RFC: Metadata Journaling to Improve Crash Survivability

John Mainzer

Introduction

HDF5 files written via calls to the HDF5 library are susceptible to corruption in the event of application or system crash. Guided by requirements gathered from several user communities, THG is in the process of developing an approach to minimize file corruption when crashes occur. While the optimal scenario is to prevent corruption altogether, that desire must be balanced with the need for very fast writes. The proposed design attempts to strike a reasonable balance between these conflicting goals by providing periodic rollback points without support for fine-grain transaction guarantees.

As the first step toward improving the ability of HDF5 to survive crashes, the design described in this RFC should guarantee an HDF5 file with consistent metadata can be reconstructed in the event of a crash. The approach and design details are presented in the main body of the RFC. The Appendix contains background materials that were considered in the development of the design.

1 Approach

The basic approach we will use is journaling. Whenever a piece of metadata is modified, and whenever the metadata is known to be in a consistent state—initially at the start of the journal, and at the end of each API call that modifies metadata—we will make a note in the journal.

If the application crashes, we will use a recovery program to replay the journal by applying in order all metadata writes until the end of the last completed transaction written to the journal file. The replay should result in a file that is internally consistent in terms of metadata. Note, however, there is no guarantee on the state of the raw data—it will contain whatever made it to disk prior to the crash.

The journaling approach described is conceptually quite simple. Implementation in the HDF5 library will require a number of modifications and additions, which can be loosely categorized as file format changes, library code changes, and implementation of the new crash recovery tool. These modifications and additions are discussed in the following sections.

2 HDF5 and Journal File Formats

The HDF5 file format changes required for the journaling implementation outlined above are relatively minor and are restricted to the superblock. Those changes, as well as the preliminary format for the journal file, are presented below.

2.1 Superblock Additions

The superblock additions must allow us to record the following information:

1) Whether or not journaling is in use

    A boolean flag indicating whether the file is currently open with journaling enabled.

    It would be convenient if we could make this flag TRUE whenever journaling is enabled and the on-disk representation of the file is not in a consistent state. However, to do this we would need some method of syncing out the superblock without forcing a sync of the rest of the HDF5 file.

    This syncing of the superblock in isolation will not in general be doable, so we will simply set this flag to TRUE and then sync out the file on file open (when the superblock is the only portion of the file that we have written), and then set it to FALSE again just before closing the file and immediately after we have synced out the rest of the file.

    If there is a crash between these two states, we will have to look at the journal to see if the file is currently in a consistent state. If the journal is empty, it is. If it is not, we have work to do.

2) Whether the journal is stored in the HDF5 file or in an external file

    In the initial version, the journal will always be in an external file. However, if there is demand for it in the future, we may move the journal into the HDF5 file on platforms where the underlying file system provides the necessary support.

3) Location of the journal

    In the initial implementation, this field will simply be the full path of the journal file.

    In future versions, it may be used to specify the base address of the journal inside the HDF5 file.

4) Journal version number

    We will probably have occasion to change the journal syntax from time to time. A journal version number should make this less painful.

In all probability, the journaling implementation information outlined above will be included in a message in the superblock extension that is part of the 1.8 release.

2.2 Journal File Format

The journal file format will be worked out in greater detail as we proceed, but at present we expect it to roughly follow this syntax:

```
<journal> --> <header> <body>

<header> --> <header_start_tag>
             <journal_version_number>
             <target_file_name_len>
             <target_file_name>
             <creation_date>
             <header_end_tag>

<body> --> (<begin_transaction> |
            <entry> |
                <end_transaction> |
                <comment>)*

        <begin_transaction> --> <begin_transaction_start_tag>
                                <transaction_number>
                                 <begin_transaction_end_tag>

        <entry> --> <begin_entry_tag>
                    <transaction_number>
                    <entry_base_addr>
                    <entry_length>
                    <entry_body>
                    <end_entry_tag>

        <end_transaction> --> <end_transaction_start_tag>
                              <transaction_number>
                               <end_transaction_end_tag>

        <comment> -> <begin_comment_tag>
                     <comment_length>
                      <comment_string>
                      <end_comment_tag>
```

