Shared Chunk Cache Design

Dr. Clay Carper
John Mainzer

May 30, 2025

1 Overview for Redesigning the Shared Chunk Cache

The Shared Chunk Cache (SCC) is a software-based data caching system that will replace the current HDF5 Chunk Cache. The intention of this work is to focus on addressing previous problematic design choices, make considerations for multi-thread applications, while creating support for new variations of chunk types. For example, building in support for the sparse data-focused structured chunk is essential at this point. Using Version 6 of the Shared Chunk Cache API document as a key reference point[1], this portion of the redesign encompass a transition to a single, unified chunk cache. The discussion offered in the subsections below is the first full draft of my proposed design for the SCC. First, the major design requirements are outlined, with some commentary concerning how they each point is addressed in the redesign. Next, the structures are outlined, with further commentary where appropriate. The third section provides discussion concerning common I/O cycles, aiding in illustrating how the outlined structures will function together. Finally, commentary concerning loose ends, such as how the single-threaded approach that will be delivered to THG was designed with an eye for how to accommodate a transition into the multi-thread-focused Lifeboat product that will be developed[2].

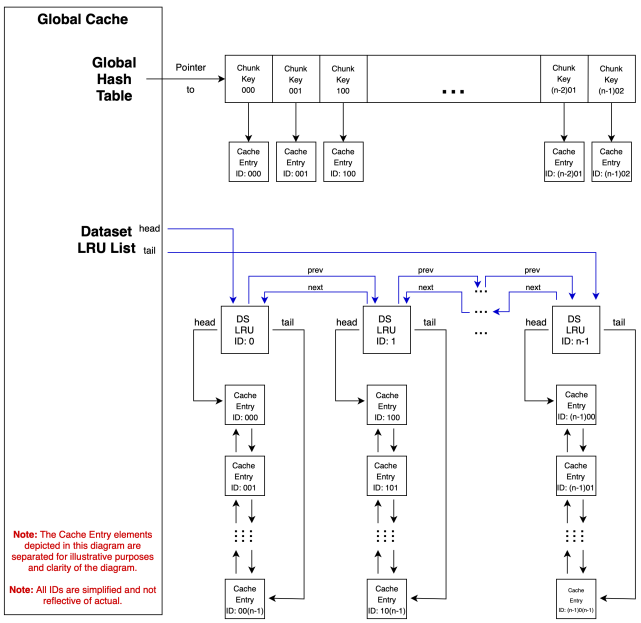

Figure 1: A brief illustration of the global shared chunk cache architecture. There are two key components - a global hash table with references to all chunks in the cache and an LRU-style list of individual dataset-specific LRU-style lists. Each individual LRU-style list asserts an ordering of chunks specific to each dataset containing data within the cache. Note that the IDs are simplified and do not reflect actual values. Additionally, the cache entry elements depicted in this diagram are separated for illustrative purposes and clarity of the diagram.

[1]Please note that I have not reviewed the current implementation of the HDF5 cache design, and have instead relied upon discussions with folks within Lifeboat and THG and the available design documents.

[2]In short, I have put a great deal of thought into this portion of the design to reduce the amount of repeated work required to develop the multi-thread version of the SCC.

2 Design Requirements and Considerations

2.1 Considerations for the Structured Chunk

The considerations listed below are duplicated directly from the original sketch design document shared with me (v3, if I'm not mistaken). To provide context for how these have influenced my approach to the redesign, I provide commentary for each statement, including brief discussions of how components of the redesign address each point of consideration. For a more detailed discussion of the structures relevant to this redesign, see the Data Structures section.

1. "Cache un-encoded versions of sections of structured chunks. The initial application of this ability will be the structured chunk sections used to store sparse datasets. In the future, we will need to add un-encoded versions of pointers into variable-length data heaps for sparse and dense variable-length datasets and tables of types used in non-homogeneous data sets."

    - Based on Vailin's work on the layout callbacks, support for sections is being considered. With respect to the shared chunk cache, the ability to track pointers to both the encoded (on-disk) and decoded (memory cache format) forms of the chunk. To my knowledge, this won't be an issue.

2. "Be able to adapt to changing structured chunk section sizes. For example, in the sparse chunk case, both the encoded selection and the fixed data sections will change size as entries in the structured chunks are defined and undefined. Similarly, for variable length data, heap size will change as variable length data is added, deleted, or modified."

    - Similar to the previous consideration, Vailin's work seems to have this covered. At the time of writing, the intermediate chunk struct contains fields which track the size of the chunk buffer, along with the total number of bytes allocated across all buffers for this chunk.

