RFC: Metadata Cache Issues under SWMR and their Solutions

Dana Robinson
Quincey Koziol

The HDF5 library contains a cache that provides more efficient access to HDF5 file metadata. Although helpful in most instances, this cache can cause problems when the single-writer/multiple-readers (SWMR) data access pattern is used to access an HDF5 file. Additionally, there are even deeper I/O issues that can affect HDF5 file metadata operations.

This RFC describes the cache-oriented problems that the HDF5 library must overcome in order to implement the SWMR feature as well as our solutions.

1 Introduction

The single-writer/multiple-readers (SWMR) feature of the HDF5 library will allow concurrent reading of an HDF5 file by multiple reader processes while a writer process modifies the file. This feature will be implemented without requiring file locking or inter-process communication (IPC).

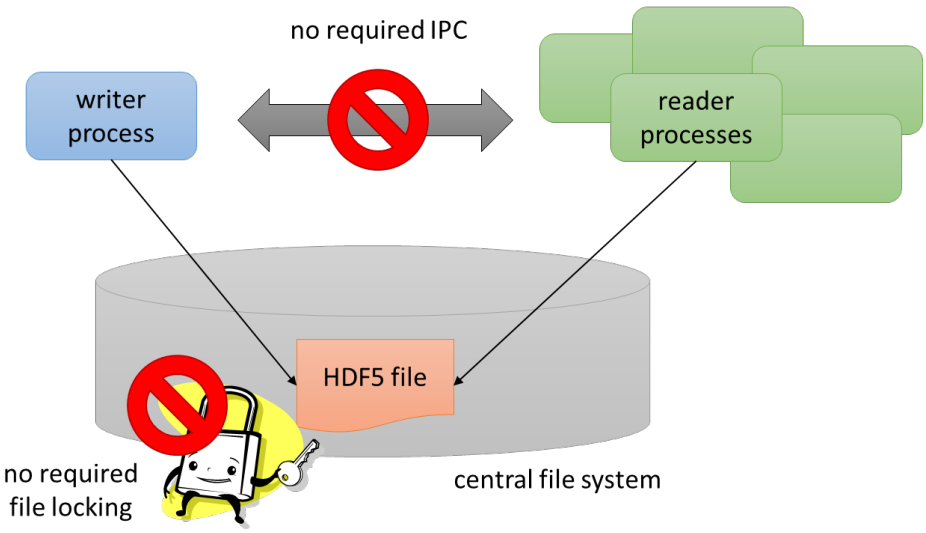

Figure 1-1: The SWMR feature as implemented in HDF5.

Although conceptually simple, SWMR is challenging to implement. The primary problem is that, in a SWMR situation, the true state of the central HDF5 file is spread across two locations: the on-storage HDF5 file and the writer process' in-memory state, including the cache layers. Unlike simple "flat" text or binary files, HDF5 files have a complex internal structure and errors in this structure can result in broken, unreadable files. Correct reader operation requires being able to see a valid HDF5 file at

all times and this may not be true if crucial structural information has not yet been propagated to disk by the writer.

In the HDF5 library, all internal structure (metadata) objects are accessed via a central metadata cache. It is in this caching layer that most of the changes for SWMR must be made. The operation of this caching layer and the changes that are being made to enable the SWMR feature are the subject of this document.

Intended audience and assumptions:

This document is primarily intended to help early adopters and funders of the SWMR feature understand the metadata cache implications of the SWMR feature. The knowledge presented here will help convey the scope of the work required for a correct implementation. Readers are assumed to have a basic understanding of cache operations and HDF5.

NOTE:

This document applies to the SWMR branch of HDF5 library development[1], which is based on the future 1.10.x version of the library. The current 1.8.x branch of development is unable to support SWMR due to a lack of checksum-containing metadata structures for key functions (described below).

2 SWMR Semantics

We define *SWMR semantics* as a single writer process writing to a single HDF5 file while one or more reader processes concurrently read from the file with no required file locking or inter-process communication between the writers and readers. All processes access the file over a common file system. At this time, there are no constraints on which processes must open or close the file first. Processes simply need to use the `H5F_ACC_SWMR_WRITE/READ` flag when creating or opening the file. This will change in the future, when we implement access blocking in the library.