In addition to the above grammar, there are semantic constraints on the journal file:

1) <begin_transaction> entries must appear in increasing transaction number order.
2) <end_transaction> entries must appear in increasing transaction number order.
3) For any given transaction number, the <begin_transaction> for that number must precede the <end_transaction> for that number. Note that the <end_transaction> will most likely not appear in the journal file if a crash occurs in the midst of the transaction.
4) <entry>s for any given transaction number must appear between the <begin_transaction> and the <end_transaction> for that transaction number.
5) <comment>s are intended to allow us to annotate the journal with text representations of the HDF5 API calls that generate the transactions.

Observe that the above semantic rules effectively disallow nested transactions. We may want to relax this constraint eventually, but it will make the initial implementation easier.

3 Library Code Changes

The library code changes required to implement journaling will mostly reside in the metadata cache, with ancillary code above and below that level. The code above and below the cache implies the changes needed in the cache proper. Therefore, we discuss the ancillary code first, followed by the metadata cache changes.

For simplicity, the bulk of the presentation focuses on changes for serial HDF5, which is the most common. Additional changes needed to support parallel HDF5 are covered at the end of the section.

3.1 Modifications above the Metadata Cache

There are several modifications above the metadata cache.

3.1.1 API additions

Working from the top down, the first change required is the definition and implementation of new and modified API calls allowing the user to:

1) Specify whether to journal metadata.
2) Specify where the journal will be kept – as a separate file or in the HDF5 file. While we will only support a journal stored in a separate file in the first implementation, we should go ahead and define the API for internal storage of the journal.
3) Select either synchronous or asynchronous metadata writes in the HDF5 library.
4) Select either synchronous or asynchronous journal file writes.
5) Select journal buffer and journal buffer ring buffer size. See section 3.3.1 for a discussion of these data structures.
6) Sync HDF5 file to disk and truncate the journal file. This will probably be implemented as a modification to the H5Fflush call.

3.1.2 Superblock modifications

Implement the superblock modifications as discussed in section 2.1 above.

3.1.3 Insert transaction start and end calls

Add calls telling the metadata cache to start and end transactions. When the metadata cache is told that a transaction is starting it will return a transaction ID. The returned transaction ID must be provided in any metadata cache entry unlock call that dirties the cache.

Initially we will place the start and end transaction calls at the beginning and end of HDF5 API calls that cause metadata to be modified. If the sizes of the resulting transactions are too large, we may change this to create smaller transactions. The "size" of a transaction refers to the number of metadata cache entries modified or inserted during the lifetime of the transaction.

3.2 Modifications below the Metadata Cache

The modifications below the metadata cache are more extensive than those above the metadata cache.

3.2.1 Journal entry creation callbacks

The metadata cache doesn't know how to serialize its entries and therefore uses callback routines to flush entries to disk. We will need similar callback routines to translate metadata cache entries into journal entries.

Alternatively, this may be a good time to move metadata I/O into the cache and use callbacks to serialize the metadata.

3.2.2 Add VFD calls for asynchronous metadata writes

To support asynchronous metadata I/O, we will have to add the necessary VFD (Virtual File Driver) calls and implement them in the MPI-IO, MPI-POSIX, and sec2 file drivers.

3.2.3 Journal entry write module

To simplify moving the journal into the HDF5 file should we ever do so, we will need to construct a module to handle journal writes. This module should handle opening the journal file, writing data to it, and truncating it. It must also support calls that report when a write has made it to disk. Also, it must be able to use either synchronous or asynchronous writes depending on configuration.

3.3 Modifications in the Metadata Cache Proper

The code modifications required in the metadata cache are fairly extensive. They are best described by listing the data structure changes and then outlining the algorithmic changes.