3. "Be able to perform operations on individual sections of a structured chunk, both on load and in preparation for write. For example, for sparse data, the selection section will have to be decoded on read, and encoded before write. Similarly, it will be useful to defragment heap sections just before filtering and writing to disk if the percentage of unused space exceeds some user-defined value."

    - Yet again, this will be handled through the layout callbacks Vailin is working on. Integration with the shared chunk cache will be based on how the components are defined within `H5Dstruct_chunk.c`.

4. "Run different filter pipelines on the different sections of structured chunks on read. Similarly, on write, run different filter pipelines on the various sections of the structured chunk, and assemble the resulting buffers into the on-disk image of the structured chunk on write."

    - Support for different filter pipelines will be layout-dependent. At the time of writing, it is yet to be determined if operations, such as conversion, will be handled internal by the shared chunk cache or in an external manner.

5. "Manage checksums on individual sections of structured chunks where required. This is necessary, as sections of structured chunks that indicate where raw data is to be found (i.e., the encoded selections used in sparse data) are in effect metadata, whose correctness must be verified to prevent buffer overflows and related run time errors."

- To my knowledge, this will be handled by the layout callbacks; that is, the shared chunk cache will simply manage the raw data. Based on my current understanding, managing checksums has been baked into the structured chunk layout.

6. "Be able to read and cache individual sections of structured chunks. A possible application is to cache only the selection sections of structured chunks in sparse datasets, to allow efficient determination of the locations of defined data across the entire dataset. Whether this will prove a useful optimization remains to be seen, but if we want to explore it, the capability should be designed into the chunk cache to begin with. The facility could also be used to support reads / writes of individual pieces of data without loading the full structured chunk in the unfiltered case – as is currently done with contiguous and unfiltered chunked data sets."

    - The ability to cache specific selections of structured chunks will depend on the underlying layout callbacks, as they dictate what data is allocated to the data buffer.

2.2 Addressing Known Deficiencies and Requirements

Similar to the previous section, the current deficiencies are, at the time of writing, an overview of the listed issues associated with the current implementation, as described in documentation shared with me. It is worth noting that I have elected not to expend energy on the exploration or analysis of the current chunk cache representation. As such, my interpretations are strictly based on what I have gleamed from the design documents and discussions with John and other folks. In essence, these elements are treated as a "guardrail" for what has been considered (or avoided) when sketching out my design. It is worth noting that some of these components may be addressed by Neil and/or Vailin's work.

Within the design document (v3), the following design decisions are outlined as being problematic:

1. Creating a chunk cache for each open chunked data set, which can be very memory intensive.

    - **Context: The intention was originally to have a single chunk cache per file, but due to poor performance (attested to I/O) taking up too much space in the chunk cache, this was altered.**
    - **Context:** It was also noted that a more appropriate (read: complex) replacement policy could alleviate this.
    - **Planned Improvement(s):** To address both of these points, the current plan is to utilize a single shared chunk cache per file by utilizing an indexing scheme paired with a dataset-focused filtering scheme. This will cut down on memory allocation, reduce the number of caches a chunk could be present in to a single global chunk cache, while maintaining the ability to operate on individual datasets. Addressing the need for supporting a more appropriate replacement policy, an altered LRU policy will form the basis for first-steps, while user-defined configurations can be swapped in as needed.

2. A 1 MiB chuck cache is created for each of the opened chunked data set(s). This creates thrashing when a chunk of size greater than 1 MiB is created or there is overlap.

    - **Planned Improvement(s):** Currently, the only size restrictions that will be enforced by the shared chunk cache are those specified by the user or the system with respect to the available memory. Since a single, unified chunk cache will be utilized, the issue of data

thrashing as described above should be avoided by design. It is worth noting that each dataset will have a doubly linked list associated with it for managing eviction candidate selection. Within each list, a minimum of 1 MB of chunk data will be maintained. Should a specific dataset require more data to be held in cache consistently, this minimum may be modified by the user to suite their needs.

3. The current indexing method, which uses a hash table for indexing, has a few issues:

    - **Context:** The hash table handles collisions by evicting the pre-existing entry.
    - **Context:** To avoid this behavior, the hash table is larger than should be necessary (fair band-aid, but worth overhauling).
    - **Planned Improvement(s):** UTHash, coupled with a unique key schema, will be utilized to address the known issues. For the key, a combination of the identifying information for a dataset TODO: (update this with the actual mechanism), along with the chunk offset, will be used to generate a unique 128-bit key. This key, coupled with UTHash, will enable the usage of a minimal hash table that, by design, will avoid collisions.

