RFC: Write-Ahead Log

Neil Fortner
Matt Larson
Vailin Choi
Glenn Song

This RFC describes a new module that implements a Write-Ahead Log (WAL) for metadata within the HDF5 library. In the event of a crash during the lifetime of an application using HDF5, this WAL can be used by the library or an external tool to restore the metadata within an HDF5 file to a self-consistent state, preventing file corruption and limiting the loss of application data with minimal I/O and memory overhead.

Revision History

Version Number	Date	Comments
v1	April 22, 2024	Version 1 released to HDF5 community.

Contents

Introduction

At present, the burden for crash prevention and recovery when using HDF5 lies entirely on application developers. The library itself has no provisions for crash recovery or backup creation. Considering the potential scale of data loss, the materials or machinery that may be required to replicate physical experiments, and the cost of compute time, the development of a native mechanism for crash recovery would be highly valuable. To limit complexity of design and I/O overhead, and because the semantics of when raw data is written to the file are already well defined, this proposal's goal is limited to restoring metadata and preventing overall file corruption.

The WAL enables crash recovery through a page-oriented replay log. This log will store enough information about metadata operations that, in the event of a crash, these operations may be replayed to restore the file's metadata to a confirmed self-consistent state. The log is page-oriented in the sense that it describes the metadata operations in terms of blocks of metadata at offsets within the file, rather than in terms of HDF5 file objects. During operation, the library will set markers in the WAL to indicate points at which the file is known to have self-consistent metadata. The WAL is an external file that may be trimmed at regular intervals to keep its size reasonable.

Previous efforts to address metadata crash-recovery through Metadata Journaling [2] had significant performance issues, leading to the decision to discontinue pursuit of that approach. There have also been efforts to implement full SWMR (Single Writer/Multiple Reader) support which could be extended to support crashproofing, but we feel that users that do not need SWMR access will appreciate a more lightweight and maintainable solution.

1 Architecture

1.1 Metadata Cache vs. Page Buffer

As the WAL requires a client module that is responsible for handling metadata I/O to the file, it could be feasibly achieved with either the metadata cache or the page buffer as its primary client. For several reasons, we chose to pursue implementation via the metadata cache:

- An implementation through the page buffer would necessitate either tracking which page sections are updated by writes, or writing exclusively entire metadata pages to the WAL. Tracking page sections would increase complexity greatly, and writing entire pages may negatively impact performance and increase the size of the WAL.
- The page buffer is often bypassed - for example, when an I/O operation is larger than one page. Implementing the WAL through the page buffer would require forcing all metadata to always pass through the page buffer.
- An implementation through the page buffer would still need to notify the metadata cache at the time of a log flush or log checkpoint (see Flush Types) in order to accurately satisfy later metadata reads. Implementing the WAL through the metadata cache means that this introduces no additional coupling.

1.2 Flush Types

There are two types of metadata flush that the library may perform:

1. A ”log flush” - Flushes metadata from the cache to the WAL file to create a self-consistent location to play forward to in the event of a crash.
2. A ”log checkpoint” - Replays metadata writes from the WAL to the HDF5 file to bring the file up-to-date with metadata operations until a specific point. This requires reading a WAL entry from the WAL file for each metadata write performed. This is the type of flush invoked by `H5Fflush()`.

Each type of flush operation leaves a 'marker' in the WAL after the last WAL entry upon which it acts.

All WAL entries before a log checkpoint marker have already been written to the HDF5 file and may be safely trimmed from the WAL. A WAL trim operation involves truncating the WAL file so it only contains the WAL superblock, effectively deleting all WAL entries.

A complete implementation of the WAL without log checkpoint markers is possible, if the WAL is always trimmed immediately after each log checkpoint. In this case, all entries which would be placed before a log checkpoint marker are instead removed from the WAL entirely, and so the marker is not necessary to indicate that these entries should be skipped during subsequent WAL operations.

A log flush marker indicates that all WAL entries since the last log checkpoint should be 'played forward' to the HDF5 file during a log checkpoint or crash recovery. Log flush markers prevent re-execution of torn metadata writes, since if a torn write occurs during a log flush and leaves a malformed WAL entry, no log flush marker will be present and no modification of the HDF5 file will be attempted.

1.3 Metadata Cache + WAL Lifecycle

In order to avoid high overhead from constant small batches of I/O, log checkpoints are performed at intermittent intervals based on either the passage of real time via a 'tick' system similar to the one proposed in the VFD SWMR RFC [1], or based on the quantity of metadata written to the file. The specific criteria for and frequency of log checkpoints during library operation is configurable by the application via property lists. Due to the partition of information between different layers of the library, specific user-provided values for intervals at which these operations act as heuristics rather than exact thresholds.

I/O to the WAL file uses the active VFD for the file, unless a different VFD is provided at the time the WAL is enabled by `H5P_set_wal()`.

