RFC: Metadata Cache Logging

Dana Robinson

As an aid for debugging, a mechanism for logging metadata cache operations will be added to the library. This functionality is motivated by the new single-writer/multiple-reader (SWMR) feature, which is likely to be difficult to debug due to the asynchronous nature of the feature and the addition of metadata flush dependencies. This document describes the new functionality as well as the log format, which is based on JSON.

This RFC is intended for advanced users, particularly users of the SWMR feature, and HDF Group library developers. The logging feature will appear in the future HDF5 1.10.0 release, but could also be added to the 1.8 branch, if desired.

# Introduction

The metadata cache is a central feature of the HDF5 library, though which all *file metadata* read and write operations take place. The metadata stored in this cache is for internal use only and is not exposed to the user. It is used by the HDF5 library to locate and characterize HDF5 objects and data. Each open file has its own metadata cache and caches are not shared among processes. File metadata should not be confused with *user metadata*, which is stored by the user as attributes attached to HDF5 objects, primarily via H5A\* API calls.

HDF5 1.10 will support the single-writer/multiple-readers (SWMR) data access pattern, which will allow multiple reader processes to access an HDF5 file that is being written to by a single writer process with no inter-process communication (IPC) required. Support for this feature requires the writer process to order metadata writes to storage so that reader processes will not encounter an invalid HDF5 file. This could happen, for example, if the writer wrote a piece of metadata to storage that targets a piece of metadata that only exists in the writer's cache (i.e., has not been propagated to storage yet). When the reader attempted to load the targeted metadata, it would find garbage, causing an error.

Due to the lack of communication between the processes, the SWMR data access pattern is inherently asynchronous and bugs are expected to be difficult to troubleshoot due to the lack of deterministic reproducibility. Since SWMR bugs will most likely involve the metadata cache at some level, a log of the cache operations would be very useful in debugging the feature, especially when the error conditions are uncommon or only occur on particular hardware.

In addition its use as a diagnostic aid for the SWMR feature, this feature would also be useful for troubleshooting general metadata cache problems or performance issues.

# Use Cases

The primary use case for this feature is diagnosing metadata cache bugs reported by SWMR users. The most important of these is expected to be broken flush dependencies. A secondary use case is tracking cache activity over time, which may be useful for diagnosing metadata cache bugs and performance issues.

## Detecting Broken Flush Dependencies

The logging functionality could be used to detect broken flush dependencies. A Python program could be used to inspect the output of each flush to ensure that no parents were flushed before their children.

## Monitoring Cache Activity

The logging functionality could also be used to monitor cache usage parameters. This would be especially useful when making use of the cache/object flush control routines.

# Enabling and Controlling the Feature

The feature will be turned off by default. It will be enabled by using the new H5Pset\_mdc\_logging functions to modify the file access property list used to open or create a file. A Boolean flag parameter of this function will determine if logging begins at file open/create. Additionally, two other new functions – H5begin/end\_mdc\_logging – will enable logging to be switched on and off as needed. Each call to will begin by dumping the current cache contents and status.

It is assumed that the logging framework overhead will be minimal[[1]](#footnote-1) when logging is switched off and thus the feature does not warrant a compile-time build option.

# New HDF5 API Functions

## H5Pset\_mdc\_logging

*herr\_t* H5Pset\_mdc\_logging(hid\_t fapl\_id, *char* \*location, *hbool\_t* start\_log\_on\_open)

*hid\_t* fapl\_id IN: file access property list ID

*char* \*location IN: location of log (file path/name)

*hbool\_t* start\_log\_on\_open IN: whether the logging will begin as soon as the file is opened or created

This function will set the logging parameters in a file access property list.

The location parameter will be a simple file path/name but may be expanded to include URLs in the future. There will be no default file name, so the location parameter must specify a file name and not a directory. The default location for the log will be the current working directory.

The start\_log\_on\_open flag will determine whether or not logging will begin on file open/create. This, combined with the begin/end functions, would allow people to selectively log troublesome areas of their code, potentially decreasing running time and keeping log files smaller and more manageable.

There is currently no plan to add a file identifier to the log messages, so it normally won't be possible to send log messages from more than one cache to the same log location.