3.3.1 Metadata cache data structure changes

In terms of size, the major new data structure inside the metadata cache will be a ring buffer of journal buffers. The size of the ring buffer will be user configurable, but must be at least two: one ring buffer entry for storing journal entries as they are accumulated, and a second ring buffer entry for holding the last set of journal entries while they are being written to disk.

The size of journal buffers will be user configurable as well, although in the parallel case it must be large enough to hold the largest set of journal entries that can be accumulated in the period between sync points. This will be some multiple of the dirty data creation threshold used to trigger sync points. Refer to the later section on adaptations for the parallel case, section 3.4

Each journal buffer will have a number of fields associated with it, most particularly its ID (issued serially from the beginning of the computation), the ID of the last end transaction message that appears in the buffer, and its current and maximum size.

Further, the metadata cache will require new fields to maintain the ID of the last journal buffer successfully written to disk and the ID of the last transaction fully written to disk.

Also, each metadata cache entry will require two new fields: an integer field containing the ID of the last transaction in which it appeared and a boolean flag that is set to TRUE whenever an asynchronous write of that entry is in progress.

Finally, we will need some data structure to keep track of any journal or metadata writes that are in progress. The particulars of these data structures will have not been worked out at this time.

3.3.2 Metadata cache algorithmic changes

To ease implementation, the design makes as few changes to the metadata cache as possible. The algorithmic changes are outlined in the following sections.

3.3.2.1 Opening the file

On file open we must test to see if the contents of the superblock indicate that journaling was enabled. If it was, the file was not closed correctly, we must refuse to open the file, and we must tell the user to run the recovery tool.

The metadata cache must be informed if journaling is desired. If it is, the cache must:

1) Load the superblock, set the journaling flag, set the journal location field (for now this will always be the full path of the journal file), and then sync the superblock back out to file.

2) Open the journal file.
3) Set up synchronous or asynchronous metadata and/or journal file writes as selected by the user.
4) Set up its internal data structures to support journaling and then proceed with processing as before.

If journaling is not enabled, we must set up asynchronous metadata writes if desired by the user.

3.3.2.2 Marking transaction starts and ends

When the metadata cache is instructed to do so, it must insert transaction start and transaction end messages in the current journal buffer. These insertions may precipitate a journal buffer write as discussed in section 3.3.2.3 below.

When told to start a transaction, the metadata cache must assign the new transaction the next transaction number, insert a begin transaction message in the journal buffer with that transaction ID, and then return the ID to the caller.

When told to end a transaction, the metadata cache must:

1) Verify that the supplied transaction ID (call it N) is the next transaction ID to be closed.
2) Generate a journal entry for each entry that has been modified or inserted in the current transaction.
3) Insert an end transaction message in the journal buffer with the supplied ID.
4) Make note that transaction N has completed and that transaction N+1 must be the next transaction to terminate, and then return to the user.

To perform item 2) above, the close transaction call must scan the list of metadata entries dirtied or inserted in the current transaction, and perform the following processing:

1) Test to see if there is sufficient space in the current journal buffer for the newly dirtied entry. If there is, go to 5).
2) Start an asynchronous write of the current journal buffer. Let N be the ID of this journal buffer.
3) Test to see if there is an asynchronous write in progress on the next journal buffer in the journal ring buffer. If there is, stall until the write completes and then update the last journal and last complete transaction written to disk fields accordingly.
4) Set the ID of the next journal buffer in the journal ring buffer to N + 1. Set the current journal buffer to N + 1.
5) Construct a journal entry containing the current value of the dirtied entry. This entry will consist of the begin entry tag, followed by the associated transaction ID, followed by the file base address of the entry, followed by the length of its on-disk or serialized image, followed by the image itself, followed by the entry end tag.

   Note that there are a number of implementation hurdles here:

   a) For some metadata cache entries, the cache is not aware of the actual structure of the entry on disk.
   b) For all metadata entries, the cache does not know how to construct its on disk image.

At present, we have callbacks for cache entry flushes. As mentioned above, we must implement modified versions of these callbacks to construct journal entries.