The aforementioned design document goes on to describe the nine requirements below to address deficiencies associated with the current chunk cache that are necessary for consideration. Similar to the previous subsections, statements given in quotations are direct quotes from the provided design documents, followed by comments relevant to the proposed redesign of the shared chunk cache.

1. "Shared chunk cache: To avoid the chunk cache size explosion when large numbers of datasets are open simultaneously, share the chunk cache between all open datasets. Whether we do this on a per-file basis, or across all open datasets in all open files, is an open question. Given the complexities of a global chunk cache, we currently lean towards a per-file chunk cache – but the idea should be explored."

    - Sticking to the notion of a single SCC per-file has been a central focus of this redesign. While, with some modification, the chunk indexing schema could support multiple files, I've been proceeding through this design as if we have a single file with multiple datasets. Keeping this issue in mind, the notion of external datasets has come up. If support for external datasets is integrated, a portion of the 128-bit hash table keys will be allocated to a file-dependent identifier[3]. This is not a central concern with the redesign; rather, it has been a consideration present in the background of this redesign.

2. "Track which chunks belong to which datasets. We need this to facilitate flushing individual datasets, both on user command and on close. It also facilitates replacement policies based on datasets as well as individual chunks."

    - The chunk indexing schema within the redesign preserves information concerning which dataset a chunk is associated with. In doing so, each dataset will have doubly linked list that can be used to maintain a least recently used (LRU) ordering on a per-dataset basis. This will enable the management of individual datasets, while also allowing for an approximate ordering of the oldest chunks to be available for eviction candidate selection across all available chunks within the shared chunk cache.

[3] As of the time of writing, this notion has not been explored beyond this initial consideration.

3. "Track what portion of each chunk in the cache has been either read or written. This data is used in the current chunk cache replacement policy and will likely be a factor in shared chunk cache replacement policies."

    - Support for detecting partially read or written data will be handled using one of three detection strategies. The differentiation of full and partial reads/writes allows for the application of replacement polices, such as LRU-based eviction policies, with various degrees of granular control within the shared chunk cache.

4. "Support for minimum chunk cache space allocations on a per-dataset basis."

    - Using the foundation of maintaining the ability to operate on individual datasets, establishing support for space allocations on a per-dataset basis will be built in with this redesign. As previously mentioned, each dataset will have an LRU-based list associated with it. Each of these lists will have a configurable field for the minimal space allocation for that dataset. The default value for this parameter will be 1MB.

5. "Support for alternate replacement policies. While the initial replacement policy will almost certainly be some variation on LRU, it is easy to come up with scenarios where alternate policies would be useful. Support for this should be designed in."

    - Flexibility with respect to replacement policies has been a consideration throughout this redesign. Generally, the datasets will be allocated to a doubly linked list, which initially, will be maintained by an LRU-style logic. Should the need arise, this structure, coupled with chunk elements such as a usage counter, to enable alternate replacement policies, such least frequently used. Support for alternate, more complex data-focused replacement policies would likely require further additions, which, at the time of writing, has been shelved as a future optimization.

6. "Support for adaptive chunk cache resizing. The metadata cache has had the ability to adjust its size to match the current working set size within user specified limits and time frames since HDF5 version 1.8. Given the success of this feature, and how the working set size of the chunk cache can vary based on access patterns, adding a similar facility to the chunk cache seems obvious. Note, however, that chunk cache entries behave very differently from metadata cache entries – which suggests that the problem may have very alternative solutions in the chunk cache."

    - Support for a version of the shared chunk cache that supports adaptive resizing will be dependent on the efficiency of the relevant callback functions, along with the requirements of a given application. At the time of writing, the proposed global cache structure will have field-based components for managing the quantity of memory available to the shared chunk cache. Dynamically resizing the cache will require some notion of adjusting the minimum space allocated on a per-dataset basis. Once the first full prototype has been implemented, this notion will need to be explored further.

7. "Support for background thread(s) prefetching chunks, preparing dirty chunks for flush when they become likely candidates for flush and/or eviction, or preforming read or write operations on chunks in the cache. Given the cost of these operations on large chunks, this should provide a significant performance boost when there is available compute to support it."

