address
*                order, which results in significant savings.
*
*             b) It facilitates checking for adjacent dirty entries when
*                attempting to evict entries from the cache.  While we
*                don't use this at present, I hope that this will allow
*                some optimizations when I get to it.
*
* num_last_entries: The number of entries in the cache that can only be
*             flushed after all other entries in the cache have
*             been flushed. At this time, this will only ever be
*             one entry (the superblock), and the code has been
*             protected with HDasserts to enforce this. This restraint
*             can certainly be relaxed in the future if the need for
*             multiple entries being flushed last arises, though
*             explicit tests for that case should be added when said
*             HDasserts are removed.
*
*             Update: There are now two possible last entries
*             (superblock and file driver info message).  This
*             number will probably increase as we add superblock
*             messages.   JRM -- 11/18/14
*
* With the addition of the fractal heap, the cache must now deal with
* the case in which entries may be dirtied, moved, or have their sizes
* changed during a flush.  To allow sanity checks in this situation, the
* following two fields have been added.  They are only compiled in when
* H5C_DO_SANITY_CHECKS is TRUE.
*
* slist_len_increase: Number of entries that have been added to the
*             slist since the last time this field was set to zero.
*             Note that this value can be negative.
*
* slist_size_increase: Total size of all entries that have been added
*             to the slist since the last time this field was set to
*             zero.  Note that this value can be negative.
*
*
* When a cache entry is protected, it must be removed from the LRU
* list(s) as it cannot be either flushed or evicted until it is unprotected.
* The following fields are used to implement the protected list (pl).
*
* pl_len:      Number of entries currently residing on the protected list.
*
* pl_size:     Number of bytes of cache entries currently residing on the
*             protected list.
*
* pl_head_ptr: Pointer to the head of the doubly linked list of protected
*             entries.  Note that cache entries on this list are linked
*             by their next and prev fields.
*
*             This field is NULL if the list is empty.
*
* pl_tail_ptr: Pointer to the tail of the doubly linked list of protected
*             entries.  Note that cache entries on this list are linked
*             by their next and prev fields.
*
*             This field is NULL if the list is empty.
*
*
* For very frequently used entries, the protect/unprotect overhead can
* become burdensome.  To avoid this overhead, I have modified the cache
* to allow entries to be "pinned".  A pinned entry is similar to a
* protected entry, in the sense that it cannot be evicted, and that
* the entry can be modified at any time.
*
* Pinning an entry has the following implications:
*
*      1) A pinned entry cannot be evicted.  Thus unprotected
*         pinned entries reside in the pinned entry list, instead
*         of the LRU list(s) (or other lists maintained by the current
*         replacement policy code).
*
*      2) A pinned entry can be accessed or modified at any time.
*         This places an additional burden on the associated pre-serialize
*         and serialize callbacks, which must ensure the the entry is in
*         a consistant state before creating an image of it.
*
*      3) A pinned entry can be marked as dirty (and possibly
*         change size) while it is unprotected.
*
*      4) The flush-destroy code must allow pinned entries to
*         be unpinned (and possibly unprotected) during the
*         flush.
*
* Since pinned entries cannot be evicted, they must be kept on a pinned
* entry list (pel), instead of being entrusted to the replacement policy
* code.
*
* Maintaining the pinned entry list requires the following fields:
*
* pel_len:     Number of entries currently residing on the pinned
*              entry list.
*
* pel_size:    Number of bytes of cache entries currently residing on
*              the pinned entry list.
*
* pel_head_ptr: Pointer to the head of the doubly linked list of pinned
*              but not protected entries.  Note that cache entries on
*              this list are linked by their next and prev fields.
*
*              This field is NULL if the list is empty.
*
* pel_tail_ptr: Pointer to the tail of the doubly linked list of pinned
*              but not protected entries.  Note that cache entries on
*              this list are linked by their next and prev fields.
*
*              This field is NULL if the list is empty.
*
*
* The cache must have a replacement policy, and the fields supporting this
* policy must be accessible from this structure.
*
* While there has been interest in several replacement policies for
* this cache, the initial development schedule is tight.  Thus I have
* elected to support only a modified LRU (least recently used) policy
* for the first cut.
*
* To further simplify matters, I have simply included the fields needed
* by the modified LRU in this structure.  When and if we add support for
* other policies, it will probably be easiest to just add the necessary
* fields to this structure as well -- we only create one instance of this
* structure per file, so the overhead is not excessive.
*
*
* Fields supporting the modified LRU policy:
*
* See most any OS text for a discussion of the LRU replacement policy.
*
* When operating in parallel mode, we must ensure that a read does not
* cause a write.  If it does, the process will hang, as the write will
* be collective and the other processes will not know to participate.
*
* To deal with this issue, I have modified the usual LRU policy by adding
* clean and dirty LRU lists to the usual LRU list.  In general, these
* lists are only exist in parallel builds.
*
* The clean LRU list is simply the regular LRU list with all dirty cache
* entries removed.
*
* Similarly, the dirty LRU list is the regular LRU list with all the clean
* cache entries removed.
*
* When reading in parallel mode, we evict from the clean LRU list only.
* This implies that we must try to ensure that the clean LRU list is
```

```
 * reasonably well stocked at all times.
 *
 * We attempt to do this by trying to flush enough entries on each write
 * to keep the cLRU_list_size >= min_clean_size.
 *
 * Even if we start with a completely clean cache, a sequence of protects
 * without unprotects can empty the clean LRU list.  In this case, the
 * cache must grow temporarily.  At the next sync point, we will attempt to
 * evict enough entries to reduce index_size to less than max_cache_size.
 * While this will usually be possible, all bets are off if enough entries
 * are protected.
 *
 * Discussions of the individual fields used by the modified LRU replacement
 * policy follow:
 *
 * LRU_list_len:  Number of cache entries currently on the LRU list.
 *
 *              Observe that LRU_list_len + pl_len + pel_len must always
 *              equal index_len.
 *
 * LRU_list_size:  Number of bytes of cache entries currently residing on the
 *              LRU list.
 *
 *              Observe that LRU_list_size + pl_size + pel_size must always
 *              equal index_size.
 *
 * LRU_head_ptr:  Pointer to the head of the doubly linked LRU list.  Cache
 *              entries on this list are linked by their next and prev fields.
 *
 *              This field is NULL if the list is empty.
 *
 * LRU_tail_ptr:  Pointer to the tail of the doubly linked LRU list.  Cache
 *              entries on this list are linked by their next and prev fields.
 *
 *              This field is NULL if the list is empty.
 *
 * cLRU_list_len: Number of cache entries currently on the clean LRU list.
 *
 *              Observe that cLRU_list_len + dLRU_list_len must always
 *              equal LRU_list_len.
 *
 * cLRU_list_size:  Number of bytes of cache entries currently residing on
 *              the clean LRU list.
 *
 *              Observe that cLRU_list_size + dLRU_list_size must always
 *              equal LRU_list_size.
 *
 * cLRU_head_ptr:  Pointer to the head of the doubly linked clean LRU list.
 *              Cache entries on this list are linked by their aux_next and
 *              aux_prev fields.
 *
 *              This field is NULL if the list is empty.
 *
 * cLRU_tail_ptr:  Pointer to the tail of the doubly linked clean LRU list.
 *              Cache entries on this list are linked by their aux_next and
 *              aux_prev fields.
 *
 *              This field is NULL if the list is empty.
 *
```

```
* dLRU_list_len: Number of cache entries currently on the dirty LRU list.
*
*            Observe that cLRU_list_len + dLRU_list_len must always
*            equal LRU_list_len.
*
* dLRU_list_size:  Number of cache entries currently on the dirty LRU list.
*
*            Observe that cLRU_list_len + dLRU_list_len must always
*            equal LRU_list_len.
*
* dLRU_head_ptr:  Pointer to the head of the doubly linked dirty LRU list.
*            Cache entries on this list are linked by their aux_next and
*            aux_prev fields.
*
*            This field is NULL if the list is empty.
*
* dLRU_tail_ptr:  Pointer to the tail of the doubly linked dirty LRU list.
*            Cache entries on this list are linked by their aux_next and
*            aux_prev fields.
*
*            This field is NULL if the list is empty.
*
*
* Automatic cache size adjustment:
*
* While the default cache size is adequate for most cases, we can run into
* cases where the default is too small.  Ideally, we will let the user
* adjust the cache size as required.  However, this is not possible in all
* cases.  Thus I have added automatic cache size adjustment code.
*
* The configuration for the automatic cache size adjustment is stored in
* the structure described below:
*
* size_increase_possible:  Depending on the configuration data given
*            in the resize_ctl field, it may or may not be possible
*            to increase the size of the cache.  Rather than test for
*            all the ways this can happen, we simply set this flag when
*            we receive a new configuration.
*
* flash_size_increase_possible: Depending on the configuration data given
*            in the resize_ctl field, it may or may not be possible
*            for a flash size increase to occur.  We set this flag
*            whenever we receive a new configuration so as to avoid
*            repeated calculations.
*
* flash_size_increase_threshold: If a flash cache size increase is possible,
*            this field is used to store the minimum size of a new entry
*            or size increase needed to trigger a flash cache size
*            increase.  Note that this field must be updated whenever
*            the size of the cache is changed.
*
* size_decrease_possible:  Depending on the configuration data given
*            in the resize_ctl field, it may or may not be possible
*            to decrease the size of the cache.  Rather than test for
*            all the ways this can happen, we simply set this flag when
*            we receive a new configuration.
*
* cache_full:  Boolean flag used to keep track of whether the cache is
*            full, so we can refrain from increasing the size of a
*            cache which hasn't used up the space alotted to it.
*
*            The field is initialized to FALSE, and then set to TRUE
*            whenever we attempt to make space in the cache.
*
* resize_enabled:  This is another convenience flag which is set whenever
*            a new set of values for resize_ctl are provided.  Very
*            simply,
*
*                resize_enabled = size_increase_possible ||
*                                 size_decrease_possible;
*
* size_decreased:  Boolean flag set to TRUE whenever the maximun cache
*            size is decreased.  The flag triggers a call to
*            H5C_make_space_in_cache() on the next call to H5C_protect().
*
* resize_ctl:  Instance of H5C_auto_size_ctl_t containing configuration
*            data for automatic cache resizing.
*
* epoch_markers_active:  Integer field containing the number of epoch
*            markers currently in use in the LRU list.  This value
*            must be in the range [0, H5C__MAX_EPOCH_MARKERS - 1].
*
* epoch_marker_active:  Array of boolean of length H5C__MAX_EPOCH_MARKERS.
*            This array is used to track which epoch markers are currently
*            in use.
*
* epoch_marker_ringbuf:  Array of int of length H5C__MAX_EPOCH_MARKERS + 1.
*
*            To manage the epoch marker cache entries, it is necessary
*            to track their order in the LRU list.  This is done with
*            epoch_marker_ringbuf.  When markers are inserted at the
*            head of the LRU list, the index of the marker in the
*            epoch_markers array is inserted at the tail of the ring
*            buffer.  When it becomes the epoch_marker_active'th marker
*            in the LRU list, it will have worked its way to the head
*            of the ring buffer as well.  This allows us to remove it
*            without scanning the LRU list if such is required.
*
* epoch_marker_ringbuf_first: Integer field containing the index of the
*            first entry in the ring buffer.
*
* epoch_marker_ringbuf_last: Integer field containing the index of the
*            last entry in the ring buffer.
*
* epoch_marker_ringbuf_size: Integer field containing the number of entries
*            in the ring buffer.
*
* epoch_markers:  Array of instances of H5C_cache_entry_t of length
*            H5C__MAX_EPOCH_MARKERS.  The entries are used as markers
*            in the LRU list to identify cache entries that haven't
*            been accessed for some (small) specified number of
*            epochs.  These entries (if any) can then be evicted and
*            the cache size reduced -- ideally without evicting any
*            of the current working set.  Needless to say, the epoch
*            length and the number of epochs before an unused entry
*            must be chosen so that all, or almost all, the working
*            set will be accessed before the limit.
*
```