1.3.1 Metadata Writes

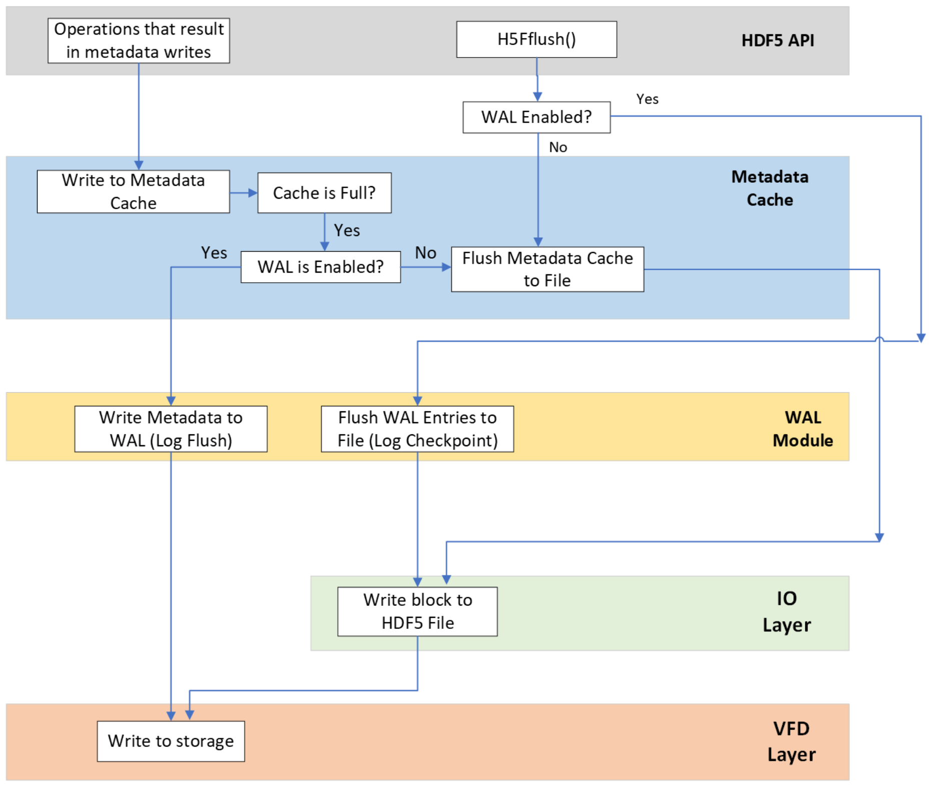


Figure 1 – Overview of different types of metadata writes

Figure 1 illustrates how the WAL alters the existing flow of metadata writes through the library. Writes which do not prompt a flush of the metadata cache have no special handling with the WAL enabled. Once a metadata write prompts a cache flush, it will be performed as a log flush which writes metadata to the WAL file and leaves stub entries in the metadata cache. The metadata in WAL entries is not written to the HDF5 file until a log checkpoint, which is either directly invoked by `H5Fflush()` or invoked each time a user-defined threshold is reached.

1.3.2 Metadata Reads

The MDC must be able to read back WAL entries in order to correctly service read operations when the WAL contains a more recent version of a metadata object than the file (e.g. after a log flush but before a log checkpoint). When a dirty object is evicted from the cache, a 'stub' entry is left in the cache. This stub entry records the address of the WAL entry within the WAL file so that it may be quickly read at a later time.

The full process for a metadata read is as follows:

- If the requested metadata object is not in the metadata cache, fulfill the read from the file.
- If the requested metadata object has a non-stub entry in the metadata cache, fulfill the read from the metadata cache.
- If the requested metadata object has a stub entry in the metadata cache, fulfill the read from the WAL entry.

1.4 Metadata Deletion and Raw Data Integrity

Consider the following series of events on a file with the WAL enabled, as illustrated in Figure 2:

1. A user operation results in a metadata write to offset `A` of length `N` within the file. This write is logged to the WAL.
2. A user operation results in the deletion of a section of metadata, `A*` to `A* + N*`, that resides entirely or partially within the address region `A` to `A + N`.
3. Due to free space recycling being enabled on the file, the region `A*` to `A* + N*` is marked as free for other usage within the library.

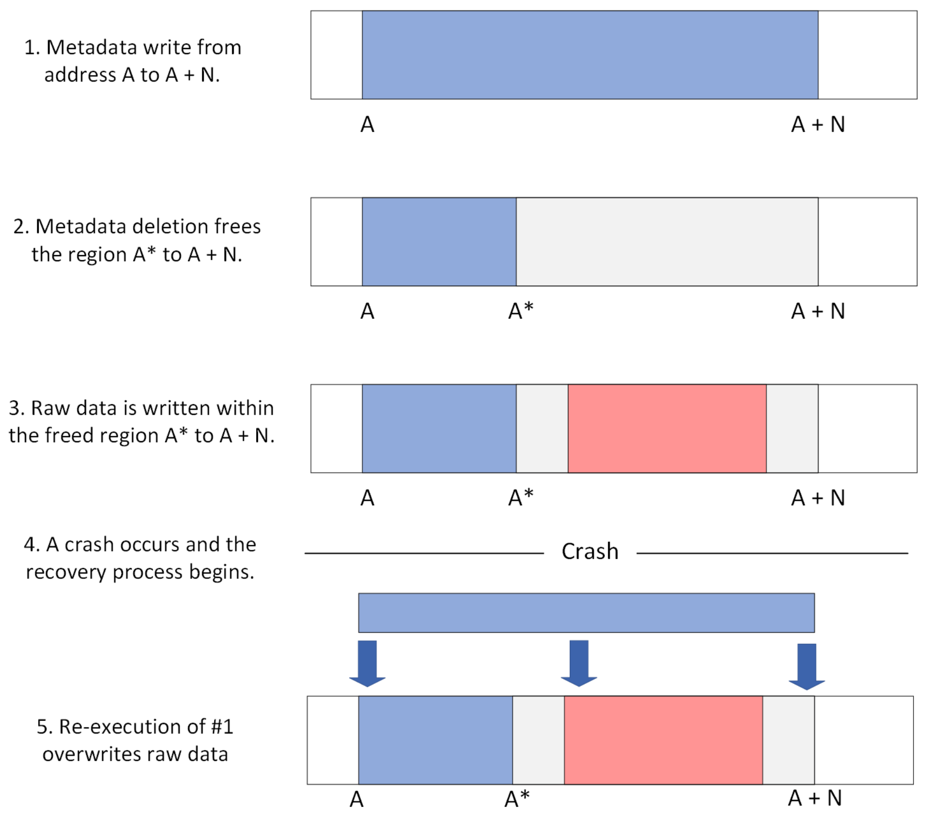

1. Metadata write from address A to A + N.
A
A + N
2. Metadata deletion frees the region A* to A + N.
A
A*
A + N
3. Raw data is written within the freed region A* to A + N.
A
A*
A + N
4. A crash occurs and the recovery process begins.
Crash
5. Re-execution of #1 overwrites raw data
A
A*
A + N

Figure 2 – A series of WAL operations that would result in raw data being overwritten during recovery

4. A user operation results in raw data being written to the file in the (now free) region `A*` to `A* + N*`. Because this is a raw data operation, it bypasses the WAL.
5. Before the next log checkpoint, a crash occurs.
6. Crash recovery begins, and eventually executes the metadata write from event #1. This results in the overwriting of whatever portion of the user's raw data was in the filespace range marked as free by event #3, from `A*` to address `A* + N*`.

Strictly speaking, raw data integrity is outside the scope of the Write-Ahead Log, but it is still preferable to preserve the integrity of application data as much as possible in the event of a crash.

To prevent this potential loss of raw data, the preferred option is to defer recycling free space for metadata until the next log checkpoint. Metadata generally takes up a relatively small amount of space in the file compared to raw data, so the potential gains from optimizing space usage in this case are limited, especially given the algorithmic overhead required to enable it.

If there exists a use case for which non-deferred metadata free space recycling is critical, the following is an outline of the changes necessary to support simultaneous raw data integrity and non-deferred metadata free space recycling:

The WAL entry file format would need to be expanded to support dividing a metadata block into isolated sections. This is necessary because repeated metadata deletions could divide a previously contiguous block of metadata into an arbitrary number of distinct sections. Tracked information would need to include the quantity of sections, their lengths, and their offsets. In order to keep each WAL entry header a fixed size (to avoid fracturing writes), there would need to be a fixed upper bound on the potential number of sections within a single entry.