More precisely, the future full semantics of SWMR operations will be:

- Multiple readers can open the same file for reading when no writer has the file open for writing.
- No reader can open the file for reading when a non-SWMR writer is accessing the same file for writing.
- No writer can open the file for writing when reader(s) are accessing the same file for reading.
- No writer can open the file for writing when a writer already holds the file open for writing.
- Multiple readers can open the same file for “reading and SWMR-read” when a writer opens the file for “writing and SWMR-write”.
- Non-SWMR readers will not be able to open a file opened for SWMR writing.

Note that these policies are not enforced by the SWMR prototype HDF5 library at this time. It is up to the user to avoid violating them.

In theory, it would be ideal if the file system guaranteed write ordering and atomicity, though recent changes to the cache (described below) relax these requirements. A write ordering guarantee

[1] Current public Subversion repository: http://svn.hdfgroup.uiuc.edu/hdf5/branches/revise_chksum_retry/

ensures that writes A, B, and C will always be read as ABC by a reader and not in some other order, such as BAC. Many network file systems, particularly NFS, do not guarantee write ordering. A write atomicity guarantee ensures that data written by a single `write()` call will be written to disk as a unit, so a partial write will never be encountered by a reader. Many file/operating systems are not atomic at the write call level but are instead atomic at some other level, such as the disk page size.

3 HDF5 Metadata and Basic Metadata Cache Operations

3.1 Metadata and Stored Objects

In addition to the primary data stored by the user, an HDF5 file contains *file metadata* that is used to organize, locate/index, and describe the contents of the file. It serves many purposes, including chunk index structures, symbol tables representing groups and links, and object headers that describe the stored data (modification times, number of elements, etc.). This file metadata is largely invisible to the user and should not be confused with *user metadata*, which is stored as attributes attached to HDF5 file objects such as groups and datasets.

The HDF5 file format document is available on the web[2,3] and describes the metadata structures used in the file. Although this is a very low-level document intended for developers, it does give a rough idea of what file metadata objects look like.

3.2 Normal Operations

The metadata cache sits between the core object manipulation (logical) parts of the library and the I/O layer. All file object reads and writes occur via the cache. The cache cannot be disabled; the logical library code never reads metadata directly from the disk. The metadata cache is one of two key caches in the library, the other being the chunk cache which is independent and managed separately (though there are some associations under SWMR, via chunk proxies, described below).

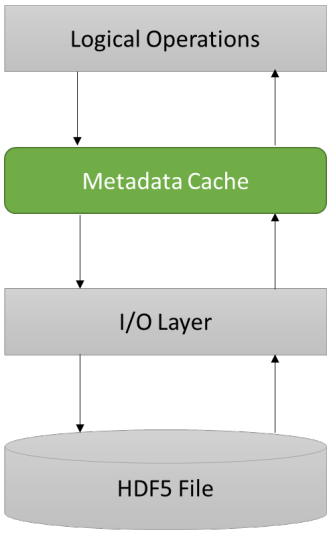

Figure 3-1: Position of the metadata cache in the HDF5 library.

As an example, when a chunk index node is required by the library, a request for the node is sent to the cache, which either returns the node immediately if it is contained in the cache or reads it into the cache from disk and then returns the node if it has not been previously cached. Writing is handled similarly. The metadata cache is aware of both the type of each metadata object and the higher-level object to which it belongs. This is tracked via tags attached to each metadata object. Cache objects are evicted and, if dirty, flushed using a modified least recently used (LRU) algorithm. It is very important to understand that the HDF5 library and thus the cache are not asynchronous in any way. The cache does not

[2] Current 1.8.x format: http://www.hdfgroup.org/HDF5/doc/H5.format.html
[3] Future 1.10.x format (supported under SWMR):
http://www.hdfgroup.org/HDF5/doc_test/revise_chunks/H5.format.html (this is a temporary location).

operate on a background thread. Instead cache operations like flush passes are triggered by conditions such as the current free space in the cache on cache access. These cache operations then run to completion before processing resumes.