6) Copy the journal entry into the current journal buffer. Update the current buffer size, and set the last transaction ID to that of the current transaction.

Observe that it is possible for transactions to be split across two or more journal buffers.

Also, note that it is possible that the act of serializing a journal entry will dirty another journal entry. We must generate journal entries for these cache entries as well before terminating the transaction.

Finally, if asynchronous journal writes are not enabled, we must modify the above accordingly.

3.3.2.3 Journaling dirtied / inserted entries

Whenever a metadata cache entry is inserted or unlocked with its dirtied flag set to TRUE, the new version of the entry must be journaled in the current transaction. We could generate the journal entry immediately, but that would result in unnecessary traffic to the journal file. Instead, we will maintain a list of metadata cache entries that have been modified in the current transaction, and then generate a journal entry for each cache entry on this list at the end of the transaction.

To do this, we proceed as follows:

1) Mark the metadata cache entry with the ID of the current transaction. Note that the transaction ID associated with the metadata change must be supplied with the unlock or insert call.
2) If the entry is not already on the list of entries modified in this transaction, append the entry to the end of that list. If it is already on the list, move it to the end of the list.

While it happens rarely, there is also the possibility that an entry will be discarded in an unlock. If it does happen and the entry is present in the list of entries modified in this transaction, we can simply remove the entry from the list and proceed as before.

3.3.2.4 **Making *space in the cache***

The protocol for scanning up the LRU list for entries to flush or evict when space is needed and/or when the maximum dirty percentage is exceeded remains almost unchanged, with only the following deltas:

1) A clean entry cannot be evicted if it has a write in progress. If such an entry is encountered, ignore it and go on to the next entry on the LRU list. *Rationale: We can't evict an entry whose write is in progress, as we don't know if the OS has copied the buffer containing the on-disk version of the entry yet.*
2) A dirty entry cannot be flushed unless the last transaction in which it appeared has been fully journaled on disk. If an entry is encountered that does not meet this criteria, ignore it, and go on to the next entry on the LRU list. *Rationale: If this is the first time this entry has been modified since the beginning of the current journal, and a write of this entry makes it to disk before a crash but the end of the associated transaction does not, then we will have a message from the future.*
3) If the transaction in which a dirty entry last appeared has been fully journaled (that is, the end transaction message for the transaction in which the entry was last dirtied has made it to

disk), post an asychronous write of the entry, mark it clean, mark it as having a write in progress, and move it to the head of the LRU list.

If asynchronous metadata writes, journaling, and/or asynchronous journal writes are not enabled, we must modify the above accordingly.

3.3.2.5 Housekeeping

Whenever we attempt to make space in the cache, and whenever we finish a transaction, we must perform the following housekeeping tasks:

1) Following "in issue order", test to see if there are any journal buffer writes that have completed. If the oldest journal buffer write has completed, update the last journal buffer written to disk and the last transaction fully written to disk fields accordingly, and then go on to the next journal buffer if one exists.

   If a journal buffer write is found to still be in progress, or if there are no journal buffer writes in progress, proceed to 2)

   Rationale: We retire journal buffer writes in issue order to ensure that all journal buffers with IDs less than or equal to the last journal ID written to disk are actually on disk. Similarly, this ensures that all transactions with IDs less than or equal to the last transaction fully written to disk are on disk.

2) If there are any pending metadata writes, test to see if any have completed. If any have, set the write in progress flags on the associated entries to FALSE.

If asynchronous metadata writes, journaling, and/or asynchronous journal writes are not enabled, we must modify the above accordingly.

Note: In previous versions of this section, these housekeeping tasks were to be performed on every entry into the cache. However, on reflection, the only times we care are when we are trying to make space in the cache, and when we are about to finish a transaction. These checks could be done more frequently, but there doesn't seem to be any reason to do so.

3.3.2.6 Flush

On a flush, we must write the current journal buffer to disk, flush all dirty entries, and then truncate the journal.

Note the rules for writing entries to disk outlined in 3.3.2.4 still apply. Thus we will have to stall as necessary while we wait for journal writes, then entry writes, and finally journal file truncation to complete.