A rough sketch of an algorithm to track these metadata sections within WAL entries is as follows:

1. At the time of a metadata deletion, search the metadata cache for any metadata writes which overlap the deleted region and have not been written

to the file. The metadata cache contains stub entries with the length and offset of the metadata, which allow this search to be done without a disk read.

2. If any of the WAL entries with overlapping writes already have more than the maximum number of distinct sections, then trigger a log checkpoint of the WAL. After this, no previous metadata writes will need to be re-executed, so this process ends.
3. For each WAL entry with a metadata block that overlaps the deleted metadata block, add one or more additional sections:
   a) Determine which of its existing sections overlap the deleted region.
   b) For each section that overlaps the deletion, decrease its length so that no valid sections of the metadata section overlap the deleted region.
   c) If the deletion results in the creation of a new section within the block (e.g. if the deletion overlaps with a section, but not the end of that section), compute the length and start address of the new section.
   d) If this process would result in the creation of enough new sections in the metadata block to push the WAL entry over the maximum number of entries, then perform a log checkpoint instead.

1.4.1 WAL Lifecycle

The WAL file will be created and its superblock initialized at file open time if the WAL is enabled on the file access property list (FAPL) for the file open/create operation. The WAL will be deleted when the file is closed normally, or after recovery if the file was reopened with the WAL disabled.

1.5 Compatibility with Other Library Features

1.5.1 Parallel HDF5

WAL compatibility with Parallel HDF5 could be achieved by designating a single process as the WAL writer, and having all I/O to the WAL be performed through that process.

Due to how parallel HDF5 handles each rank's metadata cache, additional synchronization points between the ranks will not be necessary for WAL compatibility. Because metadata writes must be done collectively, each rank will have the same dirty metadata cache entries to be flushed at the same time. Each rank will create a stub entry containing the target address in the WAL file, but only the designated writer will actually perform the write to the WAL file. Each rank will have up-to-date read access to the WAL file without additional work or I/O.

Because each process needs to be able to read directly from the WAL, the WAL file must be exposed to all processes in a similar manner to the HDF5 file itself. In order to maintain consistency, there will be an additional requirement that all ranks use the same log flush and log checkpoint intervals, and these intervals must be based on bytes written. In this way every process will have the same understanding of when log flushes and log checkpoints occur, and thereby will know when to eliminate stub entries.

1.5.2 SWMR (Single Writer/Multiple Reader)

There are multiple ways to enable SWMR access with crashproofing enabled. Perhaps the simplest method would be for the SWMR readers to ignore the WAL file and only look at the HDF5 file. This would of course prevent them from seeing any updates until a log checkpoint. Correct ordering of writes when replaying the WAL during the log checkpoint should happen naturally since the existing SWMR mechanism will have correctly ordered the writes in the WAL file, and they are replayed in that order.

Alternatively, we may wish for the readers to be aware of the WAL file and scan it for updates to metadata. They could either only look at WAL entries up to the latest log flush marker, or look at all entries. In either case we would want to be careful to handle the case where a log checkpoint occurs during a SWMR read operation. The need to avoid caching information about the location of metadata in the WAL would necessitate scanning the WAL frequently, which may make the first approach more feasible.

1.5.3 Page Buffer

Because the WAL largely replaces metadata writes to the HDF5 file with writes to the WAL file, the page buffer is less relevant than when not using the WAL. That said, we may be able to achieve a similar effect on writes by, when flushing multiple entries to the WAL (at log flush time or otherwise), assembling WAL entries into a single buffer before writing them to the WAL in a single operation. It should also be possible to use the page buffer only at log checkpoint time to perform metadata writes in larger blocks, though this will only benefit the log checkpoint and file close operations.

In the future it will be possible to implement a page buffer like scheme for the WAL itself, which will aid in prefetching entries that exist only in the WAL, but this will not be in the initial implementation.

The standard HDF5 page buffer will still be available for use with raw data when the WAL is active, since the WAL does not modify the I/O pattern for raw data.

1.5.4 Metadata Accumulators

Metadata accumulators sit below the metadata cache and group small metadata I/O requests into larger batches to improve performance. Simultaneous use of the metadata accumulators and the WAL is not possible with the current design.

If an accumulator worked simultaneously with the WAL, then what the metadata cache considers distinct writes to the WAL file from distinct cache entries could be grouped into a single I/O request at the VFD layer. This would write a single

WAL entry with a metadata block that contains metadata blocks from distinct metadata cache entries. At metadata read time, if a stub entry in the cache is used to read metadata from the WAL file, the metadata block retrieved would be larger than expected. While it would contain the requested metadata block, the metadata cache lacks the capabilities to perform introspection on the accumulated metadata block to determine which region has the requested data.

As log checkpoints involve no writes to the WAL file, metadata accumulators may be used during log checkpoint operations, with the constraint that the accumulator be flushed afterwards. Since this would provide little to no opportunity for the accumulator to bundle I/O requests, the potential usefulness of enabling the accumulators for this case is debatable.

1.5.5 Variable-Length and Reference Datatypes

Variable-length and reference datatypes require special consideration, since the ’raw data’ for these types has a metadata component. Data with a variable-length or reference datatype is written to file as global heap IDs which point to the location of the actual data within the file’s global heap (i.e. within the metadata portion of the file).

As the metadata cache does not draw a distinction between metadata for these types and ’regular’ metadata, the most straightforward solution is to treat these metadata operations in the same manner as other metadata writes.

Because the amount of data in variable-length and reference datasets is likely to be much higher than other kinds of metadata, using these datatypes extensively will likely increase the I/O and storage overhead of the WAL. Therefore, we may decide to add an option to exclude these types of objects from the WAL and remove consistency guarantees for variable length and reference data.

1.5.6 Compact Datasets

Compact datasets store the dataset’s raw data within the object header. Just like variable-length and reference datatypes, this results in what the application views as raw data being stored and retrieved as metadata. As in that case, because the semantic meaning of the metadata block is not known to the metadata cache, the WAL will treat metadata describing a compact dataset in the same manner as all other metadata.

Unlike for variable-length and reference datatypes, the size of compact datasets is strictly bounded, so it is unlikely that any option to disable the WAL for compact datasets will be added.

