RFC: Flush/Evict Individual Objects in Metadata Cache

Mike McGreevy

Allow metadata for individual HDF5 objects to be flushed or evicted from the metadata cache within an application.

# Introduction

This RFC proposes modifying the HDF5 library to support flushing (during writing) or evicting (during reading) individual objects in the metadata cache. This will allow changes to individual objects to be published by the writing application and updated by a reading application, without flushing all the metadata for the file (during writing) or closing and re-opening the file (during reading). These fine-grained flush/evict operations will improve application performance and reduce/eliminate the number of redundant I/O operations performed.

# Approach

The HDF5 library does not currently track which metadata belongs to which object, so in order to flush or evict groups of metadata belonging to the same object, some additional tracking changes will be included with this feature. The method of tracking will be to tag each piece of metadata in the cache with information relating it to the object that it belongs to. With that information, any function that requires acting upon a single object’s metadata can search through the tag values of metadata entries and act accordingly.

After this additional tagging framework is added throughout the library, there will be enough information available in the cache and on each piece of metadata to write a set of API functions to give a user control over flushing individual objects.

# Implementation

## Metadata Tagging

### Background

Currently, the cache level of the library is quite abstracted from any layers above it, and from within the cache, it is impossible to tell where and how various entries came into existence. Metadata cache entries consist of, more or less, some metadata that belongs on disk, and the address at which this piece of metadata belongs. Along with other information to indicate how the cache is allowed to operate on an entry, there is no additional information that can be used to trace back through the library to determine the object it belongs to and from where it originated. As such, the largest obstacle in implementing single object flushing/evicting is adding enough information into the cache in order to know what to flush during any given partial-flush request.

### Metadata Tags

The solution to the above predicament is to add an additional field to all metadata cache entries that indicate which object it belongs to. This tag field will hold values that are unique to individual objects, and thus groups of metadata that belong to the same object will all be assigned the same tag. Once in place, this framework will allow the cache to search through its entries and investigate their tag values in order to return a set of entries belonging to a single object, where they can be further operated on (be it to flush them, evict them, et cetera). Applying specific object-identifying tags to all pieces of metadata, however, is more easily said than done.

### Application of Tags

The current cache is quite separate from the interfaces it is attached to. It is easy to send information into the cache level, but hard to pull information into it from within. Thus, at the time a metadata entry is created in the cache, there is no way to figure out where the request came from in order to determine which tag to apply to it, so the determination of what tag to apply to a piece of metadata must be performed in a level above the cache, before the entry is created, and then passed into the cache for it to be assigned to an entry.

At first glance, the call to set (create) or protect (look-up) something in the cache would be a nice point to indicate a tag value to assign to the entry it might add to the cache. Unfortunately, the solution is not as simple as adding an extra field to all H5AC\_set and H5AC\_protect calls. This is because there are locations in the library at which, when these calls are made, the object to which the metadata belongs has long been abstracted out. In some cases there are a few interfaces between the calling interface like H5D, which would point to a dataset to modify, and the H5AC call, which results in new metadata in the cache.

In an effort to avoid modifying a large number of internal function signatures in the library in order to pass down the tag parameter, the proposed solution is to add this information into the data access property list, which is fully propagated throughout the interfaces and down into the cache. Shortly after the library is aware of the object being operated on, a tag value unique to that object will be set into the data access property list that is sent deeper into the library, which will ultimately be assigned to any metadata created during the call.

### Tag Value

All objects have an object header metadata entry, and this is always set up (during creation) or protected (during other operations) before any other entries belonging to that object are created in the cache. Thus, the tag value for an object can simply be the address of its object header metadata entry on disk.

### Global Tags

Unfortunately, setting up a tag value in the data access property list and sending it on its way doesn’t encompass all situations, as there are a few additional quirks and problems that need to be dealt with. Particularly, some metadata might be created within an API call that has nothing to do with any specific object a user would want to flush, an example being the free space management, which has metadata that can be modified within the cache during several API calls. Alternatively, some metadata entries might have information pertaining to more than one object, such as global heap metadata. Rather than managing a setup that allows multiple tags to be applied to the same piece of metadata, or arbitrarily assigning non-essential entries a specific object’s tag, a group of tag values have been set aside that can be used in these special cases.

These specific tag values, referred to as global tags, are predefined values intended to encompass all of one type of metadata entry. For example, all free space header and section info metadata will always get the same tag that identifies it as being related to the free space manager. The same goes for superblock metadata, global heap metadata, and shared object header message metadata.

This serves two purposes. First, in the case where metadata is created that doesn’t belong to any specific object, such as free space management metadata, we give it a global tag that can be ignored during any specific flush or evict of an individual object. This prevents unnecessarily including things in flushes that really do not need to be flushed until the whole file is flushed.

Second, in the case where metadata might have shared information and is used by multiple objects, such as the global heap entries, we can use a specific global tag as an additional search parameter when finding metadata belonging to a single object, and flush that as well. This ensures that entries that have information shared amongst multiple objects are always flushed.