An option for the future would be to add a bitwise flag parameter that would be used to determine which types of messages are of interest (e.g., flush dependencies). If this proved to be of use, it could be added while the SWMR feature is being developed (before the official HDF5 1.10 release).

Another option for the future would be to add a parameter that would control how often cache statistics were emitted.

## H5Pget\_mdc\_logging

*herr\_t* H5Pget\_mdc\_logging(hid\_t fapl\_id, *hbool\_t* \*is\_enabled, *char* \*location, *hbool\_t* \*start\_log\_on\_open)

*hid\_t* fapl\_id IN: file access property list ID

*hbool\_t* \*is\_enabled OUT: whether the logging is enabled

*char* \*location OUT: location of log (just a file path/name for now)

*hbool\_t* \*start\_log\_on\_open OUT: whether the logging begins as soon as the file is opened or created

This function gets the current status of the logging (enabled/disabled), whether the logging begins at file open/create, and the location (file/path name) of the log file.

## H5begin\_mdc\_logging

*herr\_t* H5begin\_mdc\_logging(*hid\_t* file\_id)

*hid\_t* file\_id IN: HDF5 file identifier on which to begin logging metadata operations.

Opens the log file and starts logging metadata cache operations for a particular file. It will dump the current contents of the cache (including flush dependencies, if any) to the log when it is called so the state will be known. Calling this function when logging has already been enabled will be considered an error.

## H5end\_mdc\_logging

*herr\_t* H5end\_mdc\_logging(*hid\_t* file\_id)

*hid\_t* file\_id IN: HDF5 file identifier on which to end logging metadata operations.

Stops logging metadata cache operations for a particular file and closes the log file. Calling this function when logging is not enabled will be considered an error.

# Log Messages

## Log Format

The log is emitted using JSON notation (a schema can be found in the appendices of this document). The entire log is a valid JSON object consisting of the file name and an array of JSON-formatted log messages.

{

"file": "<file path (string)>",

"messages":

[

<log message 1 (as described below) (object)>,

<log message 2 (object)>,

…

<log message n (object)>

]

}

JSON was selected due to its ability to handle rich data and ubiquity, especially with dynamic analysis languages (e.g., Python) and display libraries. Simple event-based log formats might be easier for humans to read, but would be less able to present rich data for more in-depth analysis.

Other log formats and/or libraries were considered, but none met our needs for a simple, yet expressive format combined with a well-supported, platform-independent, appropriately licensed library with a C API. Two libraries deserve mention, however:

SLOG (<http://www.mcs.anl.gov/research/projects/perfvis/software/log_format/>) is a part of MPE and might be interesting for viewing process activity as a function of time. It's not clear if the library is suitable for this purpose (SWMR does not pass messages, for example) and the problem of time skew between separate machines might be troublesome.

Pantheios (<http://www.pantheios.org/>) is a platform-independent logging library that might be considered for the future, however it was decided to not add this dependency into the library code.

## Log Messages

Each JSON message consists of a timestamp, a string describing the action being recorded, and a JSON object that contains any auxiliary data required (offsets in the file, state transitions, etc.). Times in the log file are always recorded in POSIX time (i.e., # of seconds since epoch). The action strings for each log entry type are described below, along with a description of the value object.

{

"time": "<POSIX/Unix timestamp (int)>",

"action": "<e.g. 'pin' or 'flush' (string)>",

"value": "<additional data as described below (object)>"

}

## Cache Entries

The generic format for a metadata cache entry includes the offset of the entry in the file, the size of the entry in bytes, the type of the entry (e.g., B-tree node, file superblock), and the entry's tag.

{

"offset": "<int>",

"size": "<int>",

"type": "<string>",

"tag": "<int>"

}

## Log Begin/End

This message is emitted whenever logging is started or stopped.

{

"time" = <int>

"action": "logging"

"value":

{

"state": <boolean>

}

}

## SWMR Begin/End

This message is emitted whenever the SWMR feature is enabled or disabled.

{

"time": <int>

"action" = "swmr"

"value" =

{

"state": <boolean>

}

}

## Load Entry

This message is emitted when a cache entry is loaded from the file.