1.6 WAL File Format

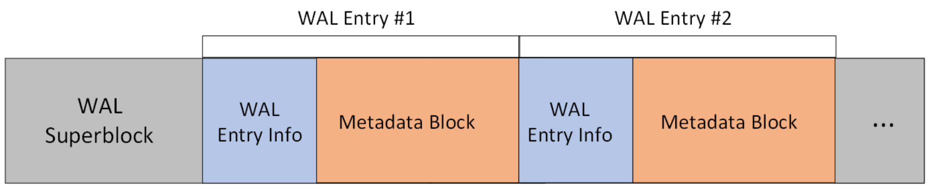


Figure 3 – The on-disk layout of a WAL file

The WAL file is composed of the WAL superblock and a contiguous series of WAL entries. A WAL entry consists of a WAL entry info block and a metadata block.

Reading an arbitrary WAL entry requires iterating through all prior WAL entries, but this is only done at log checkpoint and recovery time. Reads during log flushes are expedited by using stub entries in the metadata cache to jump directly to the address of the desired WAL entry. The speed of recovery is not a critical concern, as recovery is expected to be infrequent.

The WAL superblock consists of the following elements:

- `version` - The version of the WAL file format in use.
- `filepath` - Filepath indicating the HDF5 file that this WAL corresponds to.

1.6.1 WAL Entries

Each WAL entry consists of a WAL entry info block, followed by the block of metadata for the logged metadata write. The fields in the WAL entry info block are as follows:

- `target_offset` - The target offset in the HDF5 file for the metadata write operation. This field has two 'magic values' which indicate this this WAL entry is a marker. A log flush marker indicates that all previous entries are safe to be replayed in the event of a crash. A log checkpoint marker indicates that all previous entries have already been flushed to the file and do not need to be replayed. As mentioned earlier, we may eliminate log checkpoint markers in favor of always trimming the WAL on a log checkpoint, in which case the start of the WAL functions as an implicit log checkpoint marker. Markers do not contain any of the other fields listed below. Markers are stored in this fashion to avoid allocating space in each entry for a field that the majority of WAL entries will not use.
- `length` - The size in bytes of this entry's metadata block.

We may also decide to add a checksum for the WAL entry info, or the metadata block, or both, though the metadata block may already be checksummed using existing HDF5 facilities so that may not be necessary.

WAL entries do not need start or end tags to avoid executing WAL entries that result from torn writes, because interrupted WAL entries are already handled by log flush markers. During recovery, the WAL is only played back until the most recent log flush marker. If the library terminates unexpectedly, any torn writes to the WAL will be after the most recent log flush marker, and thus will not be re-executed.

Most database implementations of a Write-Ahead-Log uniquely assign each log entry a monotonically increasing log sequence number. At the present stage, the HDF5 WAL would have no use for such a feature beyond potential consistency checks. The HDF5 WAL is not currently planned to support multiple simultaneous writers, removing a large motivation for the log sequence number. Finally, because the WAL file format is not intended to be future-proof in the same manner as the HDF5 file format, adding a log sequence number in a subsequent version if the need arises will be relatively painless.

This design places the marker for log flushes/log checkpoints within individual WAL entries via a magic value in the 'offset' field. Another potential option is to store the marker in the WAL superblock as the offset of the latest WAL entry that has been log flushed/log checkpointed. This would have the advantage of improving recovery time, because storing markers in individual entries requires a preliminary scan of the entire WAL to determine which entries to replay. However, this would incur a performance penalty by adding a WAL header write to log flushes and log checkpoints.

1.7 Additions to the API

- `herr_t H5Pset_wal(hid_t fapl_id,`
  `const char *wal_path,`
  `H5WLinterval_type_t log_flush_interval_type,`
  `size_t log_flush_interval,`
  `H5WLinterval_type_t log_checkpoint_interval_type,`
  `size_t log_checkpoint_interval,`
  `hid_t wal_fapl_id)`
  Modifies the file access property list `fapl_id` to log metadata writes to a WAL file at `wal_path`. `log_flush_interval_type` determines which metric is used to initiate periodic log flush execution (bytes written, time, or none) and the value for that threshold is `log_flush_interval`. `log_checkpoint_interval_type` determines which metric is used to initiate periodic log checkpoint execution (bytes written, time, or none) and the value for that threshold is `log_checkpoint_interval`. `wal_fapl_id` is a file access property list which is used to determine which VFD is used to perform I/O to the WAL file. If this parameter is `H5I_INVALID_HID`, the active VFD from `fapl_id` is used for WAL file I/O.

- `herr_t H5Pget_wal(hid_t fapl_id, const char **wal_path, H5WLinterval_type_t *log_flush_interval_type, size_t *log_flush_interval, H5WLinterval_type_t *log_checkpoint_interval_type, size_t *log_checkpoint_interval, hid_t *wal_fapl_id)`
  Retrieves whether a given file access property list has the WAL enabled, the path to the WAL file if so, and the thresholds types and values for when log checkpoints are performed. If `*wal_path` is returned as NULL then the WAL is disabled for the file.
- `herr_t H5Pset_wal_auto_recovery((hid_t fapl_id, bool enable_auto_recovery)`
  Determines whether, if a valid WAL path is detected at file open time (indicating a library crash) the metadata recovery process will automatically begin.
- `herr_t H5Pget_wal_auto_recovery((hid_t fapl_id, bool *enable_auto_recovery)`
  Retrieves whether automatic metadata recovery on file open is enabled.
- `herr_t H5Fwal_flush(hid_t file_id)`
  Performs a log flush to the WAL. A counterpart to `H5Fflush()`, which performs a log checkpoint.

1.8 The Recovery Process

If WAL usage is enabled for a file, then recovery takes place at file open time if the file did not close properly. Whether the file was closed properly is tracked in the HDF5 superblock, similar to the ”journal open” flag from the Metadata Journaling RFC. It is tracked via a superblock message containing the filepath of the WAL file.

At file open, this message is written, and it is deleted only at the end of the file closing process. Any premature termination of the library or an application using the library will leave the message in the superblock, which can be detected at file open time.

The process to restore a file's metadata to a self-consistent state is as follows:

- Scan the WAL from the beginning to the end to find both the most recent log checkpoint marker and the most recent log flush marker.
- Starting from the most recent log checkpoint marker in the WAL (or from the beginning if there are none), read each WAL entry. Copy each entry's block of metadata to the specified offset in the file.
- When the recovery algorithm reaches the most recent log flush marker, stop processing any further WAL entries.
- Trim the WAL if the file was opened with WAL enabled. If the file was opened with the WAL not enabled, delete the WAL file and remove the HDF5 superblock message.

If a file was not closed cleanly and is reopened with read only access, the recovery process is different:

- Scan the WAL from the beginning to the end to find both the most recent log checkpoint marker and the most recent log flush marker.
- Starting from the most recent log checkpoint marker in the WAL (or from the beginning if there are none), read each WAL entry. Create a stub entry in the cache for each WAL entry.
- When the recovery algorithm reaches the most recent log flush marker, stop reading any further WAL entries.