3.3.2.7 File close

Flush as above. Then load the superblock, set the journaling tag to false, sync out the modified superblock, and then proceed as before.

3.4 Adaptations for Parallel HDF5

The above design requires relatively few adaptations for the parallel HDF5 case. The process n (n > 0) caches will operate as they currently do, with the addition of the maintenance of the list of entries inserted / dirtied in this transaction. The entries in this list must be serialized at the end of the transaction and at sync points (see below), so as to keep memory allocation in sync between the processes. With the possible exception of sanity checking, all other start/end transaction processing can be omitted on the process n (n > 0) caches.

Processing on process 0 at sync points will be as follows:

1) If a transaction is in progress, scan the list of entries inserted / modified in this transaction, generating journal entries, writing them to the journal buffer, and deleting entries from the list of entries inserted / modified in this transaction as we go.
2) Verify that the asynchronous write (if any) of the last journal buffer on the ring buffer has completed. Stall if it hasn't, and then update the last journal and last complete transaction written to disk fields accordingly.
3) Start an asynchronous write of the current journal buffer. Let N be the ID of this journal buffer.
4) Set the ID of the next journal buffer in the journal ring buffer to N + 1. Set the current journal buffer to N + 1.
5) If the cache is over size or below its minimum clean fraction, scan the LRU for entries to evict or flush. The rules given in 3.3.2.3 apply here when selecting candidates for eviction or flushing.

   Start an asynchronous write of any dirty entries selected for flushing, place these entries at the head of the LRU, mark them as clean, and as having a flush in progress.

   Do not add these entries to the cleaned list at this time, as we must wait until the entries are visible to the other processes to avoid the possibility of messages from the past.
6) Check to see if any metadata writes have completed since the last time we checked. If any have, set the flush in progress flags of the associated entries to FALSE, and add the entries to the cleaned list if and only if they are still clean.
7) Broadcast the cleaned list, and then truncate it.

As before, the above must be modified if asynchronous metadata writes, journaling, and/or asynchronous journal writes are disabled.

Housekeeping in the process 0 metadata cache is as described in 3.3.2.5, with the addition of a provision for inserting entries on the cleaned list if and only if they are still clean.

Journaling of dirtied entries on process 0 will be as described in Sections 3.3.2.2 and 3.3.2.3, with the following deltas:

1) The journal buffers must be sized so as to ensure that they cannot fill up between sync points.
2) When an entry is dirtied, we must check to see if it is on the cleaned list, and remove it from that list if it is present.
3) Since we generate journal entries for all entries in the list of entries inserted / modified in this transaction and truncate the list at sync points, it is possible that the list will be empty on transaction end. More importantly, the journal entries for any transaction that spans a sync point may contain one or more entries that are overwritten by subsequent journal entries for this transaction.

At present, we plan to attempt to make space in the cache only at the sync points, as this should result in posting larger numbers of metadata writes at the same time. We believe this approach should be good for efficiency.

If this proves not to be the case, we can let process 0 cache post writes as needed as per section 3.3.2.4, with the added constraint that entries cannot be written to disk unless they were dirtied prior to the last sync point. This is necessary to avoid messages from the future.

In closing, we observe there is no reason to post journal writes only at sync points. If we wished, we could simply post them as they fill. The posting of journal writes have been tied to the sync points in this design as a conceptual simplification.

3.5 The Recovery Tool

In its first incarnation the recovery tool will be very simple and will work as follows:

1) Open the target HDF5 file and attempt to find its superblock. If unsuccessful, the file cannot be recovered. Exit with a fatal error.
2) Examine the superblock to see if the journaling in use flag is set. If it isn't, exit with a message saying that there is nothing to do.
3) If the journaling in use flag is set, try to find the journal. At least initially, it will always be stored in an external journal file. If the user has supplied a journal file, use it. Otherwise try to find the journal file specified in the superblock. If unsuccessful, generate an error message and exit.
4) Open the journal file and validate it. If any syntactic or semantic errors are detected, issue an error message and exit. If the journal file is empty, we have nothing to do. Issue a message to this effect, and then go to 6).
5) Apply all the metadata writes specified in the journal entries up through the last completed transaction.
6) Reset the journaling in use flag in the superblock and flush the modified version of the HDF5 file to disk. Exit.