```
*            Epoch markers only appear in the LRU list, never in
*            the index or slist.  While they are of type
*            H5C__EPOCH_MARKER_TYPE, and have associated class
*            functions, these functions should never be called.
*
*            The addr fields of these instances of H5C_cache_entry_t
*            are set to the index of the instance in the epoch_markers
*            array, the size is set to 0, and the type field points
*            to the constant structure epoch_marker_class defined
*            in H5C.c.  The next and prev fields are used as usual
*            to link the entry into the LRU list.
*
*            All other fields are unused.
*
*
* Cache hit rate collection fields:
*
* We supply the current cache hit rate on request, so we must keep a
* simple cache hit rate computation regardless of whether statistics
* collection is enabled.  The following fields support this capability.
*
* cache_hits: Number of cache hits since the last time the cache hit
*      rate statistics were reset.  Note that when automatic cache
*      re-sizing is enabled, this field will be reset every automatic
*      resize epoch.
*
* cache_accesses: Number of times the cache has been accessed while
*      since the last since the last time the cache hit rate statistics
*      were reset.  Note that when automatic cache re-sizing is enabled,
*      this field will be reset every automatic resize epoch.
*
*
* Statistics collection fields:
*
* When enabled, these fields are used to collect statistics as described
* below.  The first set are collected only when H5C_COLLECT_CACHE_STATS
* is true.
*
* hits:        Array of int64 of length H5C__MAX_NUM_TYPE_IDS + 1.  The cells
*            are used to record the number of times an entry with type id
*            equal to the array index has been in cache when requested in
*            the current epoch.
*
* misses:      Array of int64 of length H5C__MAX_NUM_TYPE_IDS + 1.  The cells
*            are used to record the number of times an entry with type id
*            equal to the array index has not been in cache when
*            requested in the current epoch.
*
* write_protects:  Array of int64 of length H5C__MAX_NUM_TYPE_IDS + 1.  The
*            cells are used to record the number of times an entry with
*            type id equal to the array index has been write protected
*            in the current epoch.
*
*            Observe that (hits + misses) = (write_protects + read_protects).
*
* read_protects: Array of int64 of length H5C__MAX_NUM_TYPE_IDS + 1.  The
*            cells are used to record the number of times an entry with
*            type id equal to the array index has been read protected in
*            the current epoch.
```

```
 *
 *              Observe that (hits + misses) = (write_protects + read_protects).
 *
 * max_read_protects:  Array of int32 of length H5C__MAX_NUM_TYPE_IDS + 1.
 *              The cells are used to maximum number of simultaneous read
 *              protects on any entry with type id equal to the array index
 *              in the current epoch.
 *
 * insertions:  Array of int64 of length H5C__MAX_NUM_TYPE_IDS + 1.  The cells
 *              are used to record the number of times an entry with type
 *              id equal to the array index has been inserted into the
 *              cache in the current epoch.
 *
 * pinned_insertions:  Array of int64 of length H5C__MAX_NUM_TYPE_IDS + 1.
 *              The cells are used to record the number of times an entry
 *              with type id equal to the array index has been inserted
 *              pinned into the cache in the current epoch.
 *
 * clears:      Array of int64 of length H5C__MAX_NUM_TYPE_IDS + 1.  The cells
 *              are used to record the number of times an entry with type
 *              id equal to the array index has been cleared in the current
 *              epoch.
 *
 * flushes:     Array of int64 of length H5C__MAX_NUM_TYPE_IDS + 1.  The cells
 *              are used to record the number of times an entry with type id
 *              equal to the array index has been written to disk in the
 *              current epoch.
 *
 * evictions:   Array of int64 of length H5C__MAX_NUM_TYPE_IDS + 1.  The cells
 *              are used to record the number of times an entry with type id
 *              equal to the array index has been evicted from the cache in
 *              the current epoch.
 *
 * moves:       Array of int64 of length H5C__MAX_NUM_TYPE_IDS + 1.  The cells
 *              are used to record the number of times an entry with type
 *              id equal to the array index has been moved in the current
 *              epoch.
 *
 * entry_flush_moves: Array of int64 of length H5C__MAX_NUM_TYPE_IDS + 1.
 *              The cells are used to record the number of times an entry
 *              with type id equal to the array index has been moved
 *              during its pre-serialize callback in the current epoch.
 *
 * cache_flush_moves: Array of int64 of length H5C__MAX_NUM_TYPE_IDS + 1.
 *              The cells are used to record the number of times an entry
 *              with type id equal to the array index has been moved
 *              during a cache flush in the current epoch.
 *
 * pins:        Array of int64 of length H5C__MAX_NUM_TYPE_IDS + 1.  The cells
 *              are used to record the number of times an entry with type
 *              id equal to the array index has been pinned in the current
 *              epoch.
 *
 * unpins:      Array of int64 of length H5C__MAX_NUM_TYPE_IDS + 1.  The cells
 *              are used to record the number of times an entry with type
 *              id equal to the array index has been unpinned in the current
 *              epoch.
 *
 * dirty_pins:  Array of int64 of length H5C__MAX_NUM_TYPE_IDS + 1.  The cells
```

```
*             are used to record the number of times an entry with type
*             id equal to the array index has been marked dirty while pinned
*             in the current epoch.
*
* pinned_flushes:  Array of int64 of length H5C__MAX_NUM_TYPE_IDS + 1.  The
*             cells are used to record the number of times an  entry
*             with type id equal to the array index has been flushed while
*             pinned in the current epoch.
*
* pinned_clears:  Array of int64 of length H5C__MAX_NUM_TYPE_IDS + 1.  The
*             cells are used to record the number of times an  entry
*             with type id equal to the array index has been cleared while
*             pinned in the current epoch.
*
* size_increases:  Array of int64 of length H5C__MAX_NUM_TYPE_IDS + 1.
*             The cells are used to record the number of times an entry
*             with type id equal to the array index has increased in
*             size in the current epoch.
*
* size_decreases:  Array of int64 of length H5C__MAX_NUM_TYPE_IDS + 1.
*             The cells are used to record the number of times an entry
*             with type id equal to the array index has decreased in
*             size in the current epoch.
*
* entry_flush_size_changes:  Array of int64 of length
*             H5C__MAX_NUM_TYPE_IDS + 1.  The cells are used to record
*             the number of times an entry with type id equal to the
*             array index has changed size while in its pre-serialize
*             callback.
*
* cache_flush_size_changes:  Array of int64 of length
*             H5C__MAX_NUM_TYPE_IDS + 1.  The cells are used to record
*             the number of times an entry with type id equal to the
*             array index has changed size during a cache flush
*
* total_ht_insertions: Number of times entries have been inserted into the
*             hash table in the current epoch.
*
* total_ht_deletions: Number of times entries have been deleted from the
*             hash table in the current epoch.
*
* successful_ht_searches: int64 containing the total number of successful
*             searches of the hash table in the current epoch.
*
* total_successful_ht_search_depth: int64 containing the total number of
*             entries other than the targets examined in successful
*             searches of the hash table in the current epoch.
*
* failed_ht_searches: int64 containing the total number of unsuccessful
*             searches of the hash table in the current epoch.
*
* total_failed_ht_search_depth: int64 containing the total number of
*             entries examined in unsuccessful searches of the hash
*             table in the current epoch.
*
* max_index_len:  Largest value attained by the index_len field in the
*             current epoch.
*
* max_index_size:  Largest value attained by the index_size field in the
*            current epoch.
*
* max_clean_index_size: Largest value attained by the clean_index_size field
*            in the current epoch.
*
* max_dirty_index_size: Largest value attained by the dirty_index_size field
*            in the current epoch.
*
* max_slist_len:  Largest value attained by the slist_len field in the
*            current epoch.
*
* max_slist_size:  Largest value attained by the slist_size field in the
*            current epoch.
*
* max_pl_len:  Largest value attained by the pl_len field in the
*            current epoch.
*
* max_pl_size: Largest value attained by the pl_size field in the
*            current epoch.
*
* max_pel_len: Largest value attained by the pel_len field in the
*            current epoch.
*
* max_pel_size: Largest value attained by the pel_size field in the
*            current epoch.
*
* calls_to_msic: Total number of calls to H5C_make_space_in_cache
*
* total_entries_skipped_in_msic: Number of clean entries skipped while
*            enforcing the min_clean_fraction in H5C_make_space_in_cache().
*
* total_entries_scanned_in_msic: Number of clean entries skipped while
*            enforcing the min_clean_fraction in H5C_make_space_in_cache().
*
* max_entries_skipped_in_msic: Maximum number of clean entries skipped
*            in any one call to H5C_make_space_in_cache().
*
* max_entries_scanned_in_msic: Maximum number of entries scanned over
*            in any one call to H5C_make_space_in_cache().
*
* entries_scanned_to_make_space: Number of entries scanned only when looking
*            for entries to evict in order to make space in cache.