Depending on available resources, we may delay implementation of the read only reopen case.

2 Implementation

2.1 Changes above the Metadata Cache

While most of the changes to the library to support the WAL will be in the metadata cache and the new WAL package, some supporting code must be added to the upper layers. Here are some of the changes that will need to be implemented:

- The superblock code needs to handle the additional WAL filepath field that indicates if the file was closed normally.
- Properties will need to be set up and passed down as appropriate.
- The file code will need to handle opening and closing the WAL file itself, checking for incorrect shutdown, and initiating a recovery if appropriate.
- The file code will also need to pass down explicit log flush requests, and will need to make sure to issue VFD layer `flush` operations after log checkpoint operations.

2.2 Changes to the Metadata Cache

2.2.1 Stub Entries

There are a few different ways to implement the metadata stub entries that instruct the metadata cache to look at the WAL to find the actual metadata. Since the entry is stored in serialized, on disk format, it behaves differently from other cached metadata entries. We can either add the stub entries in the same tables as the standard entries, or we can create an entirely separate table of stub entries, which use an entirely different structure. Using the same table prevents the need for multiple table lookups when loading an entry. However, in the case where an entry is flushed to the WAL but not evicted until a later operation, while still clean, the MDC needs to know the address in the WAL that the new stub entry should point to. While this could be handled by adding a WAL address field to the normal entry struct, using a separate table may be simpler and clearer. In general, attempting to use the same cache entry struct for two very different things can cause maintainability issues. It may be possible to use an intermediate struct or ”common prefix” style nested struct to eliminate the extra lookup while keeping the structs for normal entries and WAL stubs mostly separate.

2.2.2 Entry Flushes

When the WAL is enabled, all flushes of entries from the metadata cache will by default go to the WAL. This operation will be similar to the existing code for flushing a single entry, except it will forward the write call to the WAL module, and create a stub entry which stores the address returned from the WAL module (and the length already known to the MDC).

On a log flush operation, the MDC will iterate over all dirty entries in the cache (these must all be non-stub entries because stub entries cannot be dirty), flushing each to the WAL (and logging the address or creating a stub entry as discussed above). After finishing this, the MDC writes a log flush marker to the WAL, then initiates a VFD layer `flush` operation on the WAL, then returns.

On a log checkpoint operation, the MDC will first perform a log flush, including the VFD `flush` operation. This ensures that the file will be recoverable if the program stops before the log checkpoint completes. Next, the MDC will iterate over all entries in the cache, flushing all entries with an associated stub entry to the HDF5 file, and deleting the associated stub entries. It will then iterate over all remaining stub entries, reading each corresponding entry from the WAL file and writing them to the appropriate place in the HDF5 file. The order does not matter, since the MDC only keeps (or keeps a stub entry to) the most recent version of each piece of metadata. Next, the MDC initiates a VFD `flush` on the HDF5 file, writes a log checkpoint marker to the WAL (or trims the WAL), then initiates a VFD `flush` on the WAL, and returns. This way we ensure that the WAL recovery start point is only advanced once we’re sure the HDF5 file has all metadata written to it.

The Metadata cache will also be responsible for determining when to perform scheduled log flushes and checkpoints, whether based on time or data written.

2.3 The WAL Module

The operations that the WAL must expose are as follows:

- Create a WAL entry for a metadata write, and append it to the end of the WAL file.
- Read the superblock of a WAL file to retrieve information about its version and parent HDF5 file.
- Read a single WAL entry from a specific address in the WAL file.
- Trim all WAL entries, removing everything from the WAL file except its superblock. Care should be taken that only log checkpointed entries are trimmed in order to avoid file corruption via loss of metadata writes.
- Create a log flush or log checkpoint marker, and append it to the end of the WAL file. A log flush marker that indicates which entries are safe to replay in the event of a crash. A log checkpoint marker indicates which entries have been written to the file and may be safely trimmed.
- Read all entries from the latest log checkpoint marker until the latest log flush marker, to implement recovery. This can be implemented using an iterator, returning the addresses of the first and last entries to read, or returning all the entries in a single call.

In addition, to improve performance of log flushes and checkpoints, we may wish to add:

- Create a series of WAL entries for a metadata write, and append them to the end of the WAL file, optionally adding a log flush or checkpoint marker after the entries.
- Read a series WAL entries from a list of specific addresses in the WAL file.

3 Testing

In order to get some preliminary results to verify whether or not this is a good approach for crashproofing HDF5, we identified two metadata-intensive benchmarks (A and B) and one non-metadata-heavy benchmark (C):

- Benchmark A: generates an HDF5 file of size 135 MB and a total of 77,111 metadata writes.
- Benchmark B: generates an HDF5 file of size 23 GB and a total of 1,821,278 metadata writes.
- Benchmark C: generates an HDF5 file of size 41 MB and a total of 66 metadata writes.

benchmark	file size	metadata writes
A	135 MB	77,111
B	23 GB	1,821,278
C	41 MB	66

We then modified the library's MPIO driver to write the address and size of all the metadata writes to a log file M.

A dummy WAL writer was created to read the log file M, and for each piece of metadata read, write the metadata address, size, and bytes equal to the size of the metadata of random data to a dummy WAL file. This is meant to simulate the writing of actual metadata into a WAL file down the line.

To determine the overhead from occasional HDF5 file syncs, which are necessary for a *Log Checkpoint*, we modified the library's MPIO driver to flush the HDF5 file intermittently based on the amount of metadata written. The time needed for the HDF5 file syncs was also recorded to see if it would be problematic to keep the WAL file small for metadata heavy applications.

The same thing was done for the dummy WAL writer. Since the dummy WAL writer would need to handle both *Log Flush* and *Log Checkpoint*, we added the ability to flush every x bytes to simulate a *Log Flush* to help sync the playback location and the ability to trim and flush the file every y bytes to simulate a *Log Checkpoint* to help reduce the size of the WAL.

For runs with no *Log Checkpoint*, the flush interval has no effect on the modified MPIO file driver, and hence we used the results from a single set of averaged runs for all flush interval cases. For Benchmark C, the dummy WAL writer does not contribute a significant amount of time to the result, hence the results for Benchmark C with only *Log Flush* show the same result for all flush intervals.