{

"time" = <int>

"action" = "load"

"value" =

{

"offset": "<int>",

"size": "<int>",

"type": "<string>",

"tag": "<int>"

}

}

## Flush Entry

This message is emitted when a dirty cache entry is flushed from the file.

{

"time" = <int>

"action" = "flush"

"value" =

{

"hygiene":

}

}

## Evict Entry

This message is emitted when a clean cache entry is evicted from the cache.

{

"time" = <int>

"action" = "evict"

"value" =

{

"hygiene": <string>, ("clean" | "dirty")

}

}

## Create/Destroy Flush Dependency

This message is emitted when a flush dependency is created or destroyed.

{

"time" = <int>

"action" = "depend"

"value" =

{

"state": <boolean>,

"parent": <int>, (offset)

"child": <int> (offset)

}

}

## Enable/Disable Flushes for a Cache/Object

This message is emitted when flushes for either the file's cache or particular objects in the cache are enabled or disabled (via the H5F/H5Oenable/disable\_mdc\_flushes functions).

{

"time" = <int>

"action" = "flush\_control"

"value" =

{

"state": <boolean>,

"scope": <string>, ("cache" | "object")

"object\_id": <int>

}

}

## Resize Cache

This message is emitted when an object is resized in the cache.

{

"time" = <int>

"action" = "resize"

"value" =

{

"old": <int>, (size in kB)

"new": <int>

}

}

## Pin/Unpin

This message is emitted when an object is pinned or unpinned in the cache.

{

"time" = <int>

"action" = "pin"

"value" =

{

"state": <boolean>,

"location": <object>

}

}

## Protect/Unprotect

This message is emitted when an object is protected or unprotected in the cache.

{

"time" = <int>

"action" = "protect"

"value" =

{

"state": <boolean>,

"location": <object>

}

}

## Delete

This message is emitted when an object is deleted from the cache.

{

"time" = <int>

"action" = "delete"

"value" =

{

"state": <boolean>,

"location": <object>

}

}

## Rename

{

"time" = <int>

"action" = "rename"

"value" =

{

"state": <boolean>,

"location": <object>

}

}

# Testing the Feature

## Unit Testing

A new test program (cache\_logging(.c)) will be added to the test/ directory. This will be a fairly simple test program that will ensure that the setup and control functions work and ensure that the cache generates appropriate log messages.

## Acceptance Testing

A new shell script (mdc\_logging.sh) will be added to the test/ directory. This will run a separate test program (mdc\_logging(.c)) that will turn cache logging on and then perform a set of operations that will exercise the cache. Since the operation of the cache is deterministic, the output log(s) will simply be compared (using diff) with a canonical log file. The mdc\_logging program will exercise all data structures and will have

# Acknowledgements

This work was supported by a customer of The HDF Group.

# Revision History

|  |  |
| --- | --- |
| *February 24, 2014:* | Version 1 circulated for comment within The HDF Group SWMR team. |

# Appendix: H5Pset\_mdc\_logging RM Entry

Will be added in version 2.

# Appendix: H5Pget\_mdc\_logging RM Entry

Will be added in version 2.

# Appendix: H5begin\_mdc\_logging RM Entry

Will be added in version 2.

# Appendix: H5end\_mdc\_logging RM Entry

Will be added in version 2.

# Glossary, Terminology

**cache entry** An item that is stored in the metadata cache. An HDF5 object will often be represented by multiple cache entries. As an example, each node in a B-tree index is represented as a separate cache entry.

**file metadata** Metadata that describes the internal structure of the file. Created by the HDF5 library and largely invisible to users.

**HDF5 object** A "thing" stored in HDF5 storage. Includes datasets, groups, and named datatypes. Note that attributes are not considered HDF5 objects in their own right, but instead are considered a part of the object to which they are attached.

**user metadata** Attributes created by the user that are attached to datasets, groups, or named datatypes.

1. A single if(logging){ } statement would be executed in each file-oriented cache operation (loads, flushes, etc.). In the context of a largely I/O-bound library, this is trivial overhead. [↑](#footnote-ref-1)