* The remaining stats are collected only when both H5C_COLLECT_CACHE_STATS
* and H5C_COLLECT_CACHE_ENTRY_STATS are true.
*
* max_accesses: Array of int32 of length H5C__MAX_NUM_TYPE_IDS + 1.  The cells
*            are used to record the maximum number of times any single
*            entry with type id equal to the array index has been
*            accessed in the current epoch.
*
* min_accesses: Array of int32 of length H5C__MAX_NUM_TYPE_IDS + 1.  The cells
*            are used to record the minimum number of times any single
*            entry with type id equal to the array index has been
*            accessed in the current epoch.
*
* max_clears:  Array of int32 of length H5C__MAX_NUM_TYPE_IDS + 1.  The cells
*            are used to record the maximum number of times any single
*            entry with type id equal to the array index has been cleared
```

```
 *              in the current epoch.
 *
 * max_flushes: Array of int32 of length H5C__MAX_NUM_TYPE_IDS + 1.  The cells
 *              are used to record the maximum number of times any single
 *              entry with type id equal to the array index has been
 *              flushed in the current epoch.
 *
 * max_size:    Array of size_t of length H5C__MAX_NUM_TYPE_IDS + 1.  The cells
 *              are used to record the maximum size of any single entry
 *              with type id equal to the array index that has resided in
 *              the cache in the current epoch.
 *
 * max_pins:    Array of size_t of length H5C__MAX_NUM_TYPE_IDS + 1.  The cells
 *              are used to record the maximum number of times that any single
 *              entry with type id equal to the array index that has been
 *              marked as pinned in the cache in the current epoch.
 *
 *
 * Fields supporting testing:
 *
 * prefix       Array of char used to prefix debugging output.  The
 *              field is intended to allow marking of output of with
 *              the processes mpi rank.
 *
 * get_entry_ptr_from_addr_counter: Counter used to track the number of
 *              times the H5C_get_entry_ptr_from_addr() function has been
 *              called successfully.  This field is only defined when
 *              NDEBUG is not #defined.
 *
 ****************************************************************************/

#define H5C__HASH_TABLE_LEN     (64 * 1024) /* must be a power of 2 */

#define H5C__H5C_T_MAGIC        0x005CAC0E
#define H5C__MAX_NUM_TYPE_IDS   28
#define H5C__PREFIX_LEN         32

struct H5C_t
{
    uint32_t                    magic;

    hbool_t                     flush_in_progress;

    FILE *                      trace_file_ptr;

    void *                      aux_ptr;

    int32_t                     max_type_id;
    const char *                (* type_name_table_ptr);

    size_t                      max_cache_size;
    size_t                      min_clean_size;

    H5C_write_permitted_func_t  check_write_permitted;
    hbool_t                     write_permitted;

    H5C_log_flush_func_t        log_flush;

    hbool_t                     evictions_enabled;
    int32_t                     index_len;
    size_t                      index_size;
    size_t                      clean_index_size;
    size_t                      dirty_index_size;
    H5C_cache_entry_t *         (index[H5C__HASH_TABLE_LEN]);

    hbool_t                     ignore_tags;

    hbool_t                     slist_changed;
    hbool_t                     slist_change_in_pre_serialize;
    hbool_t                     slist_change_in_serialize;
    int32_t                     slist_len;
    size_t                      slist_size;
    H5SL_t *                    slist_ptr;
    int32_t                     num_last_entries;
#if H5C_DO_SANITY_CHECKS
    int64_t                     slist_len_increase;
    int64_t                     slist_size_increase;
#endif /* H5C_DO_SANITY_CHECKS */

    int32_t                     pl_len;
    size_t                      pl_size;
    H5C_cache_entry_t *         pl_head_ptr;
    H5C_cache_entry_t *         pl_tail_ptr;

    int32_t                     pel_len;
    size_t                      pel_size;
    H5C_cache_entry_t *         pel_head_ptr;
    H5C_cache_entry_t *         pel_tail_ptr;

    int32_t                     LRU_list_len;
    size_t                      LRU_list_size;
    H5C_cache_entry_t *         LRU_head_ptr;
    H5C_cache_entry_t *         LRU_tail_ptr;

    int32_t                     cLRU_list_len;
    size_t                      cLRU_list_size;
    H5C_cache_entry_t *         cLRU_head_ptr;
    H5C_cache_entry_t *         cLRU_tail_ptr;

    int32_t                     dLRU_list_len;
    size_t                      dLRU_list_size;
    H5C_cache_entry_t *         dLRU_head_ptr;
    H5C_cache_entry_t *         dLRU_tail_ptr;

    hbool_t                     size_increase_possible;
    hbool_t                     flash_size_increase_possible;
    size_t                      flash_size_increase_threshold;
    hbool_t                     size_decrease_possible;
    hbool_t                     resize_enabled;
    hbool_t                     cache_full;
    hbool_t                     size_decreased;
    H5C_auto_size_ctl_t         resize_ctl;

    int32_t                     epoch_markers_active;
    hbool_t                     epoch_marker_active[H5C__MAX_EPOCH_MARKERS];
    int32_t                     epoch_marker_ringbuf[H5C__MAX_EPOCH_MARKERS+1];
    int32_t                     epoch_marker_ringbuf_first;
    int32_t                 epoch_marker_ringbuf_last;
    int32_t                 epoch_marker_ringbuf_size;
    H5C_cache_entry_t           epoch_markers[H5C__MAX_EPOCH_MARKERS];

    int64_t                 cache_hits;
    int64_t                 cache_accesses;

#if H5C_COLLECT_CACHE_STATS

    /* stats fields */
    int64_t                 hits[H5C__MAX_NUM_TYPE_IDS + 1];
    int64_t                 misses[H5C__MAX_NUM_TYPE_IDS + 1];
    int64_t                 write_protects[H5C__MAX_NUM_TYPE_IDS + 1];
    int64_t                 read_protects[H5C__MAX_NUM_TYPE_IDS + 1];
    int32_t                 max_read_protects[H5C__MAX_NUM_TYPE_IDS + 1];
    int64_t                 insertions[H5C__MAX_NUM_TYPE_IDS + 1];
    int64_t                 pinned_insertions[H5C__MAX_NUM_TYPE_IDS + 1];
    int64_t                 clears[H5C__MAX_NUM_TYPE_IDS + 1];
    int64_t                 flushes[H5C__MAX_NUM_TYPE_IDS + 1];
    int64_t                 evictions[H5C__MAX_NUM_TYPE_IDS + 1];
    int64_t                 moves[H5C__MAX_NUM_TYPE_IDS + 1];
    int64_t                 entry_flush_moves[H5C__MAX_NUM_TYPE_IDS + 1];
    int64_t                 cache_flush_moves[H5C__MAX_NUM_TYPE_IDS + 1];
    int64_t                 pins[H5C__MAX_NUM_TYPE_IDS + 1];
    int64_t                 unpins[H5C__MAX_NUM_TYPE_IDS + 1];
    int64_t                 dirty_pins[H5C__MAX_NUM_TYPE_IDS + 1];
    int64_t                 pinned_flushes[H5C__MAX_NUM_TYPE_IDS + 1];
    int64_t                 pinned_clears[H5C__MAX_NUM_TYPE_IDS + 1];
    int64_t                 size_increases[H5C__MAX_NUM_TYPE_IDS + 1];
    int64_t                 size_decreases[H5C__MAX_NUM_TYPE_IDS + 1];
    int64_t                 entry_flush_size_changes[H5C__MAX_NUM_TYPE_IDS + 1];
    int64_t                 cache_flush_size_changes[H5C__MAX_NUM_TYPE_IDS + 1];

    int64_t                 total_ht_insertions;
    int64_t                 total_ht_deletions;
    int64_t                 successful_ht_searches;
    int64_t                 total_successful_ht_search_depth;
    int64_t                 failed_ht_searches;
    int64_t                 total_failed_ht_search_depth;

    int32_t                 max_index_len;
    size_t                  max_index_size;
    size_t                  max_clean_index_size;
    size_t                  max_dirty_index_size;

    int32_t                 max_slist_len;
    size_t                  max_slist_size;

    int32_t                 max_pl_len;
    size_t                  max_pl_size;

    int32_t                 max_pel_len;
    size_t                  max_pel_size;

    int64_t                 calls_to_msic;
    int64_t                 total_entries_skipped_in_msic;
    int64_t                 total_entries_scanned_in_msic;
    int32_t                 max_entries_skipped_in_msic;
    int32_t                 max_entries_scanned_in_msic;
    int64_t                   entries_scanned_to_make_space;

#if H5C_COLLECT_CACHE_ENTRY_STATS

    int32_t                   max_accesses[H5C__MAX_NUM_TYPE_IDS + 1];
    int32_t                   min_accesses[H5C__MAX_NUM_TYPE_IDS + 1];
    int32_t                   max_clears[H5C__MAX_NUM_TYPE_IDS + 1];
    int32_t                   max_flushes[H5C__MAX_NUM_TYPE_IDS + 1];
    size_t                    max_size[H5C__MAX_NUM_TYPE_IDS + 1];
    int32_t                   max_pins[H5C__MAX_NUM_TYPE_IDS + 1];

#endif /* H5C_COLLECT_CACHE_ENTRY_STATS */

#endif /* H5C_COLLECT_CACHE_STATS */

    char                      prefix[H5C__PREFIX_LEN];

#ifndef NDEBUG

    int64_t                   get_entry_ptr_from_addr_counter;

#endif /* NDEBUG */
};
```

Appendix 2: H5C_cache_entry_t

The definition of struct H5C_cache_entry_t is reproduced below, along with its header comment. Please note the documentation of each field in the structure. Developers are encouraged to update and correct this header comment in H5Cprivate.h as appropriate. While the following was correct as of the preparation of this document, the version in H5Cprivate.h is the master copy, and should be referred to for up to date information.

```
/****************************************************************************
 *
 * structure H5C_cache_entry_t
 *
 * Instances of the H5C_cache_entry_t structure are used to store cache
 * entries in a hash table and sometimes in a skip list.
 * See H5SL.c for the particulars of the skip list.
 *
 * In typical application, this structure is the first field in a
 * structure to be cached.  For historical reasons, the external module
 * is responsible for managing the is_dirty field (this is no longer
 * completely true.  See the comment on the is_dirty field for details).
 * All other fields are managed by the cache.
 *
 * The fields of this structure are discussed individually below:
 *
 *                                    JRM - 4/26/04
 *
 * magic:       Unsigned 32 bit integer that must always be set to
 *              H5C__H5C_CACHE_ENTRY_T_MAGIC when the entry is valid.
 *              The field must be set to H5C__H5C_CACHE_ENTRY_T_BAD_MAGIC
 *              just before the entry is freed.
 *
 *              This is necessary, as the LRU list can be changed out
 *              from under H5C_make_space_in_cache() by the serialize
 *              callback which may change the size of an existing entry,
 *              and/or load a new entry while serializing the target entry.
 *
```