### Setting and resetting tag values

Within a given function call, there exists the possibility that the tag value currently residing in the data access property list needs to be switched to something else, like in the case where we want to apply a global tag to an entry. In these cases, it is necessary to ensure that the tag value is restored to whatever it was prior to application of the global tag, in the event that the calling function has further use for whatever the initial value in the data access property list was. Thus, in most cases where a tag value is set up prior to function calls that might modify metadata, since a tag might already exist in the property list, the tag value should be reset afterwards to what it originally was, in order to ensure that the value in the property list when entering the function is what resides when the function concludes.

### Tag setup/takedown locations

Since there are a large (>80) number of locations in the library that need to have a specific tag set up, a lot of extra and repetitive code will need to be inserted in these locations. To make life a little easier now and for future developers who will be making modifications in these areas, the majority of metadata tags will be set up within special FUNC\_ENTER and FUNC\_LEAVE macros. This will serve to hide most of this framework and ensure that current functions aren’t going to be littered with a bunch of code unrelated to the purpose of the function.

### Other tags

In addition to global tags set aside for specific types of metadata entries, a few additional tag values have been reserved for various other behaviors, described below:

1) Invalid Tag. This is the default tag value, and is used for debugging purposes. This will be helpful when modifying code that will result in a shift of where new metadata is added into the cache, and will indicate to developers when additional tag application locations need to be set up. If an entry is being created in the cache and the tag value in the data access property list is equal to the Invalid Tag, then a failure will occur, since a tag was not set up properly.

2) Ignore Tag. This tag is used for testing purposes. Since tags are set up in more or less arbitrary locations at any point in a given function hierarchy, testing can be problematic when writing unit tests or non-API level tests that jump over the place where a tag is set up. Thus, some entries would never get changed from the Invalid Tag, and the test would fail in the cache when metadata is created. Therefore, a function will be provided for use in the tests that indicates to the cache to ignore all tag values, as well as to set default tag values to the ignore tag, so as to not get caught by the cache’s sanity checking.

3) Copied Tag. This tag will be used only in the case of copying objects. When copying an object with metadata in the cache, the destination object’s metadata is created and put in the cache before its object header is added to the cache. Thus, unlike all other locations in the library, the tag value to apply to this new metadata is not known at its time of creation. Therefore, a temporary copied flag is applied to this metadata, indicating that this metadata was recently copied from existing metadata. After an object copy occurs, the address of the new object’s object header is known, and can then be applied to all entries in the cache that were created as a result of an object copy. This is the only time an entry in the cache would have its tag value changed to something else after it was initially inserted or protected.

## New API Functions

Two sets of API functions are included in this feature. They are H5\*flush and H5\*refresh, where \* represents a version of the function in the H5D, H5G, H5T, and H5O interfaces, for datasets, groups, named datatypes, and generic objects, respectively.

The use case for the flush function is to allow a writing application to immediately flush all metadata belonging to a specific object to disk in order to allow a reading application to grab an up-to-date version of said object without the writing application having to flush the entire file to do so. In this operation, only metadata associated with the targeted object is flushed to disk, and no entries are unnecessarily evicted from the cache.

The use case of the refresh function is to allow a reading application to immediately refresh all the buffers associated with an object, in order to pick up the most recent data that has been sent to disk via a writing application. This action requires all metadata in the cache to be evicted so that more recent metadata can be re-loaded from disk. In order to evict all desired metadata, the function will first close the object (to make sure all associated metadata is not protected), evict the appropriate metadata, and then re-open the object, which will re-load the object header with the version on disk. The function will re-register the object’s ID, so the application need not worry about doing anything beyond calling the refresh function on the targeted object.

In both instances, the functions will use the tag values contained on all pieces of metadata, as described above, in order to determine which metadata to flush, with the address of the object header in question being the targeted search value.

While the individual object flush functions have an associated (already implemented) H5Fflush function which operates on the entire file, this RFC does not include plans to introduce a complementary H5Frefresh function, due to the problem of having to re-register all open IDs for the file’s objects after closing and re-opening them along with the file. This may be a desired feature to be eventually included in the set of H5\*refresh functions, which is why it is being mentioned here, but that work is not currently scheduled for implementation.

## New API Function Reference Manual Entries

### H5Dflush

**Name:** H5Dflush

**Signature:**

*herr\_t* H5Dflush (*hid\_t* dataset\_id)

**Purpose:**

Flushes all buffers associated with a dataset to disk.

**Description:**

H5Dflush causes all buffers associated with a dataset to be immediately flushed to disk without removing the data from the cache.

**Note:**

HDF5 does not possess full control over buffering. H5Dflush flushes internal HDF5 buffers then asks the operating system (the OS) to flush the system buffers for the open files. After that, the OS is responsible for ensuring that the data is actually flushed to disk.

**Parameters:**

*hid\_t* dataset\_id IN: Identifier of the dataset to be flushed.

**Returns:**

Returns a non-negative value if successful; otherwise returns a negative value.