Specifically, we tested:

- A *Log Flush* every 64 KB and a *Log Checkpoint* every 1 MB
- A *Log Flush* every 256 KB and a *Log Checkpoint* every 4 MB
- A *Log Flush* every 1 MB and a*Log Checkpoint* every 16 MB
- A *Log Flush* every 4 MB and a *Log Checkpoint* every 64 MB
- A *Log Flush* every 16 MB and a *Log Checkpoint* every 256 MB
- A *Log Flush* every 64 KB
- A *Log Flush* every 256 KB
- A *Log Flush* every 1 MB
- A *Log Flush* every 4 MB
- A *Log Flush* every 16 MB

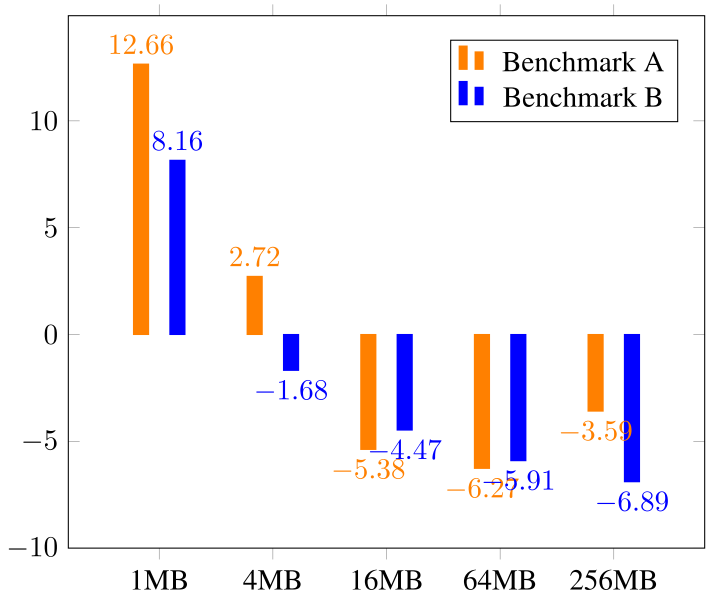
The preliminary results for benchmark A & B indicated that:

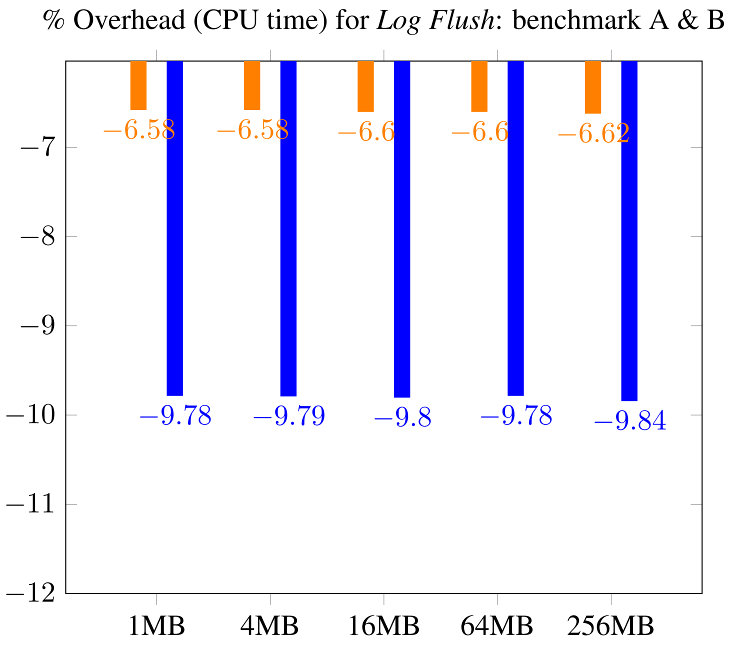
- The overhead for *Log Flush* + *Log Checkpoint* both decrease as we increase the interval at which we perform HDF5 file syncs.
- The overhead for *Log Flush* is minimal, and mostly comes from adding an HDF5 file sync before file close.

See the graphs below for more information.

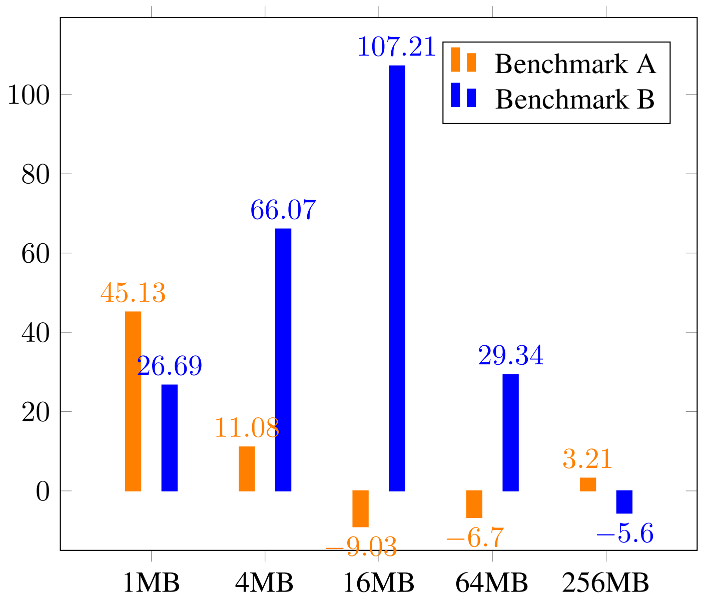
Note the X-axis for the graphs list the *Log Checkpoint* interval, even for runs with only *Log Flush*. The *Log Flush* interval is always as listed above for the corresponding specified *Log Checkpoint* interval.

% Overhead (CPU time) for *Log Flush* + *Log Checkpoint*: benchmark A & B

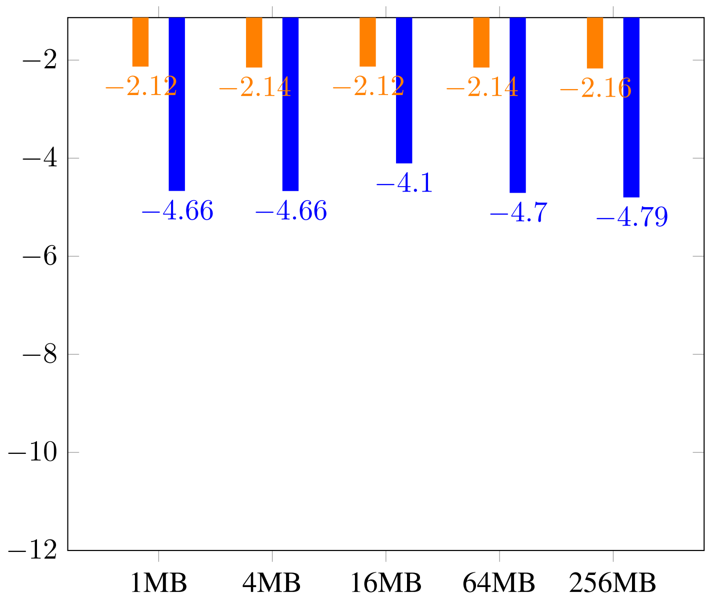



% Overhead (CPU time) for *Log Flush*: benchmark A & B
−7
−8
−9
−10
−11
−12
−6.58
−6.58
−6.6
−6.6
−6.62
−9.78
−9.79
−9.8
−9.78
−9.84
1MB
4MB
16MB
64MB
256MB