```
*            This in turn can cause a recursive call to
*            H5C_make_space_in_cache() which may either flush or evict
*            the next entry that the first invocation of that function
*            was about to examine.
*
*            The magic field allows H5C_make_space_in_cache() to
*            detect this case, and re-start its scan from the bottom
*            of the LRU when this situation occurs.
*
* cache_ptr:   Pointer to the cache that this entry is contained within.
*
* addr:        Base address of the cache entry on disk.
*
* size:        Length of the cache entry on disk.  Note that unlike normal
*            caches, the entries in this cache are of variable length.
*            The entries should never overlap, and when we do writebacks,
*            we will want to writeback adjacent entries where possible.
*
*            NB: At present, entries need not be contiguous on disk.  Until
*                we fix this, we can't do much with writing back adjacent
*                entries.
*
*            Update: This has now been changed -- all metadata cache
*            entries must now be associated with a single contiguous
*            block of memory on disk.  The image of this block (i.e.
*            the on disk image) is stored in *image_ptr (discussed below).
*
* image_ptr:   Pointer to void.  When not NULL, this field points to a
*            dynamically allocated block of size bytes in which the
*            on disk image of the metadata cache entry is stored.
*
*            If the entry is dirty, the pre-serialize and serialize
*            callbacks must be used to update this image before it is
*            written to disk
*
* image_up_to_date:  Boolean flag that is set to TRUE when *image_ptr
*            is up to date, and set to false when the entry is dirtied.
*
* type:        Pointer to the instance of H5C_class_t containing pointers
*            to the methods for cache entries of the current type.  This
*            field should be NULL when the instance of H5C_cache_entry_t
*            is not in use.
*
*            The name is not particularly descriptive, but is retained
*            to avoid changes in existing code.
*
* is_dirty:    Boolean flag indicating whether the contents of the cache
*            entry has been modified since the last time it was written
*            to disk.
*
*            NOTE: For historical reasons, this field is not maintained
*                  by the cache.  Instead, the module using the cache
*                  sets this flag when it modifies the entry, and the
*                  flush and clear functions supplied by that module
*                  reset the dirty when appropriate.
*
*                  This is a bit quirky, so we may want to change this
*                  someday.  However it will require a change in the
*                  cache interface.
```

```
*
*           Update: Management of the is_dirty field has been largely
*                moved into the cache.  The only remaining exceptions
*                are the flush and clear functions supplied by the
*                modules using the cache.  These still clear the
*                is_dirty field as before.  -- JRM 7/5/05
*
*           Update: Management of the is_dirty field is now entirely
*                in the cache.             -- JRM 7/5/07
*
* dirtied:    Boolean flag used to indicate that the entry has been
*           dirtied while protected.
*
*           This field is set to FALSE in the protect call, and may
*           be set to TRUE by the H5C_mark_entry_dirty() call at any
*           time prior to the unprotect call.
*
*           The H5C_mark_entry_dirty() call exists as a convenience
*           function for the fractal heap code which may not know if
*           an entry is protected or pinned, but knows that is either
*           protected or pinned.  The dirtied field was added as in
*           the parallel case, it is necessary to know whether a
*           protected entry is dirty prior to the protect call.
*
* is_protected: Boolean flag indicating whether this entry is protected
*           (or locked, to use more conventional terms).  When it is
*           protected, the entry cannot be flushed or accessed until
*           it is unprotected (or unlocked -- again to use more
*           conventional terms).
*
*           Note that protected entries are removed from the LRU lists
*           and inserted on the protected list.
*
* is_read_only: Boolean flag that is only meaningful if is_protected is
*           TRUE.  In this circumstance, it indicates whether the
*           entry has been protected read only, or read/write.
*
*           If the entry has been protected read only (i.e. is_protected
*           and is_read_only are both TRUE), we allow the entry to be
*           protected more than once.
*
*           In this case, the number of readers is maintained in the
*           ro_ref_count field (see below), and unprotect calls simply
*           decrement that field until it drops to zero, at which point
*           the entry is actually unprotected.
*
* ro_ref_count: Integer field used to maintain a count of the number of
*           outstanding read only protects on this entry.  This field
*           must be zero whenever either is_protected or is_read_only
*           are TRUE.
*
* is_pinned:   Boolean flag indicating whether the entry has been pinned
*           in the cache.
*
*           For very hot entries, the protect / unprotect overhead
*           can become excessive.  Thus the cache has been extended
*           to allow an entry to be "pinned" in the cache.
*
*           Pinning an entry in the cache has several implications:
*
*              1) A pinned entry cannot be evicted.  Thus unprotected
*                 pinned entries must be stored in the pinned entry
*                 list, instead of being managed by the replacement
*                 policy code (LRU at present).
*
*              2) A pinned entry can be accessed or modified at any time.
*                 This places an extra burden on the pre-serialize and
*                 serialize callbacks, which must ensure that a pinned
*                 entry is consistant and ready to write to disk before
*                 generating an image.
*
*              3) A pinned entry can be marked as dirty (and possibly
*                 change size) while it is unprotected.
*
*              4) The flush-destroy code must allow pinned entries to
*                 be unpinned (and possibly unprotected) during the
*                 flush.
*
*                                      JRM -- 3/16/06
*
* in_slist:    Boolean flag indicating whether the entry is in the skip list
*              As a general rule, entries are placed in the list when they
*              are marked dirty.  However they may remain in the list after
*              being flushed.
*
*              Update: Dirty entries are now removed from the skip list
*                      when they are flushed.
*
* flush_marker:  Boolean flag indicating that the entry is to be flushed
*              the next time H5C_flush_cache() is called with the
*              H5C__FLUSH_MARKED_ENTRIES_FLAG.  The flag is reset when
*              the entry is flushed for whatever reason.
*
* flush_me_last:  Boolean flag indicating that this entry should not be
*              flushed from the cache until all other entries without
*              the flush_me_last flag set have been flushed.
*
* flush_me_collectively:  Boolean flag indicating that this entry needs
*              to be flushed collectively when in a parallel situation.
*
*              Note:
*
*              At this time, the flush_me_last and flush_me_collectively
*              flags will only be applied to one entry, the superblock,
*              and the code utilizing these flags is protected with HDasserts
*              to enforce this. This restraint can certainly be relaxed in
*              the future if the the need for multiple entries getting flushed
*              last or collectively arises, though the code allowing for that
*              will need to be expanded and tested appropriately if that
*              functionality is desired.
*
*              Update: There are now two possible last entries
*                      (superblock and file driver info message).  This
*                      number will probably increase as we add superblock
*                      messages.   JRM -- 11/18/14
*
* clear_on_unprotect:  Boolean flag used only in PHDF5.  When H5C is used
*              to implement the metadata cache In the parallel case, only
```

```
*		the cache with mpi rank 0 is allowed to actually write to
*		file -- all other caches must retain dirty entries until they
*		are advised that the entry is clean.
*
*		This flag is used in the case that such an advisory is
*		received when the entry is protected.  If it is set when an
*		entry is unprotected, and the dirtied flag is not set in
*		the unprotect, the entry's is_dirty flag is reset by flushing
*		it with the H5C__FLUSH_CLEAR_ONLY_FLAG.
*
* flush_immediately:  Boolean flag used only in Phdf5 -- and then only
*		for H5AC_METADATA_WRITE_STRATEGY__DISTRIBUTED.
*
*		When a destributed metadata write is triggered at a
*		sync point, this field is used to mark entries that
*		must be flushed before leaving the sync point.  At all
*		other times, this field should be set to FALSE.
*
* flush_in_progress:  Boolean flag that is set to true iff the entry
*		is in the process of being flushed.  This allows the cache
*		to detect when a call is the result of a flush callback.
*
* destroy_in_progress:  Boolean flag that is set to true iff the entry
*		is in the process of being flushed and destroyed.
*
*
* Fields supporting the 'flush dependency' feature:
*
* Entries in the cache may have a 'flush dependency' on another entry in the
* cache.  A flush dependency requires that all dirty child entries be flushed
* to the file before a dirty parent entry (of those child entries) can be
* flushed to the file.  This can be used by cache clients to create data
* structures that allow Single-Writer/Multiple-Reader (SWMR) access for the
* data structure.
*
* The leaf child entry will have a "height" of 0, with any parent entries
* having a height of 1 greater than the maximum height of any of their child
* entries (flush dependencies are allowed to create asymmetric trees of
* relationships).
*
* flush_dep_parent:    Pointer to the parent entry for an entry in a flush
*		dependency relationship.
*
* child_flush_dep_height_rc:   An array of reference counts for child entries,
*		where the number of children of each height is tracked.
*
* flush_dep_height:    The height of the entry, which is one greater than the
*		maximum height of any of its child entries..
*
* pinned_from_client:  Whether the entry was pinned by an explicit pin request
*		from a cache client.
*
* pinned_from_cache:   Whether the entry was pinned implicitly as a
*		request of being a parent entry in a flush dependency
*		relationship.
*
*
* Fields supporting the hash table:
*
```

```
* Fields in the cache are indexed by a more or less conventional hash table.
* If there are multiple entries in any hash bin, they are stored in a doubly
* linked list.
*
* ht_next:     Next pointer used by the hash table to store multiple
*              entries in a single hash bin.  This field points to the
*              next entry in the doubly linked list of entries in the
*              hash bin, or NULL if there is no next entry.
*
* ht_prev:     Prev pointer used by the hash table to store multiple
*              entries in a single hash bin.  This field points to the
*              previous entry in the doubly linked list of entries in
*              the hash bin, or NULL if there is no previuos entry.
*
*
* Fields supporting replacement policies:
*
* The cache must have a replacement policy, and it will usually be
* necessary for this structure to contain fields supporting that policy.
*
* While there has been interest in several replacement policies for
* this cache, the initial development schedule is tight.  Thus I have
* elected to support only a modified LRU policy for the first cut.
*
* When additional replacement policies are added, the fields in this
* section will be used in different ways or not at all.  Thus the
* documentation of these fields is repeated for each replacement policy.
*
* Modified LRU:
*
* When operating in parallel mode, we must ensure that a read does not
* cause a write.  If it does, the process will hang, as the write will
* be collective and the other processes will not know to participate.
*
* To deal with this issue, I have modified the usual LRU policy by adding
* clean and dirty LRU lists to the usual LRU list.  When reading in
* parallel mode, we evict from the clean LRU list only.  This implies
* that we must try to ensure that the clean LRU list is reasonably well
* stocked.  See the comments on H5C_t in H5Cpkg.h for more details.
*
* Note that even if we start with a completely clean cache, a sequence
* of protects without unprotects can empty the clean LRU list.  In this
* case, the cache must grow temporarily.  At the next write, we will
* attempt to evict enough entries to get the cache down to its nominal
* maximum size.
*
* The use of the replacement policy fields under the Modified LRU policy
* is discussed below:
*
* next:        Next pointer in either the LRU, the protected list, or
*              the pinned list depending on the current values of
*              is_protected and is_pinned.  If there is no next entry
*              on the list, this field should be set to NULL.
*
* prev:        Prev pointer in either the LRU, the protected list,
*              or the pinned list depending on the current values of
*              is_protected and is_pinned.  If there is no previous
*              entry on the list, this field should be set to NULL.
*
```