Various metadata cache parameters can be adjusted via the public H5Pset_mdc_config() API call. This function takes an input H5AC_cache_config_t structure that contains many members. Most of these parameters are relatively unimportant for SWMR aside from eviction control, discussed below in the corking section.

3.3 Corking

Evictions from the metadata cache can be prevented via the internal H5C_set_evictions_enabled() function, which is known as *corking* the cache. When evictions are disabled, the metadata cache will grow in size until it runs out of available memory. Control over cache corking allows advanced users control when objects become visible in the file and to avoid some of the extra flush overhead imparted by SWMR. This feature is can be enabled by the H5Pset_mdc_config() API function by setting the passed-in H5AC_cache_config_t struct's evictions_enabled member to FALSE. In the future, a more fine-grained public H5Ocork() API function that operates at the object (dataset, etc.) level will be implemented.

3.4 Aggregation

To improve efficiency, the metadata cache supports aggregating small metadata objects into a contiguous block so that they can be written out using a single write() call. This feature can be adjusted with the H5Pset_meta_block_size() function. The default size is 2k and aggregation is turned on by default.

The I/O subsystem of the library uses pluggable drivers called *virtual file drivers* (VFDs) as a way of abstracting low-level I/O operations from the main library logic and not all of these drivers support metadata aggregation. Currently, the multi and split VFDs do not support this feature. All others, including the default sec2 VFD, support aggregation.

4 Problems That Affect SWMR

4.1 Flush Ordering

HDF5 metadata objects can refer to other objects in the file by storing, either explicitly or implicitly, the second object's offset in the file. The first object that stores the offset is termed the metadata *parent* and the object that is referred to is the metadata *child* (see figure). Note that the child may itself be a parent of some other object, and this can form long chains of parent-child relationships in the cache. Parents may also have multiple children (e.g., B-trees may have multiple child nodes) and children may have multiple parents, though cycles are not allowed. For SWMR-safe file modifications to work correctly, metadata children must be written to storage before their parents. If this were not done, a reader that attempts to resolve a missing child will encounter errors and possibly crash.

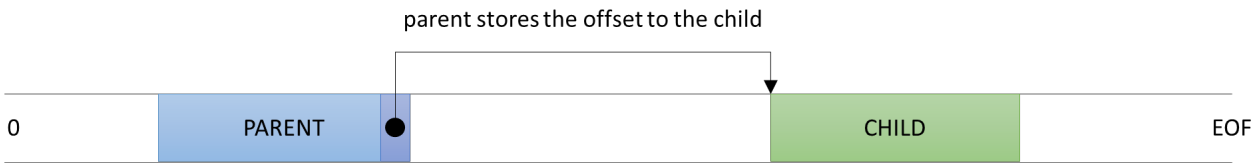

Figure 4-1: Parent-child relationships between metadata objects in the HDF5 file.

4.2 Torn Writes

Another potential problem exhibited by the SWMR access pattern is that writes may not be atomic with respect to the platform's `write()` call. When this occurs, the higher level write call is split into multiple lower-level write calls. A logical `write()` call which has only partially completed due to non-atomicity is called a *torn write*. These torn writes can result in a reader seeing a partial data object in the file which can result in incorrect behavior and crashes.

As an example, on Linux systems the POSIX `write()` call is only atomic with respect to the disk's page size, which is typically 4k. This means that a 32k data object may be split into as many as 8 separate writes, which could be preempted at any point by the OS, allowing a reader to see a subset of the full 32k, with the rest of the object containing garbage data. This is illustrated in the figure below.