% Overhead (WALL time) for *Log Flush* + *Log Checkpoint*: benchmark A & B

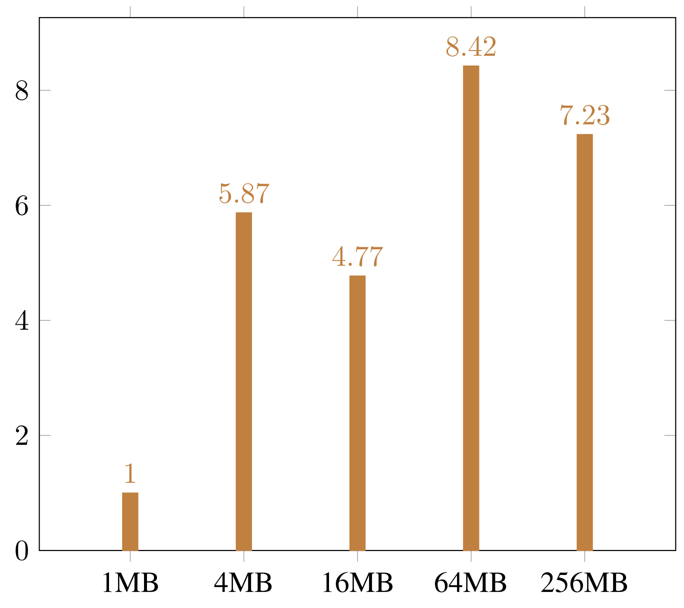


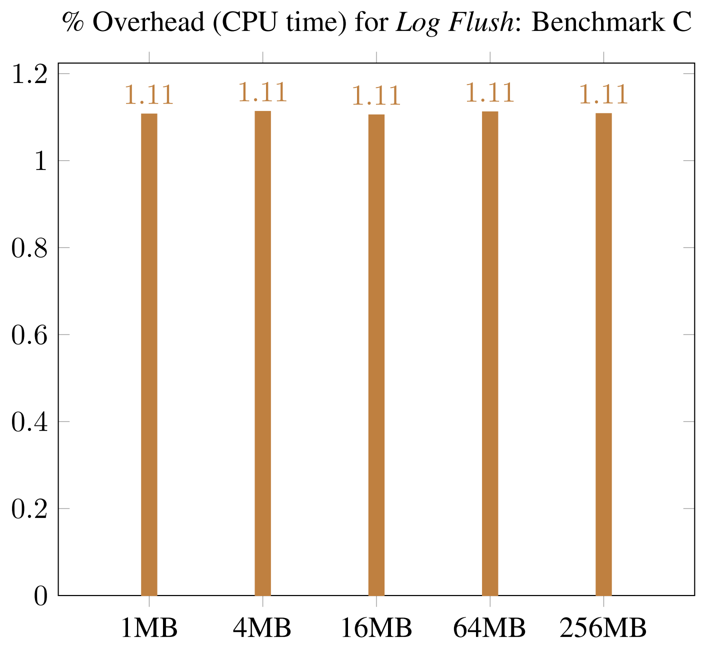
% Overhead (WALL time) for *Log Flush*: benchmark A & B



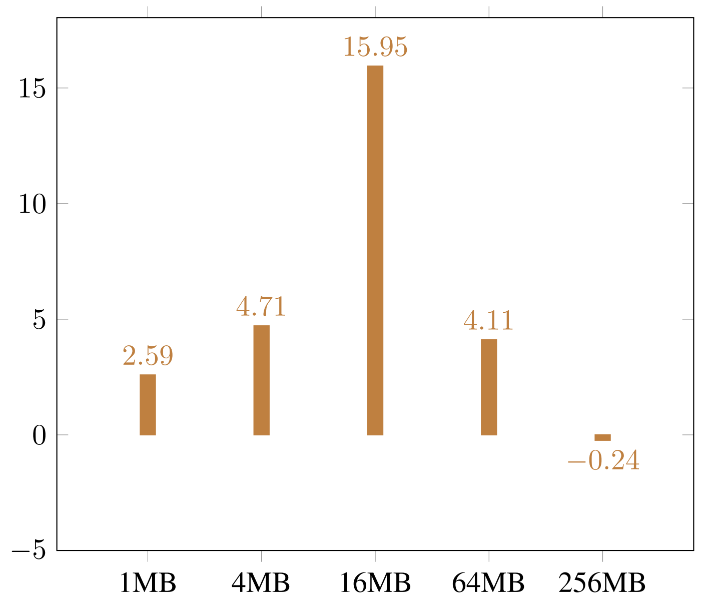
As for benchmark C, the four graphs below indicated that the overhead for *Log Flush* as well as *Log Flush* + *Log Checkpoint* is minimal.

% Overhead (CPU time) for *Log Flush* + *Log Checkpoint*: Benchmark C

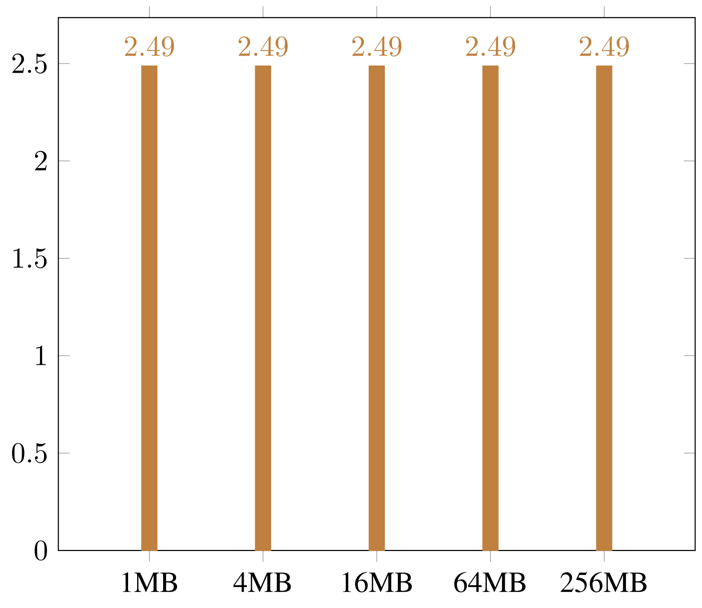



% Overhead (CPU time) for *Log Flush*: Benchmark C
1.2
1
0.8
0.6
0.4
0.2
0
1.11
1.11
1.11
1.11
1.11
1MB
4MB
16MB
64MB
256MB