```
 * aux_next:    Next pointer on either the clean or dirty LRU lists.
 *             This entry should be NULL when either is_protected or
 *             is_pinned is true.
 *
 *             When is_protected and is_pinned are false, and is_dirty is
 *             true, it should point to the next item on the dirty LRU
 *             list.
 *
 *             When is_protected and is_pinned are false, and is_dirty is
 *             false, it should point to the next item on the clean LRU
 *             list.  In either case, when there is no next item, it
 *             should be NULL.
 *
 * aux_prev:    Previous pointer on either the clean or dirty LRU lists.
 *             This entry should be NULL when either is_protected or
 *             is_pinned is true.
 *
 *             When is_protected and is_pinned are false, and is_dirty is
 *             true, it should point to the previous item on the dirty
 *             LRU list.
 *
 *             When is_protected and is_pinned are false, and is_dirty
 *             is false, it should point to the previous item on the
 *             clean LRU list.
 *
 *             In either case, when there is no previous item, it should
 *             be NULL.
 *
 * Cache entry stats collection fields:
 *
 * These fields should only be compiled in when both H5C_COLLECT_CACHE_STATS
 * and H5C_COLLECT_CACHE_ENTRY_STATS are true.  When present, they allow
 * collection of statistics on individual cache entries.
 *
 * accesses:    int32_t containing the number of times this cache entry has
 *             been referenced in its lifetime.
 *
 * clears:      int32_t containing the number of times this cache entry has
 *             been cleared in its life time.
 *
 * flushes:     int32_t containing the number of times this cache entry has
 *             been flushed to file in its life time.
 *
 * pins:        int32_t containing the number of times this cache entry has
 *             been pinned in cache in its life time.
 *
 ****************************************************************************/

#ifndef NDEBUG
#define H5C__H5C_CACHE_ENTRY_T_MAGIC           0x005CAC0A
#define H5C__H5C_CACHE_ENTRY_T_BAD_MAGIC       0xDeadBeef
#endif /* NDEBUG */

typedef struct H5C_cache_entry_t
{
#ifndef NDEBUG
    uint32_t                    magic;
#endif /* NDEBUG */
    H5C_t *                     cache_ptr;
    haddr_t                   addr;
    size_t                    size;
    void *                    image_ptr;
    hbool_t                   image_up_to_date;
    const H5C_class_t *       type;
    haddr_t                   tag;
    hbool_t                   is_dirty;
    hbool_t                   dirtied;
    hbool_t                   is_protected;
    hbool_t                   is_read_only;
    int                       ro_ref_count;
    hbool_t                   is_pinned;
    hbool_t                   in_slist;
    hbool_t                   flush_marker;
    hbool_t                   flush_me_last;
#ifdef H5_HAVE_PARALLEL
    hbool_t                   flush_me_collectively;
    hbool_t                   clear_on_unprotect;
    hbool_t                   flush_immediately;
#endif /* H5_HAVE_PARALLEL */
    hbool_t                   flush_in_progress;
    hbool_t                   destroy_in_progress;

    /* fields supporting the 'flush dependency' feature: */

    struct H5C_cache_entry_t *  flush_dep_parent;
    uint64_t                  child_flush_dep_height_rc[H5C__NUM_FLUSH_DEP_HEIGHTS];
    unsigned                  flush_dep_height;
    hbool_t                   pinned_from_client;
    hbool_t                   pinned_from_cache;

    /* fields supporting the hash table: */

    struct H5C_cache_entry_t *  ht_next;
    struct H5C_cache_entry_t *  ht_prev;

    /* fields supporting replacement policies: */

    struct H5C_cache_entry_t *  next;
    struct H5C_cache_entry_t *  prev;
    struct H5C_cache_entry_t *  aux_next;
    struct H5C_cache_entry_t *  aux_prev;

#if H5C_COLLECT_CACHE_ENTRY_STATS

    /* cache entry stats fields */

    int32_t                   accesses;
    int32_t                   clears;
    int32_t                   flushes;
    int32_t                   pins;

#endif /* H5C_COLLECT_CACHE_ENTRY_STATS */

} H5C_cache_entry_t;
```

Appendix 3. H5C_class_t and friends

The definition of struct H5C_class_t is reproduced below, along with its header comment and the associated #defines and function type definitions. Please note the documentation of each field in the structure. Developers are encouraged to update and correct this header comment in H5Cprivate.h as appropriate. While the following was correct as of the preparation of this document, the version in H5Cprivate.h is the master copy, and should be referred to for up to date information.