- During the redesign of the serial version of the shared chunk cache, support for utilizing additional threads, such as background threads, has been treated as an integral consideration. While utilizing background threads is venturing dangerously close to multi-thread support, the notions of prefetching chunks, preparing dirty chunks, or performing read/write operations on chunks in the cache are all features that will be supported. Prefetching, especially when working with contiguous chunks, shouldn't be an issue. Dirty chunks across available datasets will likely be batch-processed, drastically reducing the computational overhead of processing them individually. As for performing read/write operations within the cache, this should be supported by the top-level APIs for the shared chunk cache, since the available I/O operations will first check if a desired chunk is present within the shared chunk cache.

8. "Multi-thread support. While the version of the chunk cache for the existing HDF5 library will have to be single thread, the Lifeboat version will have to be multi-thread to fit into the H5+ product."

    - Multi-thread support has been a central consideration of the redesign. Currently, the intention is to transition away from UTHash in favor of utilizing a lock-free hash table, along with utilizing atomics within each cache entry (i.e. data chunk). Outside these notions, there is still a fair amount of work to be done when considering the additional lock-free data structures that will be required to transition to supporting multi-threaded applications.

9. "Make chunk cache available for parallel I/O. To accomplish this, move some or all support for collective I/O on chunked datasets with variable-size chunks to the chunk cache and/or the chunk format layer."

    - In V6 of the SCC API document, Neil states: "Currently we plan to have the shared chunk cache query MPI collective settings and the selection I/O setting, and track and report the actual MPI modes and actual selection I/O mode, so this info does not need to be passed in the shared chunk cache API." Based on how this statement reads to me, it seems like the intention is to keep I/O exterior to the SCC. This would be worth clarifying in the near future.

2.3 Redesign Requirements

Note: This subsection will be worked into the previous sections to make things more cohesive. Some refactoring will be required, especially with the alterations to the redesign (see the notes in the Data Structures section for more details).

During the redesign process, many factors have come to light, showcasing the importance of a configurable Shared Chunk Cache (SCC) that addresses previous design restrictions while paving the way for novel HDF5 features, such as supporting sparse data. Under these considerations, the requirements listed below have been the central focus of the redesigned SCC.

- Simplifying the process for identifying when a single chunk of data is within the dataset cache, without the need to iterate through data structures.

- Maintaining a file-level hash table that avoids collisions without being oversized.

- Creating support for multiple eviction policies, allowing eviction candidate selection decisions to not exclusively depend on least recently used ordering (though this is still the default).
- Differentiating between full and partially written/read chunks, allowing for fine-tuned eviction selection and more refined chunk management.
- Optimizing I/O requests based on memory availability, while making considerations for situations where contiguous data will be processed.
- Last but not least, the necessity of multi-thread support in the near future.

While the details of the necessary structures are outlined in the next section, it will likely be helpful to provide a high-level overview of how the above requirements will be addressed. For reference through this discussion, Figure 2 provides a UML diagram illustrating how the various structures in the redesign are related. The linchpin of this redesign lies in how a single chunk of data is identified. Within a HDF5 file, the ID assigned to a dataset within a file (to my knowledge) is fixed, while the indexing of chunks within a dataset is non-unique; that is, chunks are sequentially indexed within a single dataset. This pair of values provides a consistent way of identifying a specific chunk, and with some minor adjustments, they may be interwoven to form a strong key for a hash table. While the details of this construction are omitted from this section, it is worth noting that each chunk key will be a 128-bit value.

To maintain consistent, constant-time lookups for chunks, a globally available hash table will be utilized[4] through the use of UTHash. The hash table implementation we will rely on the mechanisms provided by UTHash, through the inclusion of a single C header file. Continuing from existing HDF5 paradigms, UTHash utilizes the Jenkins hash function by default, while providing access to additional options. Simply stated, a UTHash-based hash table is made up of structs, each containing a key/value pair. UTHash is highly compatible with structures, one or more fields encompassing the key. Of note, the structure pointer acts as the value. The chunk key is well suited for utilization within UTHash, providing a unique, data-focused identification for each individual chunk[5]. When paired with UTHash, the previous need for a large hash table is outright avoided, and through the usage of a global hash table, a single hash lookup is required to determine whether a chunk is present within the SCC. Hash collisions are resolved using a chaining method, with repeated entries are linked in a list within the same bucket.

Regarding eviction strategies, the global cache points to the head of a doubly linked list which maintains an LRU-based ordering for datasets; allowing for dataset-specific chunk management[6]. When considering chunks for eviction under LRU, the dataset cache at the tail of the dataset LRU list is nominated first. Global size statistics are updated after each eviction, with iteration through datasets continuing until the desired amount of space is available[7]. Should support for alternative eviction strategies be desired, such as Least Frequently Used (LFU), or Most Recently Used (MRU), minimal modifications will be required. To support rank-based eviction strategies, such as frequency-based eviction, an integer-based counter is made available within each cache entry, while alternate order-based procedures need to simply correctly manipulate the available pointers. As it stands,

[4]With an additional hash table for each dataset, forming a dataset cache for each dataset within the HDF5 file.