% Overhead (WALL time) for *Log Flush* + *Log Checkpoint*: Benchmark C

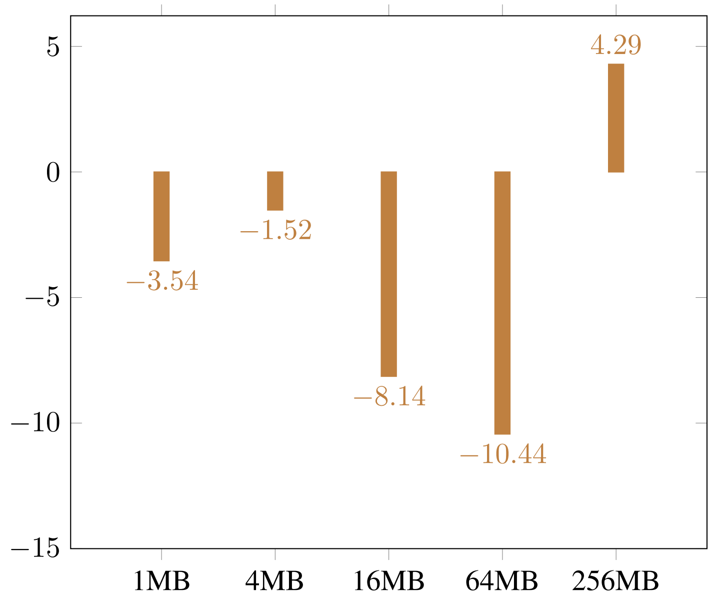


% Overhead (WALL time) for *Log Flush*: Benchmark C

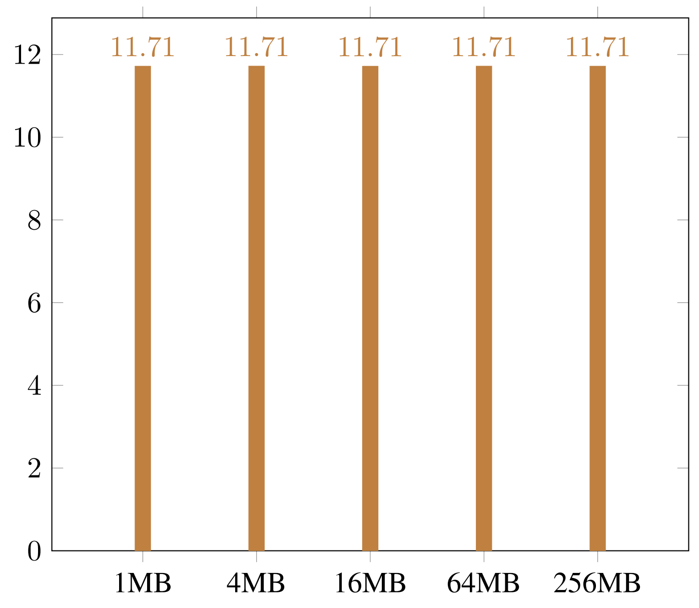


The following four graphs (CPU and WALL time) report the overhead for *Log Flush* as well as *Log Flush* + *Log Checkpoint* when running benchmark C with a larger file size (167MB) and 4 ranks.

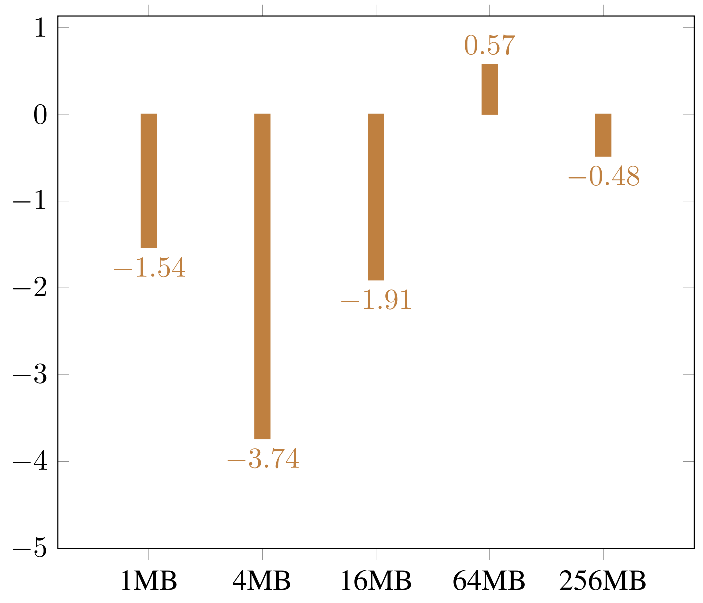
% Overhead (CPU time) for *Log Flush* + *Log Checkpoint*: Benchmark C (larger file size)



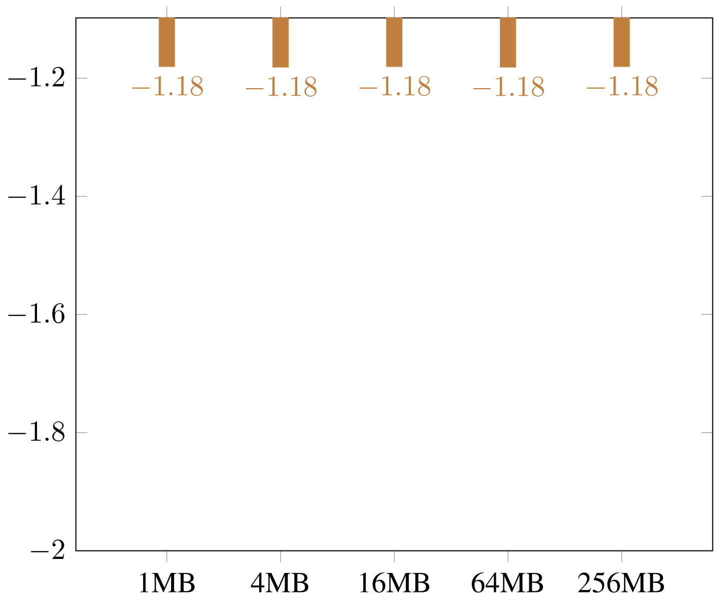
% Overhead (CPU time) for *Log Flush*: Benchmark C (larger file size)



% Overhead (WALL time) for *Log Flush* + *Log Checkpoint*: Benchmark C (larger file size)



% Overhead (WALL time) for *Log Flush*: Benchmark C (larger file size)



ss