```
/****************************************************************************
 *
 * Struct H5C_class_t
 *
 * Instances of H5C_class_t are used to specify the callback functions
 * used by the metadata cache for each class of metadata cache entry.
 * The fields of the structure are discussed below:
 *
 * id:  Integer field containing the unique ID of the class of metadata
 *      cache entries.
 *
 * name: Pointer to a string containing the name of the class of metadata
 *      cache entries.
 *
 * mem_type:  Instance of H5FD_mem_t, that is used to supply the
 *      mem type passed into H5F_block_read().
 *
 * flags:  Flags indicating class-specific behavior.
 *
 *      Whoever created the flags field neglected to document the meanings
 *      of the flags he created.  Hence the following discussions of the
 *      H5C__CLASS_SPECULATIVE_LOAD_FLAG and the H5C__CLASS_COMPRESSED_FLAG
 *      should be viewed with suspicion, as the meanings are divined from
 *      the source code, and thus may be inaccurate.  Please correct any
 *      errors you find.
 *
 *      Possible flags are:
 *
 *      H5C__CLASS_NO_FLAGS_SET: No special processing.
 *
 *      H5C__CLASS_SPECULATIVE_LOAD_FLAG: This flag appears to be used
 *              only in H5C_load_entry().  When it is set, entries are
 *              permitted to change their sizes on the first attempt
 *              to load.
 *
 *              If the new size is larger than the old, the read buffer
 *              is reallocated to the new size, loaded from file, and the
 *              deserialize routine is called a second time on the
 *              new buffer.  The entry returned by the first call to
 *              the deserialize routine is discarded (via the free_icr
 *              call) after the new size is retrieved (via the image_len
 *              call).  Note that the new size is used as the size of the
 *              entry in the cache.
 *
 *              If the new size is smaller than the old, no new loads
 *              or desearializes are performed, but the new size becomes
 *              the size of the entry in the cache.
 *
 *              When this flag is set, an attempt to read past the
 *              end of file is pemitted.  In this case, if the size
 *              returned get_load_size callback would result in a
 *              read past the end of file, the size is trunkated to
 *              avoid this, and processing proceeds as normal.
 *
 *      H5C__CLASS_COMPRESSED_FLAG: This flags appears to be used
 *           only in H5C_load_entry().  It seems to mean that
 *           an entry can change its size on load, but that no
 *           retry of the load from file or deserialize call is
 *           needed.  This only makes sense if the new size is
 *           less than the old -- but there is no requirement for
 *           this in the code.  I have inserted an assert to test
 *           this.  So far it hasn't been triggered.
 *
 *           As with the H5C__CLASS_SPECULATIVE_LOAD_FLAG, the new
 *           size is used as the size of the entry in the cache.
 *
 *     The following flags may only appear in test code.
 *
 *     H5C__CLASS_NO_IO_FLAG:  This flag is intended only for use in test
 *           code.  When it is set, any attempt to load an entry of
 *           the type with this flag set will trigger an assertion
 *           failure, and any flush of an entry with this flag set
 *           will not result in any write to file.
 *
 *     H5C__CLASS_SKIP_READS: This flags is intended only for use in test
 *           code.  When it is set, reads on load will be skipped,
 *           and an uninitialize buffer will be passed to the
 *           deserialize function.
 *
 *     H5C__CLASS_SKIP_WRITES: This flags is intended only for use in test
 *           code.  When it is set, writes of buffers prepared by the
 *           serialize callback will be skipped.
 *
 * GET_LOAD_SIZE: Pointer to the 'get load size' function.
 *
 *     This function must be able to determine the size of the disk image of
 *     a metadata cache entry, given the 'udata' that will be passed to the
 *     'deserialize' callback.
 *
 *     Note that if either the H5C__CLASS_SPECULATIVE_LOAD_FLAG or
 *     the H5C__CLASS_COMPRESSED_FLAG is set, the disk image size
 *     returned by this callback is either a first guess (if the
 *     H5C__CLASS_SPECULATIVE_LOAD_FLAG is set) or an upper bound
 *     (if the H5C__CLASS_COMPRESSED_FLAG is set).  In all other cases,
 *     the value returned should be correct.
 *
 *     The typedef for the deserialize callback is as follows:
 *
 *        typedef herr_t (*H5C_get_load_size_func_t)(void *udata_ptr,
 *                                                size_t *image_len_ptr);
 *
 *     The parameters of the deserialize callback are as follows:
 *
 *     udata_ptr: Pointer to user data provided in the protect call, which
 *           will also be passed through to the deserialize callback.
 *
 *     image_len_ptr: Pointer to the location in which the length in bytes
 *           of the in file image to be deserialized is to be returned.
 *
 *           This value is used by the cache to determine the size of
 *           the disk image for the metadata, in order to read the disk
 *           image from the file.
 *
 *     Processing in the get_load_size function should proceed as follows:
  *
  *      If successful, the function will place the length of the on disk
  *      image associated with the in core representation provided in the
  *      thing parameter in *image_len_ptr, and then return SUCCEED.
  *
  *      On failure, the function must return FAIL and push error information
  *      onto the error stack with the error API routines, without modifying
  *      the value pointed to by the image_len_ptr.
  *
  *
  * DESERIALIZE: Pointer to the deserialize function.
  *
  *      This function must be able to read an on disk image of a metadata
  *      cache entry, allocate and load the equivalent in core representation,
  *      and return a pointer to that representation.
  *
  *      The typedef for the deserialize callback is as follows:
  *
  *         typedef void *(*H5C_deserialize_func_t)(const void * image_ptr,
  *                                            size_t len,
  *                                            void * udata_ptr,
  *                                            boolean * dirty_ptr);
  *
  *      The parameters of the deserialize callback are as follows:
  *
  *      image_ptr: Pointer to a buffer of length len containing the
  *              contents of the file starting at addr and continuing
  *              for len bytes.
  *
  *      len:    Length in bytes of the in file image to be deserialized.
  *
  *              This parameter is supplied mainly for sanity checking.
  *              Sanity checks should be performed when compiled in debug
  *              mode, but the parameter may be unused when compiled in
  *              production mode.
  *
  *      udata_ptr: Pointer to user data provided in the protect call, which
  *              must be passed through to the deserialize callback.
  *
  *      dirty_ptr:  Pointer to boolean which the deserialize function
  *              must use to mark the entry dirty if it has to modify
  *              the entry to clean up file corruption left over from
  *              an old bug in the HDF5 library.
  *
  *      Processing in the deserialize function should proceed as follows:
  *
  *      If the image contains valid data, and is of the correct length,
  *      the deserialize function must allocate space for an in core
  *      representation of that data, load the contents of the image into
  *      the space allocated for the in core representation, and return
  *      a pointer to the in core representation.  Observe that an
  *      instance of H5C_cache_entry_t must be the first item in this
  *      representation.  The cache will initialize it after the callback
  *      returns.
  *
  *      Note that the structure of the in core representation is otherwise
  *      up to the cache client.  All that is required is that the pointer
  *      returned be sufficient for the clients purposes when it is returned
  *      on a protect call.
  *
  *      If the deserialize function has to clean up file corruption
  *      left over from an old bug in the HDF5 library, it must set
  *      *dirty_ptr to TRUE.  If it doesn't, no action is needed as
  *      *dirty_ptr will be set to FALSE before the deserialize call.
  *
  *      If the operation fails for any reason (i.e. bad data in buffer, bad
  *      buffer length, malloc failure, etc.) the function must return NULL and
  *      push error information on the error stack with the error API routines.
  *
  *      Exceptions to the above:
  *
  *      If the H5C__CLASS_SPECULATIVE_LOAD_FLAG is set, the buffer supplied
  *      to the function need not be currect on the first invocation of the
  *      callback in any single attempt to load the entry.
  *
  *      In this case, if the buffer is larger than necessary, the function
  *      should load the entry as described above and not flag an error due
  *      to the oversized buffer.  The cache will correct its mis-apprehension
  *      of the entry size with a subsequent call to the image_len callback.
  *
  *      If the buffer is too small, and this is the first desrialize call
  *      in the entry load operation, the function should not flag an error.
  *      Instead, it must compute the correct size of the entry, allocate an
  *      in core representation and initialize it to the extent that an
  *      immediate call to the image len callback will return the correct
  *      image size.
  *
  *      In this case, when the deserialize callback returns, the cache will
  *      call the image length callback, realize that the supplied buffer was
  *      too small, discard the returned in core representation, allocate
  *      and load a new buffer of the correct size from file, and then call
  *      the deserialize callback again.
  *
  *      If the H5C__CLASS_COMPRESSED_FLAG is set, exceptions are as per the
  *      H5C__CLASS_SPECULATIVE_LOAD_FLAG, save that only oversized buffers
  *      are permitted.
  *
  *
  * IMAGE_LEN: Pointer to the image length callback.
  *
  *      In the best of all possible worlds, we would not have this callback.
  *      It exists to allow clients to change the size of the on disk image
  *      of an entry in the deserialize callback (see discussion above).
  *
  *      The typedef for the image_len callback is as follows:
  *
  *      typedef herr_t (*H5C_image_len_func_t)(void *thing,
  *                                             size_t *image_len_ptr);
  *
  *      The parameters of the image_len callback are as follows:
  *
  *      thing:  Pointer to the in core representation of the entry.
  *
  *      image_len_ptr: Pointer to size_t in which the callback will return
  *              the length of the on disk image of the cache entry.
  *
  *      Processing in the image_len function should proceed as follows:
  *
  *      If successful, the function will place the length of the on disk
  *      image associated with the in core representation provided in the
  *      thing parameter in *image_len_ptr, and then return SUCCEED.
  *
  *      On failure, the function must return FAIL and push error information
  *      onto the error stack with the error API routines, without modifying
  *      the value pointed to by the image_len_ptr.
  *
  *
  * PRE_SERIALIZE: Pointer to the pre-serialize callback.
  *
  *      The pre-serialize callback is invoked by the metadata cache before
  *      it needs a current on-disk image of the metadata entry for purposes
  *      either constructing a journal or flushing the entry to disk.
  *
  *      If the client needs to change the address or length of the entry on
  *      disk prior to flush, the pre-serialize callback is responsible for
  *      these actions, so that the actual serialize callback (described
  *      below) is only responsible for serializing the data structure, not
  *      moving it on disk or resizing it.
  *
  *      In addition, the client may use the pre-serialize callback to
  *      ensure that the entry is ready to be flushed -- in particular,
  *      if the entry contains references to other entries that are in
  *      temporary file space, the pre-serialize callback must move those
  *      entries into real file space so that the serialzed entry will
  *      contain no invalid data.
  *
  *      One would think that the base address and length of
  *      the length of the entry's image on disk would be well known.
  *      However, that need not be the case as fractal heap blocks can
  *      change size (and therefor possible location as well) on
  *      serialization if compression is enabled.  Similarly, it may
  *      be necessary to move entries from temporary to real file space.
  *
  *      The pre-serialize callback must report any such changes to the
  *      cache, which must then update its internal structures as needed.
  *
  *      The typedef for the pre-serialize callback is as follows:
  *
  *      typedef herr_t (*H5C_pre_serialize_func_t)(const H5F_t *f,
  *                                              hid_t dxpl_id,
  *                                              void * thing,
  *                                              haddr_t addr,
  *                                              size_t len,
  *                                              haddr_t * new_addr_ptr,
  *                                              size_t * new_len_ptr,
  *                                              unsigned * flags_ptr);
  *
  *      The parameters of the pre-serialize callback are as follows:
  *
  *      f:      File pointer -- needed if other metadata cache entries
  *              must be modified in the process of serializing the
  *              target entry.
  *
  *      dxpl_id: dxpl_id passed with the file pointer to the cache, and
  *              passed on to the callback.  Necessary as some callbacks
  *              revise the size and location of the target entry, or
  *              possibly other entries on pre-serialize.
  *
  *      thing:  Pointer to void containing the address of the in core
  *              representation of the target metadata cache entry.
  *              This is the same pointer returned by a protect of the
  *              addr and len given above.
  *
  *      addr:   Base address in file of the entry to be serialized.
  *
  *              This parameter is supplied mainly for sanity checking.
  *              Sanity checks should be performed when compiled in debug
  *              mode, but the parameter may be unused when compiled in
  *              production mode.
  *
  *      len:    Length in bytes of the in file image of the entry to be
  *              serialized.  Also the size the image passed to the
  *              serialize callback (discussed below) unless that value
  *              is altered by this function
  *
  *              This parameter is supplied mainly for sanity checking.
  *              Sanity checks should be performed when compiled in debug
  *              mode, but the parameter may be unused when compiled in
  *              production mode.
  *
  *      new_addr_ptr:  Pointer to haddr_t.  If the entry is moved by
  *              the serialize function, the new on disk base address must
  *              be stored in *new_addr_ptr, and the appropriate flag set
  *              in *flags_ptr.
  *
  *              If the entry is not moved by the serialize function,
  *              *new_addr_ptr is undefined on pre-serialize callback
  *              return.
  *
  *      new_len_ptr:  Pointer to size_t.  If the entry is resized by the
  *              serialize function, the new length of the on disk image
  *              must be stored in *new_len_ptr, and the appropriate flag set
  *              in *flags_ptr.
  *
  *              If the entry is not resized by the pre-serialize function,
  *              *new_len_ptr is undefined on pre-serialize callback
  *              return.
  *
  *      flags_ptr:  Pointer to an unsigned integer used to return flags
  *              indicating whether the resize function resized or moved
  *              the entry.  If the entry was neither resized or moved,
  *              the serialize function must set *flags_ptr to zero.
  *              H5C__SERIALIZE_RESIZED_FLAG and H5C__SERIALIZE_MOVED_FLAG
  *              must be set to indicate a resize and a move respectively.
  *
  *              If the H5C__SERIALIZE_RESIZED_FLAG is set, the new length
  *              must be stored in *new_len_ptr.
  *
  *              If the H5C__SERIALIZE_MOVED_FLAG flag is set, the
  *              new image base address must be stored in *new_addr_ptr.
  *
  *      Processing in the pre-serialize function should proceed as follows:
  *
  *      The pre-serialize function must examine the in core representation
  *      indicated by the thing parameter, if the pre-serialize function does
  *      not need to change the size or location of the on-disk image, it must
  *      set *flags_ptr to zero.
  *
  *      If the size of the on-disk image must be changed, the pre-serialize
  *      function must load the length of the new image into *new_len_ptr,
  *      and set the H5C__SERIALIZE_RESIZED_FLAG in *flags_ptr.
  *
  *      If the base address of the on disk image must be changed, the
  *      pre-serialize function must set *new_addr_ptr to the new base address,
  *      and set the H5C__SERIALIZE_MOVED_FLAG in *flags_ptr.
  *
  *      In addition, the pre-serialize callback may perform any other
  *      processing required before the entry is written to disk
  *
  *      If it is successful, the function must return SUCCEED.
  *
  *      If it fails for any reason, the function must return FAIL and
  *      push error information on the error stack with the error API
  *      routines.
  *
  *
  * SERIALIZE: Pointer to the serialize callback.
  *
  *      The serialize callback is invoked by the metadata cache whenever
  *      it needs a current on disk image of the metadata entry for purposes
  *      either constructing a journal or flushing the entry to disk.
  *
  *      At this point, the base address and length of the entry's image on
  *      disk must be well known and not change during the serialization
  *      process.
  *
  *      While any size and/or location changes must have been handled
  *      by a pre-serialize call, the client may elect to handle any other
  *      changes to the entry required to place it in correct form for
  *      writing to disk in this call.
  *
  *      The typedef for the serialize callback is as follows:
  *
  *      typedef herr_t (*H5C_serialize_func_t)(const H5F_t *f,
  *                                             void * image_ptr,
  *                                             size_t len,
  *                                             void * thing);
  *
  *      The parameters of the serialize callback are as follows:
  *
  *      f:      File pointer -- needed if other metadata cache entries
  *              must be modified in the process of serializing the
  *              target entry.
  *
  *      image_ptr: Pointer to a buffer of length len bytes into which a
  *              serialized image of the target metadata cache entry is
  *              to be written.
  *
  *              Note that this buffer will not in general be initialized
  *              to any particular value.  Thus the serialize function may
  *              not assume any initial value and must set each byte in
  *              the buffer.
  *
  *      len:    Length in bytes of the in file image of the entry to be
  *              serialized.  Also the size of *image_ptr (below).
```

```
*
*           This parameter is supplied mainly for sanity checking.
*           Sanity checks should be performed when compiled in debug
*           mode, but the parameter may be unused when compiled in
*           production mode.
*
*     thing:  Pointer to void containing the address of the in core
*           representation of the target metadata cache entry.
*           This is the same pointer returned by a protect of the
*           addr and len given above.
*
*     Processing in the serialize function should proceed as follows:
*
*     If there are any remaining changes to the entry required before
*     write to disk, they must be dealt with first.
*
*     The serialize function must then examine the in core
*     representation indicated by the thing parameter, and write a
*     serialized image of its contents into the provided buffer.
*
*     If it is successful, the function must return SUCCEED.
*
*     If it fails for any reason, the function must return FAIL and
*     push error information on the error stack with the error API
*     routines.
*
*
* NOTIFY: Pointer to the notify callback.
*
*     The notify callback is invoked by the metadata cache when a cache
*     action on an entry has taken/will take place and the client indicates
*     it wishes to be notified about the action.
*
*     The typedef for the notify callback is as follows:
*
*     typedef herr_t (*H5C_notify_func_t)(H5C_notify_action_t action,
*                                         void *thing);
*
*     The parameters of the notify callback are as follows:
*
*     action: An enum indicating the metadata cache action that has taken/
*           will take place.
*
*     thing:  Pointer to void containing the address of the in core
*           representation of the target metadata cache entry.  This
*           is the same pointer that would be returned by a protect
*           of the addr and len of the entry.
*
*     Processing in the notify function should proceed as follows:
*
*     The notify function may perform any action it would like, including
*     metadata cache calls.
*
*     If the function is successful, it must return SUCCEED.
*
*     If it fails for any reason, the function must return FAIL and
*     push error information on the error stack with the error API
*     routines.
*
```