[5]As a general note, UTHash does not verify the uniqueness of a key; this is left as an exercise to the user. However, the `HASH_FIND` macro is provided to check whether a key is present within the hash table.

[6]Please note that data chunks are maintained in a single, global cache. This secondary structure simply maintains references to loaded chunks

[7]Please note that this description is operating under the assumption that the requested data will fit within the available memory constraints of the global chunk cache. This may not always be the case.

support for non-LRU eviction policies has been designed around from the start, while considering the need for more configuration options without requiring heavy rewriting of cache components.

Omitted from the above discussion, the distinction between fully or partially written/read chunks is a critical consideration in eviction candidate selection. Within a given dataset cache, chunks will be differentiated based on which category a given chunk falls into. To distinguish between these two classes of chunks, three heuristics will be developed, with the default being the bounding box strategy.

- *Bounding Box:* A size-based filtering process that, when the amount of data processed matches or exceeds a threshold value, is considered a full read/write.
- *Metadata for Partial States:* TODO: Expand on the purpose of this component. Existing information, such as the chunk offset, is stored as part of the chunk data. This provides a simplistic way of evaluating full/partial chunks, while reducing redundant reprocessing.
- *Checksum/Hash Per Chunk*: TODO: Expand on the purpose of this component. At the cost of increasing computational overhead, allows for validation of partial chunks though utilizing a checksum or data hash. Works to reduce the complexity of reassembling chunks, which is highly desirable for multi-thread support.

In general, fully read/written chunks will be prioritized when eviction is necessary. Rather than juggling multiple data structures, each chunk labeled as being partially read/written will be moved to the head of the list directly ahead of a separator. Once the separator reaches the tail, this process may be repeated until a preset threshold is reached, with the second pass being indicated by the presence of an additional separator block[8].

As for optimizing I/O requests, batching will be implemented to minimize overhead, reduce the number of on-disk data fetches, and to further utilize the redesigned SCC. This process is outlined in great detail later in this document. Additionally, this design accounts for dirty chunks, those which have been modified but not yet written back to disk, through the support of write-aware caching. By identifying and properly processing dirty chunks, batching can be further utilized to reduce unnecessary writes to disk. This inclusion provides further eviction optimization, allowing for clean chunks to be priority targets when eviction is necessary.

To account for this, each dataset LRU list will track the current size, regardless of whether the chunk contains fixed-size or variable-length data. Global cache size is also maintained, ensuring that memory allocation is consistently tracked. Currently, there are no plans to conduct eviction based on chunk size, though existing structures could be repurposed to do so through appropriate pointer manipulation[9].

Finally, multi-thread support has been a guiding force throughout this redesign. After discussing the data structures in the next section, a discussion focused on the adjustments necessary to facilitate multi-threaded applications is provided.

[8]Currently, I plan to have this threshold be two or three, though some fine-tuning will be necessary during implementation.

[9]For example, rather than maintaining an LRU-based ordering, one could develop a data ordering based on chunk size.