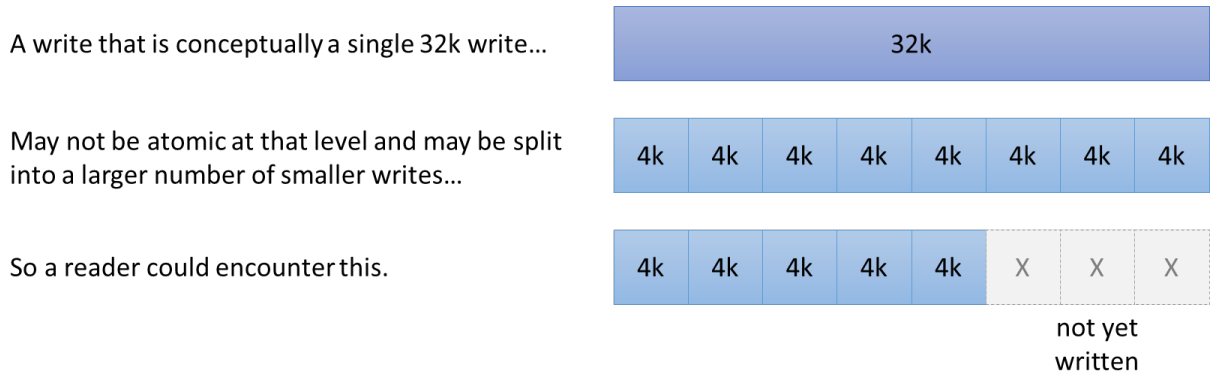

Figure 4-2: Graphical depiction of torn writes.

5 Solutions

5.1 Flush Dependencies

The metadata cache has been modified so that when a parent file object stores the offset of a second child object, the child is written before the parent. This ensures that an offset stored in a parent will always resolve to a correctly stored child object. These parent-child arrangements are called *flush dependencies* and they are tracked in the cache.

Under SWMR, the cache flush algorithm must be modified. When dirty objects are flushed from the cache, they can only be written out after all children have been flushed. This is accomplished by making multiple iterations over the dirty objects in the cache, flushing the lowest level children until all dirty objects have been flushed.

This flush ordering is transparent to the user. Is it always in effect when SWMR writes are enabled and no special API calls are necessary to enable the feature. It is disabled when a file has not been opened for SWMR writing.

Flush dependencies have not been set up for all data structures at this time. All chunk indexing structures have been modified, so appending to datasets is currently supported under SWMR. Object headers and symbol tables have not, however, so creating new file objects under SWMR is not yet supported, though this will be added in future versions of the SWMR prototype.

5.2 Chunk Proxies

Metadata objects can store offsets to raw data as well as metadata. Raw data can be stored in metadata (as an optimization when data are small), written directly to disk (when unchunked), or written via a chunk cache (when chunked). The latter poses a problem for SWMR, since this cache is managed separately from the metadata cache. Without some form of synchronization, metadata that refers to the offset of a chunk (e.g., in a chunk index) could be written to disk before the chunk. A reader could attempt to load data at this offset, resulting in an error.

The solution to this problem is to create metadata cache objects that represent chunks stored in the chunk cache (see http://www.hdfgroup.org/HDF5/doc/Advanced/Chunking/index.html for more information on chunking). These metadata objects are called *chunk proxies*. They exist only in the cache and are not propagated to storage. If SWMR writes are enabled, a chunk proxy is created in the metadata cache whenever a chunk is created in chunk cache. This proxy object acts as a cross-cache dependency between a metadata cache object and a chunk cache object. Like any other flush dependency, this prevents the parent such as a B-tree node from being written to disk before the child (chunk, via the proxy).