```
*
* FREE_ICR: Pointer to the free ICR callback.
*
*     The free ICR callback is invoked by the metadata cache when it
*     wishes to evict an entry, and needs the client to free the memory
*     allocated for the in core representation.
*
*     The typedef for the free ICR callback is as follows:
*
*     typedef herr_t (*H5C_free_icr_func_t)(void * thing));
*
*     The parameters of the free ICR callback are as follows:
*
*     thing:  Pointer to void containing the address of the in core
*             representation of the target metadata cache entry.  This
*             is the same pointer that would be returned by a protect
*             of the addr and len of the entry.
*
*     Processing in the free ICR function should proceed as follows:
*
*     The free ICR function must free all memory allocated to the
*     in core representation.
*
*     If the function is successful, it must return SUCCEED.
*
*     If it fails for any reason, the function must return FAIL and
*     push error information on the error stack with the error API
*     routines.
*
*     At least when compiled with debug, it would be useful if the
*     free ICR call would fail if the in core representation has been
*     modified since the last serialize of clear callback.
*
* CLEAR: Pointer to the clear callback.
*
*     In principle, there should be no need for the clear callback,
*     as the dirty flag should be maintained by the metadata cache.
*.
*     However, some clients maintain dirty bits on internal data,
*     and we need some way of keeping these dirty bits in sync with
*     those maintained by the metadata cache.  This callback exists
*     to serve this purpose.  If defined, it is called whenever the
*     cache marks dirty entry clean, or when the cache is about to
*     discard a dirty entry without writing it to disk (This
*     happens as the result of an unprotect call with the
*     H5AC__DELETED_FLAG set, and the H5C__TAKE_OWNERSHIP_FLAG not
*     set.)
*
*     Arguably, this functionality should be in the NOTIFY callback.
*     However, this callback is specific to only a few clients, and
*     it will be called relatively frequently.  Hence it is made its
*     own callback to minimize overhead.
*
*     The typedef for the clear callback is as follows:
*
*     typedef herr_t (*H5C_clear_func_t)(const H5F_t *f,
*                                        void * thing,
*                                        hbool_t about_to_destroy);
*
```

```
*     The parameters of the clear callback are as follows:
*
*     f:      File pointer.
*
*     thing:  Pointer to void containing the address of the in core
*             representation of the target metadata cache entry.  This
*             is the same pointer that would be returned by a protect()
*             call of the associated addr and len.
*
*     about_to_destroy: Boolean flag used to indicate whether the
*             metadata cache is about to destroy the target metadata
*             cache entry.  The callback may use this flag to omit
*             operations that are irrelevant it the entry is about
*             to be destroyed.
*
*     Processing in the clear function should proceed as follows:
*
*     Reset all internal dirty bits in the target metadata cache entry.
*
*     If the about_to_destroy flag is TRUE, the clear function may
*     ommit any dirty bit that will not trigger a sanity check failure
*     or otherwise cause problems in the subsequent free icr call.
*     In particular, the call must ensure that the free icr call will
*     not fail due to changes prior to this call, and after the
*     last serialize or clear call.
*
*     If the function is successful, it must return SUCCEED.
*
*     If it fails for any reason, the function must return FAIL and
*     push error information on the error stack with the error API
*     routines.
*
* GET_FSF_SIZE: Pointer to the get file space free size callback.
*
*     In principle, there is no need for the get file space free size
*     callback.  However, as an optimization, it is sometimes convenient
*     to allocate and free file space for a number of cache entries
*     simultaneously in a single contiguous block of file space.
*
*     File space allocation is done by the client, so the metadata cache
*     need not be involved.  However, since the metadata cache typically
*     handles file space release when an entry is destroyed, some
*     adjustment on the part of the metadata cache is required for this
*     operation.
*
*     The get file space free size callback exists to support this
*     operation.
*
*     If a group of cache entries that were allocated as a group are to
*     be discarded and their file space released, the type of the first
*     (i.e. lowest address) entry in the group must implement the
*     get free file space size callback.
*
*     To free the file space of all entries in the group in a single
*     operation, first expunge all entries other than the first without
*     the free file space flag.
*
*     Then, to complete the operation, unprotect or expunge the first
*     entry in the block with the free file space flag set.  Since
```

```
 *     the get free file space callback is implemented, the metadata
 *     cache will use this callback to get the size of the block to be
 *     freed, instead of using the size of the entry as is done otherwise.
 *
 *     At present this callback is used only by the H5FA and H5EA dblock
 *     and dblock page client classes.
 *
 *     The typedef for the clear callback is as follows:
 *
 *     typedef herr_t (*H5C_get_fsf_size_func_t)(void * thing,
 *                                          size_t *fsf_size_ptr);
 *
 *     The parameters of the clear callback are as follows:
 *
 *     thing:  Pointer to void containing the address of the in core
 *             representation of the target metadata cache entry.  This
 *             is the same pointer that would be returned by a protect()
 *             call of the associated addr and len.
 *
 *     fs_size_ptr: Pointer to size_t in which the callback will return
 *             the size of the piece of file space to be freed.  Note
 *             that the space to be freed is presumed to have the same
 *             base address as the cache entry.
 *
 *     The function simply returns the size of the block of file space
 *     to be freed in *fsf_size_ptr.
 *
 *     If the function is successful, it must return SUCCEED.
 *
 *     If it fails for any reason, the function must return FAIL and
 *     push error information on the error stack with the error API
 *     routines.
 *
 ***************************************************************************/

/* Actions that can be reported to 'notify' client callback */
typedef enum H5C_notify_action_t {
   H5C_NOTIFY_ACTION_AFTER_INSERT,     /* Entry has been added to the cache
                                 * the insert call
                                 */
   H5C_NOTIFY_ACTION_AFTER_LOAD,       /* Entry has been loaded into the
                                 * from file via the protect call
                                 */
   H5C_NOTIFY_ACTION_AFTER_FLUSH,      /* Entry has just been flushed to
                                 * file.
                                 */
   H5C_NOTIFY_ACTION_BEFORE_EVICT      /* Entry is about to be evicted
                                 * from cache .
                                 */
} H5C_notify_action_t;

typedef herr_t (*H5C_get_load_size_func_t)(const void *udata_ptr,
                                 size_t *image_len_ptr);

typedef void *(*H5C_deserialize_func_t)(const void *image_ptr,
                                 size_t len,
                                 void *udata_ptr,
                                 hbool_t *dirty_ptr);
```

```
typedef herr_t (*H5C_image_len_func_t)(const void *thing,
                                       size_t *image_len_ptr);

#define H5C__SERIALIZE_NO_FLAGS_SET     ((unsigned)0)
#define H5C__SERIALIZE_RESIZED_FLAG     ((unsigned)0x1)
#define H5C__SERIALIZE_MOVED_FLAG       ((unsigned)0x2)

typedef herr_t (*H5C_pre_serialize_func_t)(const H5F_t *f,
                                       hid_t dxpl_id,
                                       void *thing,
                                       haddr_t addr,
                                       size_t len,
                                       haddr_t *new_addr_ptr,
                                       size_t *new_len_ptr,
                                       unsigned *flags_ptr);

typedef herr_t (*H5C_serialize_func_t)(const H5F_t *f,
                                       void *image_ptr,
                                       size_t len,
                                       void *thing);

typedef herr_t (*H5C_notify_func_t)(H5C_notify_action_t action,
                                       void *thing);

typedef herr_t (*H5C_free_icr_func_t)(void *thing);

typedef herr_t (*H5C_clear_func_t)(const H5F_t *f,
                                       void * thing,
                                       hbool_t about_to_destroy);

typedef herr_t (*H5C_get_fsf_size_t)(void * thing,
                                       size_t *fsf_size_ptr);

#define H5C__CLASS_NO_FLAGS_SET                 ((unsigned)0x0)
#define H5C__CLASS_SPECULATIVE_LOAD_FLAG        ((unsigned)0x1)
#define H5C__CLASS_COMPRESSED_FLAG              ((unsigned)0x2)

/* Following flags may only appear in test code */
#define H5C__CLASS_NO_IO_FLAG                   ((unsigned)0x4)
#define H5C__CLASS_SKIP_READS                   ((unsigned)0x8)
#define H5C__CLASS_SKIP_WRITES                  ((unsigned)0x10)

typedef struct H5C_class_t {
    int                             id;
    const char *                    name;
    H5FD_mem_t                      mem_type;
    unsigned                        flags;
    H5C_get_load_size_func_t        get_load_size;
    H5C_deserialize_func_t          deserialize;
    H5C_image_len_func_t            image_len;
    H5C_pre_serialize_func_t        pre_serialize;
    H5C_serialize_func_t            serialize;
    H5C_notify_func_t               notify;
    H5C_free_icr_func_t             free_icr;
    H5C_clear_func_t                clear;
    H5C_get_fsf_size_t              fsf_size;
} H5C_class_t;
```

Appendix 4. H5AC_aux_t

The definition of struct H5AC_aux_t is reproduced below, along with its header comment. Please note the documentation of each field in the structure. Developers are encouraged to update and correct this header comment in H5ACpkg.h as appropriate. While the following was correct as of the preparation of this document, the version in H5ACpkg.h is the master copy, and should be referred to for up to date information.