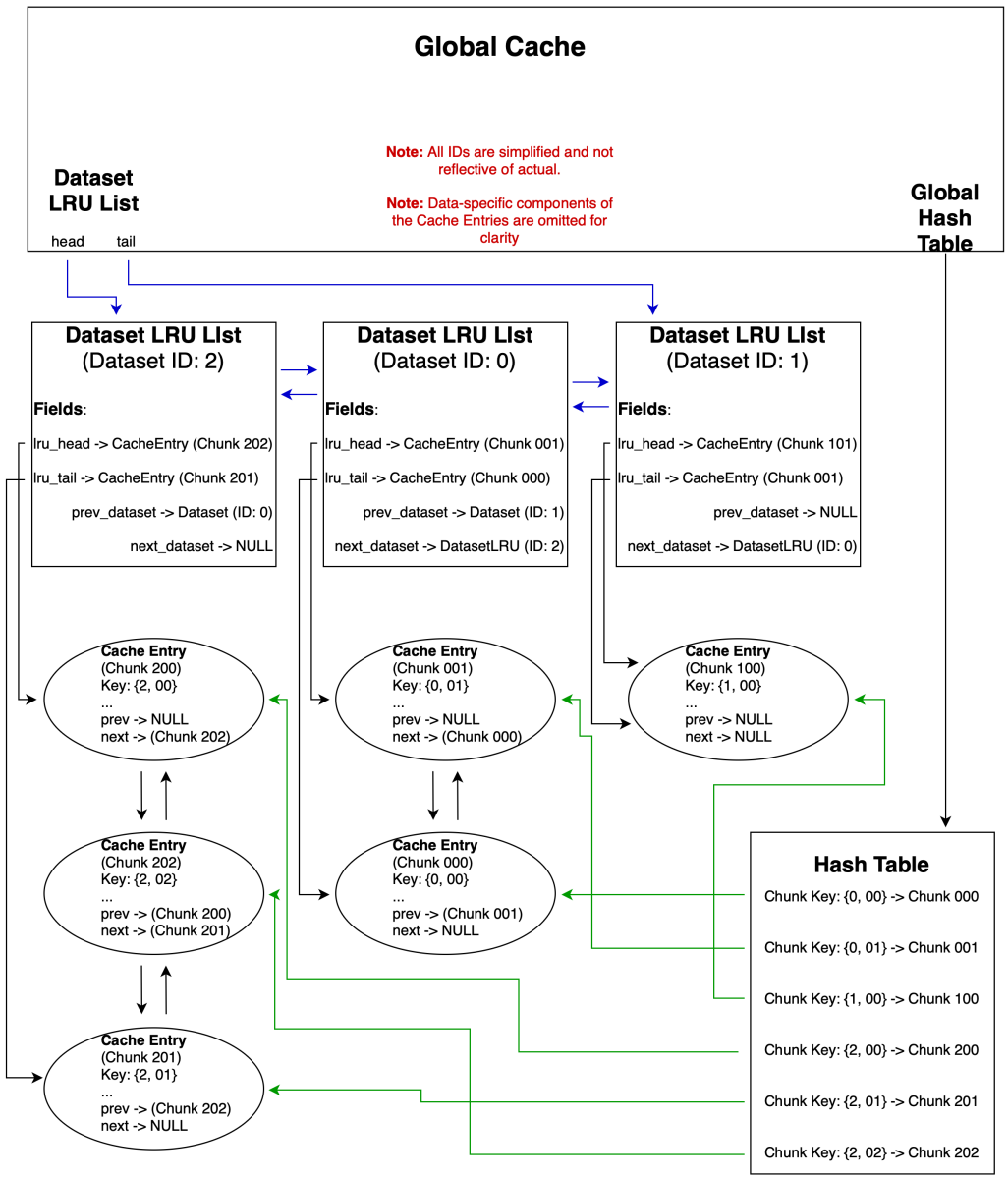

Figure 2: A detailed outline of the global shared chunk cache architecture. There are two key components - a global hash table with references to all chunks in the cache and an LRU-style list of individual dataset-specific LRU-style lists. Each individual LRU-style list asserts an ordering of chunks specific to each dataset containing data within the cache. Note that the IDs are simplified and do not reflect actual values. Additionally, data-specific components of cache entries are omitted for clarity.

3 Data Structures

This section provides an overview of the data structures developed during the shared chunk cache redesign.

3.1 Chunk Indexing with `ChunkKey`

- **Purpose:** Provides a unique identifier for cached chunks that is suitable for UTHash.
- **Basic Data Type:** Structure
- **Usage:** Utilized as the key for the global hash table.

```
/*
 * Structure: ChunkKey
 * --------------------
 * Stores a unique chunk key derived from the dataset ID (i.e. the object
 * header) and the chunk index.
 */
typedef structure ChunkKey {
    long long int low_half;  /* Bottom 64 bits of the 128-bit int */
    long long int high_half; /* Top 64 bits of the 128-bit int */
} ChunkKey;
```

Further Commentary: Within the redesign, one of the central requirements is to establish an indexing strategy that avoids collisions while remaining simplistic. the `ChunkKey` structure, which will be derived from information available in `H5D_dset_io_info_t`[10]. In particular, the proposed index will consist of an analog to the dataset index combined with a derived chunk index. Each of these two values will first be converted into a 64-bit integer; after which the bits from these two values will be combined through taking one bit from each integer at a time, starting with the least significant bit and working up to the most significant bit. This newly formed 128-bit-reversed value represents a unique index for an individual chunk within a specific dataset. So long as the dataset index is unique, there will not be any repeated values when operating on a singular data file.