### H5Gflush

**Name:** H5Gflush

**Signature:**

*herr\_t* H5Gflush (*hid\_t* group\_id)

**Purpose:**

Flushes all buffers associated with a group to disk.

**Description:**

H5Gflush causes all buffers associated with a group to be immediately flushed to disk without removing the data from the cache.

**Note:**

HDF5 does not possess full control over buffering. H5Gflush flushes internal HDF5 buffers then asks the operating system (the OS) to flush the system buffers for the open files. After that, the OS is responsible for ensuring that the data is actually flushed to disk.

**Parameters:**

*hid\_t* group\_id IN: Identifier of the group to be flushed.

**Returns:**

Returns a non-negative value if successful; otherwise returns a negative value.

### H5Tflush

**Name:** H5Tflush

**Signature:**

*herr\_t* H5Tflush (*hid\_t* dtype\_id)

**Purpose:**

Flushes all buffers associated with a named datatype to disk.

**Description:**

H5Tflush causes all buffers associated with a named datatype to be immediately flushed to disk without removing the data from the cache.

**Note:**

HDF5 does not possess full control over buffering. H5Tflush flushes internal HDF5 buffers then asks the operating system (the OS) to flush the system buffers for the open files. After that, the OS is responsible for ensuring that the data is actually flushed to disk.

**Parameters:**

*hid\_t* dtype\_id IN: Identifier of the named datatype to be flushed.

**Returns:**

Returns a non-negative value if successful; otherwise returns a negative value.

### H5Oflush

**Name:** H5Oflush

**Signature:**

*herr\_t* H5Oflush (*hid\_t* object\_id)

**Purpose:**

Flushes all buffers associated with an HDF5 object to disk.

**Description:**

H5Oflush causes all buffers associated with an object to be immediately flushed to disk without removing the data from the cache.

**Note:**

HDF5 does not possess full control over buffering. H5Oflush flushes internal HDF5 buffers then asks the operating system (the OS) to flush the system buffers for the open files. After that, the OS is responsible for ensuring that the data is actually flushed to disk.

**Parameters:**

*hid\_t* object\_id IN: Identifier of the object to be flushed.

**Returns:**

Returns a non-negative value if successful; otherwise returns a negative value.

### H5Drefresh

**Name:** H5Drefresh

**Signature:**

*herr\_t* H5Drefresh (*hid\_t* dataset\_id)

**Purpose:**

Refreshes all buffers associated with a dataset with data from disk.

**Description:**

H5Drefresh causes all buffers associated with a dataset to be cleared and immediately re-loaded with updated contents from disk.

This function essentially closes the dataset, evicts all metadata associated with it from the cache, and then re-opens the dataset, pulling new metadata from the file back into the cache.

**Parameters:**

*hid\_t* dataset\_id IN: Identifier of the dataset to be flushed.

**Returns:**

Returns a non-negative value if successful; otherwise returns a negative value.

### H5Grefresh

**Name:** H5Grefresh

**Signature:**

*herr\_t* H5Grefresh (*hid\_t* group\_id)

**Purpose:**

Refreshes all buffers associated with a group with data from disk.

**Description:**

H5Grefresh causes all buffers associated with a group to be cleared and immediately re-loaded with updated contents from disk.

This function essentially closes the group, evicts all metadata associated with it from the cache, and then re-opens the group, pulling new metadata from the file back into the cache.

**Parameters:**

*hid\_t* group\_id IN: Identifier of the group to be flushed.

**Returns:**

Returns a non-negative value if successful; otherwise returns a negative value.

### H5Trefresh

**Name:** H5Trefresh

**Signature:**

*herr\_t* H5Trefresh (*hid\_t* dtype\_id)

**Purpose:**

Refreshes all buffers associated with a named datatype with data from disk.

**Description:**

H5Trefresh causes all buffers associated with a named datatype to be cleared and immediately re-loaded with updated contents from disk.

This function essentially closes the named datatype, evicts all metadata associated with it from the cache, and then re-opens the named datatype, pulling new metadata from the file back into the cache.

**Parameters:**

*hid\_t* dtype\_id IN: Identifier of the named datatype to be flushed.

**Returns:**

Returns a non-negative value if successful; otherwise returns a negative value.

### H5Orefresh

**Name:** H5Orefresh

**Signature:**

*herr\_t* H5Orefresh (*hid\_t* object\_id)

**Purpose:**

Refreshes all buffers associated with an HDF5 object with data from disk.

**Description:**

H5Orefresh causes all buffers associated with an object to be cleared and immediately re-loaded with updated contents from disk.

This function essentially closes the object, evicts all metadata associated with it from the cache, and then re-opens the object, pulling new metadata from the file back into the cache.

**Parameters:**

*hid\_t* object\_id IN: Identifier of the object to be flushed.

**Returns:**

Returns a non-negative value if successful; otherwise returns a negative value.

# Recommendation

The new routines and supporting library modifications should be made available in the 1.10.0 release of HDF5.

# Revision History

May 5, 2010: Version 1 circulated for comment within The HDF Group.