```
/****************************************************************************
 *
 * structure H5AC_aux_t
 *
 * While H5AC has become a wrapper for the cache implemented in H5C.c, there
 * are some features of the metadata cache that are specific to it, and which
 * therefore do not belong in the more generic H5C cache code.
 *
 * In particular, there is the matter of synchronizing writes from the
 * metadata cache to disk in the PHDF5 case.
 *
 * Prior to this update, the presumption was that all metadata caches would
 * write the same data at the same time since all operations modifying
 * metadata must be performed collectively.  Given this assumption, it was
 * safe to allow only the writes from process 0 to actually make it to disk,
 * while metadata writes from all other processes were discarded.
 *
 * Unfortunately, this presumption is in error as operations that read
 * metadata need not be collective, but can change the location of dirty
 * entries in the metadata cache LRU lists.  This can result in the same
 * metadata write operation triggering writes from the metadata caches on
 * some processes, but not all (causing a hang), or in different sets of
 * entries being written from different caches (potentially resulting in
 * metadata corruption in the file).
 *
 * To deal with this issue, I decided to apply a paradigm shift to the way
 * metadata is written to disk.
 *
 * With this set of changes, only the metadata cache on process 0 is able
 * to write metadata to disk, although metadata caches on all other
 * processes can read metadata from disk as before.
 *
 * To keep all the other caches from getting plugged up with dirty metadata,
 * process 0 periodically broadcasts a list of entries that it has flushed
 * since that last notice, and which are currently clean.  The other caches
 * mark these entries as clean as well, which allows them to evict the
 * entries as needed.
 *
 * One obvious problem in this approach is synchronizing the broadcasts
 * and receptions, as different caches may see different amounts of
 * activity.
 *
 * The current solution is for the caches to track the number of bytes
 * of newly generated dirty metadata, and to broadcast and receive
 * whenever this value exceeds some user specified threshold.
 *
 * Maintaining this count is easy for all processes not on process 0 --
 * all that is necessary is to add the size of the entry to the total
 * whenever there is an insertion, a move of a previously clean entry,
 * or whever a previously clean entry is marked dirty in an unprotect.
```

```
*
* On process 0, we have to be careful not to count dirty bytes twice.
* If an entry is marked dirty, flushed, and marked dirty again, all
* within a single reporting period, it only th first marking should
* be added to the dirty bytes generated tally, as that is all that
* the other processes will see.
*
* At present, this structure exists to maintain the fields needed to
* implement the above scheme, and thus is only used in the parallel
* case.  However, other uses may arise in the future.
*
* Instance of this structure are associated with metadata caches via
* the aux_ptr field of H5C_t (see H5Cpkg.h).  The H5AC code is
* responsible for allocating, maintaining, and discarding instances
* of H5AC_aux_t.
*
* The remainder of this header comments documents the individual fields
* of the structure.
*
*                                    JRM - 6/27/05
*
*
* Update: When the above was written, I planned to allow the process
*     0 metadata cache to write dirty metadata between sync points.
*     However, testing indicated that this allowed occasional
*     messages from the future to reach the caches on other processes.
*
*     To resolve this, the code was altered to require that all metadata
*     writes take place during sync points -- which solved the problem.
*     Initially all writes were performed by the process 0 cache.  This
*     approach was later replaced with a distributed write approach
*     in which each process writes a subset of the metadata to be
*     written.
*
*     After thinking on the matter for a while, I arrived at the
*     conclusion that the process 0 cache could be allowed to write
*     dirty metadata between sync points if it restricted itself to
*     entries that had been dirty at the time of the previous sync point.
*
*     To date, there has been no attempt to implement this optimization.
*     However, should it be attempted, much of the supporting code
*     should still be around.
*
*                                    JRM -- 1/6/15
*
* magic:       Unsigned 32 bit integer always set to
*            H5AC__H5AC_AUX_T_MAGIC.  This field is used to validate
*            pointers to instances of H5AC_aux_t.
*
* mpi_comm:    MPI communicator associated with the file for which the
*            cache has been created.
*
* mpi_rank:    MPI rank of this process within mpi_comm.
*
* mpi_size:    Number of processes in mpi_comm.
*
* write_permitted:  Boolean flag used to control whether the cache
*            is permitted to write to file.
*
```

```
* dirty_bytes_threshold: Integer field containing the dirty bytes
*           generation threashold.  Whenever dirty byte creation
*           exceeds this value, the metadata cache on process 0
*           broadcasts a list of the entries it has flushed since
*           the last broadcast (or since the beginning of execution)
*           and which are currently clean (if they are still in the
*           cache)
*
*           Similarly, metadata caches on processes other than process
*           0 will attempt to receive a list of clean entries whenever
*           the threshold is exceeded.
*
* dirty_bytes:  Integer field containing the number of bytes of dirty
*           metadata generated since the beginning of the computation,
*           or (more typically) since the last clean entries list
*           broadcast.  This field is reset to zero after each such
*           broadcast.
*
* metadata_write_strategy: Integer code indicating how we will be
*           writing the metadata.  In the first incarnation of
*           this code, all writes were done from process 0.  This
*           field exists to facilitate experiments with other
*           strategies.
*
*           At present, this field must be set to either
*           H5AC_METADATA_WRITE_STRATEGY__PROCESS_0_ONLY or
*           H5AC_METADATA_WRITE_STRATEGY__DISTRIBUTED.
*
* dirty_bytes_propagations: This field only exists when the
*           H5AC_DEBUG_DIRTY_BYTES_CREATION #define is TRUE.
*
*           It is used to track the number of times the cleaned list
*           has been propagated from process 0 to the other
*           processes.
*
* unprotect_dirty_bytes:  This field only exists when the
*           H5AC_DEBUG_DIRTY_BYTES_CREATION #define is TRUE.
*
*           It is used to track the number of dirty bytes created
*           via unprotect operations since the last time the cleaned
*           list was propagated.
*
* unprotect_dirty_bytes_updates: This field only exists when the
*           H5AC_DEBUG_DIRTY_BYTES_CREATION #define is TRUE.
*
*           It is used to track the number of times dirty bytes have
*           been created via unprotect operations since the last time
*           the cleaned list was propagated.
*
* insert_dirty_bytes:  This field only exists when the
*           H5AC_DEBUG_DIRTY_BYTES_CREATION #define is TRUE.
*
*           It is used to track the number of dirty bytes created
*           via insert operations since the last time the cleaned
*           list was propagated.
*
* insert_dirty_bytes_updates:  This field only exists when the
*           H5AC_DEBUG_DIRTY_BYTES_CREATION #define is TRUE.
*
```

```
*             It is used to track the number of times dirty bytes have
*             been created via insert operations since the last time
*             the cleaned list was propagated.
*
* move_dirty_bytes:  This field only exists when the
*             H5AC_DEBUG_DIRTY_BYTES_CREATION #define is TRUE.
*
*             It is used to track the number of dirty bytes created
*             via move operations since the last time the cleaned
*             list was propagated.
*
* move_dirty_bytes_updates:  This field only exists when the
*             H5AC_DEBUG_DIRTY_BYTES_CREATION #define is TRUE.
*
*             It is used to track the number of times dirty bytes have
*             been created via move operations since the last time
*             the cleaned list was propagated.
*
* Things have changed a bit since the following four fields were defined.
* If metadata_write_strategy is H5AC_METADATA_WRITE_STRATEGY__PROCESS_0_ONLY,
* all comments hold as before -- with the caviate that pending further
* coding, the process 0 metadata cache is forbidden to flush entries outside
* of a sync point.
*
* However, for different metadata write strategies, these fields are used
* only to maintain the correct dirty byte count on process zero -- and in
* most if not all cases, this is redundant, as process zero will be barred
* from flushing entries outside of a sync point.
*
*                                        JRM -- 3/16/10
*
* d_slist_ptr:  Pointer to an instance of H5SL_t used to maintain a list
*             of entries that have been dirtied since the last time they
*             were listed in a clean entries broadcast.  This list is
*             only maintained by the metadata cache on process 0 -- it
*             it used to maintain a view of the dirty entries as seen
*             by the other caches, so as to keep the dirty bytes count
*             in synchronization with them.
*
*             Thus on process 0, the dirty_bytes count is incremented
*             only if either
*
*             1) an entry is inserted in the metadata cache, or
*
*             2) a previously clean entry is moved, and it does not
*                already appear in the dirty entry list, or
*
*             3) a previously clean entry is unprotected with the
*                dirtied flag set and the entry does not already appear
*                in the dirty entry list.
*
*             Entries are added to the dirty entry list whever they cause
*             the dirty bytes count to be increased.  They are removed
*             when they appear in a clean entries broadcast.  Note that
*             moves must be reflected in the dirty entry list.
*
*             To reitterate, this field is only used on process 0 -- it
*             should be NULL on all other processes.
*
```

```
 * d_slist_len: Integer field containing the number of entries in the
 *             dirty entry list.  This field should always contain the
 *             value 0 on all processes other than process 0.  It exists
 *             primarily for sanity checking.
 *
 * c_slist_ptr: Pointer to an instance of H5SL_t used to maintain a list
 *             of entries that were dirty, have been flushed
 *             to disk since the last clean entries broadcast, and are
 *             still clean.  Since only process 0 can write to disk, this
 *             list only exists on process 0.
 *
 *             In essence, this slist is used to assemble the contents of
 *             the next clean entries broadcast.  The list emptied after
 *             each broadcast.
 *
 * c_slist_len: Integer field containing the number of entries in the clean
 *             entries list (*c_slist_ptr).  This field should always
 *             contain the value 0 on all processes other than process 0.
 *             It exists primarily for sanity checking.
 *
 * The following two fields are used only when metadata_write_strategy
 * is H5AC_METADATA_WRITE_STRATEGY__DISTRIBUTED.
 *
 * candidate_slist_ptr: Pointer to an instance of H5SL_t used by process 0
 *             to construct a list of entries to be flushed at this sync
 *             point.  This list is then broadcast to the other processes,
 *             which then either flush or mark clean all entries on it.
 *
 * candidate_slist_len: Integer field containing the number of entries on the
 *             candidate list.  It exists primarily for sanity checking.
 *
 * write_done:  In the parallel test bed, it is necessary to ensure that
 *             all writes to the server process from cache 0 complete
 *             before it enters the barrier call with the other caches.
 *
 *             The write_done callback allows t_cache to do this without
 *             requiring an ACK on each write.  Since these ACKs greatly
 *             increase the run time on some platforms, this is a
 *             significant optimization.
 *
 *             This field must be set to NULL when the callback is not
 *             needed.
 *
 *             Note: This field has been extended for use by all processes
 *                  with the addition of support for the distributed
 *                  metadata write strategy.
 *                                        JRM -- 5/9/10
 *
 * sync_point_done:  In the parallel test bed, it is necessary to verify
 *             that the expected writes, and only the expected writes,
 *             have taken place at the end of each sync point.
 *
 *             The sync_point_done callback allows t_cache to perform
 *             this verification.  The field is set to NULL when the
 *             callback is not needed.
 *
 ****************************************************************************/

#ifdef H5_HAVE_PARALLEL
```

```
#define H5AC__H5AC_AUX_T_MAGIC        (unsigned)0x00D0A01

typedef struct H5AC_aux_t
{
    uint32_t    magic;

    MPI_Comm    mpi_comm;

    int         mpi_rank;

    int         mpi_size;

    hbool_t     write_permitted;

    int32_t     dirty_bytes_threshold;

    int32_t     dirty_bytes;

    int32_t     metadata_write_strategy;

#if H5AC_DEBUG_DIRTY_BYTES_CREATION

    int32_t     dirty_bytes_propagations;

    int32_t     unprotect_dirty_bytes;
    int32_t     unprotect_dirty_bytes_updates;

    int32_t     insert_dirty_bytes;
    int32_t     insert_dirty_bytes_updates;

    int32_t     move_dirty_bytes;
    int32_t     move_dirty_bytes_updates;

#endif /* H5AC_DEBUG_DIRTY_BYTES_CREATION */

    H5SL_t *    d_slist_ptr;

    int32_t     d_slist_len;

    H5SL_t *    c_slist_ptr;

    int32_t     c_slist_len;

    H5SL_t *    candidate_slist_ptr;

    int32_t     candidate_slist_len;

    void        (* write_done)(void);

    void        (* sync_point_done)(int num_writes,
                                haddr_t * written_entries_tbl);

} H5AC_aux_t; /* struct H5AC_aux_t */
```