3.2 Individual Chunk Management with `CacheEntry`

- **Purpose:** Representation of a single cached chunk, with local metadata, associated chunk data, and state information
- **Basic Data Type:** Structure with pointers
- **Usage:** Tracked through the global hash table and linked to the appropriate dataset-specific doubly linked list for an LRU-style ordering

```
/*
 * Structure: CacheEntry
 * ---------------------
 * Represents an individual chunk, including metadata, the chunk's position
```

[10]The minutia of this derivation will be determined during the implementation phase; however, I have dug into H5O enough to know this information *should* be available.

```
     * in its associated dataset LRU list, and information related to this entry
     * in the global hash table.
     */
typedef struct CacheEntry {
    ChunkKey data_key;       /* Unique key for the cache entry */

    void *cached_chunk;      /* Pointer to the in-cache chunk data (NULL if not
    yet loaded, decoded, etc.) */

    void *on_disk_chunk;     /* Pointer to on-disk representation of this chunk
    (NULL if newly created and not yet written to the file) */

    size_t chunk_size;       /* Size of the cached data, computed after
    compression/conversion */

    size_t file_offset;      /* Offset for the specified data chunk */

    int chunk_counter;       /* Unsigned integer available for tracking data
    access. Set to 0 by default */

    int ditry_flag;          /* Flag to indicate when a chunk has been modified
    (a value of 1 indicates dirty) */

    bool status_flab;        /* Flag to indicate when a chunk is being operated
    on by an I/O process, or is otherwise "busy" */

    struct CacheEntry *prev; /* Pointer to the previous node in the associated
    dataset's LRU list */

    struct CacheEntry *next; /* Pointer to the next node in the associated
    dataset's LRU list */

    UT_hash_handle hval;     /* Handle for this chunk; used by UTHash to track
    this struct in the global hash table */
} CacheEntry;
```

Further Commentary: The representation of a data chunk, `CacheEntry`, is a UTHashable structure, which, as previously alluded to, relies on a 128-bit integer key in the form of a `ChunkKey`. Otherwise, the in-memory data type is pointed to[11], which should be available from the `H5D_type_info_t` structure. Beyond this, a cache entry contains the necessary pointers to support the doubly linked list which encompasses the dataset cache structure which will manage instances of this structure.

[11] At the time of writing, the decision has been made to move conversion into the shared chunk cache. To accommodate this change, a pointer to the in-file chunk has been added to the cache entry structure. The details of this topic are yet to be fully discussed. Should further adjustments be required, they will be made and discussed in future versions of this design document.

3.3 Dataset Chunk Tracking for LRU Eviction Logic with `DatasetLRUList`

- **Purpose:** Maintains an LRU list to accommodate dataset-specific evictions.
- **Doubly Linked List**
- **Usage:** Collates cached chunks for a specific dataset and enables enforcement of per-dataset memory limits.

```
/*
 * Structure: DatasetLRUList
 * ---------------------
 * Represents the LRU ordering of chunks associated with a specific dataset and
 * provides pointers to establish a global, partial LRU ordering of datasets.
 */
typedef structure DatasetLRUList {
    size_t dataset_name;         /* Identifier, derived from the dataset
    header, for the dataset associated with this LRU list () */

    size_t current_dataset_size; /* Current total size of the chunks cached in
    this dataset */

    size_t min_dataset_size;     /* Minimum size allocated for this  dataset */

    CacheEntry *lru_head;        /* Pointer to the head of the dataset-specific
    LRU list (most recently used chunk) */

    CacheEntry *lru_tail;        /* Pointer to the tail of the dataset-specific
    LRU list (least recently used chunk) */

    struct DatasetLRUList *next; /* Pointer to the next dataset in the global LRU
    list */

    struct DatasetLRUList *prev; /* Pointer to the previous dataset in the global
    LRU list */
} DatasetLRUList;
```

Further Commentary: Similar to `GlobalHashTable`, establishing which datasets currently have chunks cached provides utility and consistency to the Shared Chunk Cache. When there is a need to iterate through all loaded datasets, such as when selecting candidates for eviction, this singly linked list provides a consistent way of establishing sequential access to all loaded datasets. Additional parameters are available in the `DatasetCacheList` structure for managing memory. As chunks within a particular dataset cache are accessed, the pointer to the appropriate `DatasetCache` will be moved to the head. This ordering enables a dual-LRU eviction strategy without a need for transversal of a complete list of currently loaded chunks[12]. In differentiating between chunk keys and their datasets, additions to this design, such as those necessary for multi-thread support, can remain flexible without a loss in single-thread performance.