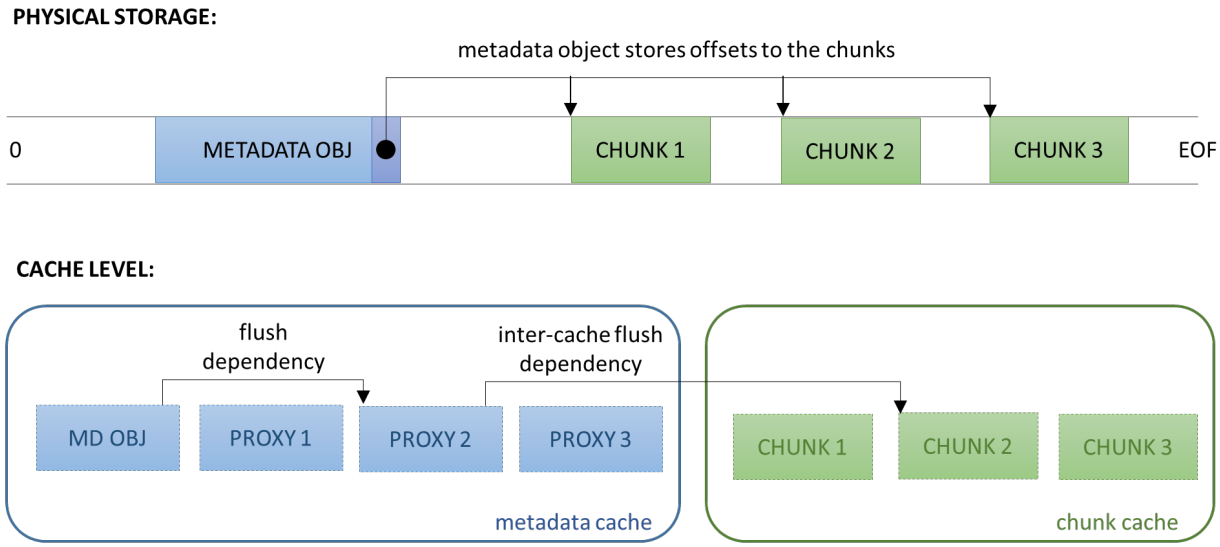

Figure 5-1: Depiction of chunk proxies in the metadata cache. For clarity, only one dependency relationship is shown in the caches.

5.3 Read Retries

The metadata cache has been modified so that file data objects are re-read from the disk when the calculated checksum does not match the stored checksum. The number of retries can be set by the

user with the H5Pset_metadata_read_attempts() function. The default for SWMR read access is 100 retries (for non-SWMR access it is 1).

This retry feature (combined with checksummed metadata) is the key behind the SWMR feature's ability to relax the atomicity and ordering requirements of the file system. File objects that cannot be read due to torn writes or that don't exist yet due to ordering issues will fail the checksum comparison on read, prompting a re-read instead of an error or crash. Although failures could still occur in extremely pathological cases, this largely mitigates the issue.

Retries are transparent to the user. They are in effect by default when SWMR read access is set and no special API calls are necessary to enable the feature.

Additionally, there is a new function H5Fget_metadata_retries_info() that will emit the number of retries for each metadata object type that occurred while the file was open. This will allow programmers to set an appropriate value for the number of retries and can be useful when diagnosing problems with excessive retries.

5.4 Prevention of Non-Checksummed Metadata Use

As mentioned in the previous section, read retries are an important part of ensuring correct behavior under SWMR. Since this feature requires a data object checksum, writes of non-checksummed data will be prohibited under SWMR write semantics. This has been partially implemented in the current version of the SWMR prototype. Version 1 B-tree nodes, which do not contain a checksum, cannot be written to storage under SWMR write semantics preventing their use. In the future, a more comprehensive solution will be implemented that enforces a policy concerning non-checksummed data use.

Improper data structure use is prevented by setting the library version bounds on the file access property list using H5Pset_libver_bounds() with both high and low bounds set to H5F_LIBVER_LATEST. This will allow the library to use the latest version of the file format, which uses file data structures that have checksums.

5.5 Disable Metadata Accumulator

The metadata accumulator can violate the ordering constraint of SWMR semantics. The violations occur when writes are delayed by being held in the cache for aggregation, potentially causing flush ordering issues.The solution was to switch off the metadata accumulator when the SWMR flag is set. This is done transparently with no action required by the user.

Acknowledgements

This work was supported by a customer of The HDF Group, Diamond Light Source.

Revision History

November 17, 2013:	Version 1 circulated for comment within The HDF Group SWMR team.
November 27, 2013:	Version 2 incorporated some comments from Vailin; sent to DLS and posted on FTP

References

1. The HDF Group. "HDF5 Single-Write/Multiple-Reader Feature Design and Semantics"
2. The HDF Group. "RFC: Read Attempts for Metadata with Checksum"
3. The HDF Group. "HDF5 User's Guide," http://www.hdfgroup.org/HDF5/doc/UG/index.html
4. The HDF Group. "HDF5 Reference Manual," http://www.hdfgroup.org/HDF5/doc/RM/RM_H5Front.html
5. The HDF Group. "HDF5 File Format Specification Version 2.0," http://www.hdfgroup.org/HDF5/doc/H5.format.html
6. Current SWMR Subversion development branch: http://svn.hdfgroup.uiuc.edu/hdf5/branches/revise_chksum_retry/