Future versions of the tool may perform more extensive checking on both the journal and the HDF5 file.

Acknowledgements

The design presented in this document resulted from a series of discussions with Quincey Koziel (THG), Lee Ward (Sandia National Labs), and Rob Ross (Argonne National Labs). Their contributions are greatly appreciated.

Ruth Aydt (THG) assisted with the final editing of the document.

Revision History

August 28, 2007: Version 1 posted for public comment. Comments should be sent to help@hdfgroup.org.

August 22, 2008: Version 2 posted. Slight formatting changes to conform to updated style. Updated version number but not base RFC number.

Appendix: Background Material

This appendix contains background material, including assumptions and high-level design decisions, which guided the final approach presented in the body of the RFC.

Underlying File System Behavior

We plan to maintain at least the option of writing both metadata and journal entries asynchronously. For this to work with a separate journal file, as is the plan for at least the initial implementation, we presume the following:

For the HDF5 file:

1) The file system handles a mix of non-overlapping synchronous and asynchronous writes and synchronous reads gracefully.
2) There is some well-defined way of ensuring that all outstanding writes have made it to disk. Typically this will require a call to fsync() or equivalent after all pending async writes have completed. However it would be useful to identify other options.
3) There is some well-defined point after which an asynchronous write will be visible to a read.

   At least in the case of POSIX file systems, the standard states that any synchronous write posted prior to a read will be visible to the read. The behavior in the case of asynchronous writes is not clearly stated. We believe asynchronous writes will be visible to reads after they complete, but have yet to confirm this behavior is guaranteed.

4) Overlapping synchronous and asynchronous writes (by which we mean that one write to a given location starts before an earlier write to the same location completes) are not necessary, and do not occur at present.

   In the unlikely case that the underlying file system(s) handle this case in a predicable way, we need to know so as to be able to exploit this fact should it prove useful.

   In the more likely case that the file system does not handle overlapping synchronous and asychronous writes predictably, we need to know that as well, so that we can be sure flag an error should the condition ever occur.

For the journal file:

5) The file system handles a sequence of non-overlapping asynchronous writes gracefully.
6) There is some well defined point after which asynchronous writes are guaranteed to have made it to disk. It would be convenient if this would be the case after the write completed (perhaps open the file with the O_SYNC flag?), but any well defined point should do (albeit at the cost of additional overhead).
7) Whenever the user syncs the HDF5 file, we will truncate the journal file after all outstanding metadata modifications have been written to the HDF5 file and have made it to disk.

   We need to know when the truncate is guaranteed to make it to disk. Either after the truncate returns, or after a sync is fine, but we need to know which.

For the parallel case, we have one additional assumption:

8) There is some point at which an asynchronous write executed on one node is guaranteed to be visible to all other nodes. Presumably this will be when the asynchronous write completes, but we must verify this.

Finally, when / if we put the journal in the HDF5 file proper, we assume that:

9) There is some way of syncing out journal writes to disk without syncing out the rest of the HDF5 file. This may not be possible on most Unix systems.

Where to Store the Journal

The preferred option was to store the journal in the HDF5 file. However, that does not seem practical for most versions of Unix.

Given that most of our target operating systems are some flavor of UNIX, this led immediately to the decision to store the journal in an external file to begin with, with the option of moving it into the HDF5 file when there is both demand and resources for the move.

We will design with the possibility of such a move in mind, and accept the fact that storing the journal in a separate file opens the possibility for the journal file to be "misplaced" by users.

Simple First Implementation

The first cut at crash proofing as simple as possible. The cost of this decision will likely be seen in performance that is not what it could be. The benefit should be a more straight-forward initial development effort.