[12]**Note:** The global LRU ordering is not approximate; that is, there is not currently a mechanism to evict chunks based on an absolute global LRU list. Instead, this system will operate on the notion of completing eviction candidate selections based on the LRU-ordering of datasets rather than an LRU-ordering of all available individual chunks.

3.4 High-Level Management with the `GlobalCache`

- **Purpose:** The structure that acts as the central coordinator for global cache management, tracking datasets, chunks, and memory usage.
- **Basic Data Type:** Structure with pointers
- **Usage:** Provides the framework for enforcing global memory limits, coordination of chunk eviction, and managing dirty chunk writes.

```
/*
 * Structure: GlobalCache
 * ----------------------
 * Manages the shared chunk cache through a UTHash-based global hash table for
 * establishing chunk lookups and maintains the pseudo-ordering of the global
 * LRU ordering of datasets.
 */
typedef struct GlobalCache {

    size_t global_mem_limit;          /* Global cache memory limit */

    size_t global_active_limit;       /* Global size for active data (reflects the
    amount of data exceeding the sum of the minimum data set size of each loaded
    dataset) */

    size_t global_quiescent_limit;    /* Global size for inactive data (reflects
    the sum of minimum dataset size of each loaded dataset)  */

    size_t global_mem_usage;          /* Current total cache size across all
    datasets (reflects the total memory usage of each loaded dataset) */

    DatasetLRUList *dataset_lru_head; /* Pointer to the head of the dataset LRU
    lists, indicating the most recently accessed dataset */

    DatasetLRUList *dataset_lru_tail; /* Pointer to the tail of the dataset LRU
    lists, indicating the least recently accessed dataset */

    CacheEntry *hash_table;           /* Pointer to the hash table for global chunk
    management. Since UTHash treats the hash table as a linked list, this is the
    head of the LL. Additionally, since the hash table consists of CacheEntry
    elements, this declaration embeds type information and allows for direct access
    to entries */
} GlobalCache;
```

Further Commentary: In contrast to the current chunk cache implementation, this redesign focuses on maintaining a single cache containing data chunks from the entirety of a single HDF5 file. In collecting all chunks under a single data cache, the need for complex, dataset dependent management strategies is omitted altogether. As the high-level cache management tool, the `GlobalCache` is responsible for enforcing global memory limits, maintaining lists of all datasets, and the coordination of chunk eviction. The two primary components, the LRU-style ordering of datasets and the global

hash table, form the basis for maintaining dataset-focused ordering of chunks. Additionally, the ability to dictate the available global memory, both in terms of a user-defined hard limit and based on some combination of the global active and quiescent memory limits, provides the ability to define application specific limitations.

4 Read/Write/Eviction Procedural Overview

This section provides an overview of how basic I/O operations will function using the previously defined structures. In particular, the following scenarios are overviewed:

- Reading a chunk from file to the SCC
- Reading a chunk from within the SCC
- Writing a dirty cached chunk to file from the SCC
- Writing a newly created, cached chunk to file from the SCC
- The LRU eviction cycle (unedited)

In all the given examples, instances of `ChunkKey`s are given as simple integers, rather than their actual in-cache representations, for the sake of clarity. It is also worth noting that the `GlobalCache` provides central management of the entire SCC; that is, individual read/write operations are delegated to the other specialized components. Note that the read/write examples below omit the distinction between fully and partially written chunks. After this first pass of writing, this distinction will be added to these examples[13].

4.1 Chunk Read From File

Suppose we have a chunk of data identified by a computed chunk key value of `1234`, and is a member of dataset number 3 which is not currently within the SCC. Assume the SCC has sufficiency memory allocation available to load the requested chunk without triggering the need for eviction. The procedural steps are as follows:

1. Data Request:
    - The system makes a request to read a data chunk with the computed chunk key `1234`
2. Global Key Lookup:
    - The SCC queries the `GlobalKeyMapping` using the chunk key `1234`
    - The key is not found, resulting in, a cache miss
3. Memory Enforcement:
    - The `GlobalCache` checks if the requested chunk will fit in memory
    - Under the assumption of space being available, the read request proceeds without triggering eviction

[13]The labeling of chunks as partially/fully written will be handled any time a read/write is invoked, which shouldn't alter these examples too much.

4. Load Chunk from File
    - A new CacheEntry is created for the chunk, using the specified key
    - Using the dataset ID and chunk offset, seek the desired chunk
    - If found, read the data chunk into a buffer[14]
5. Add Chunk to Cache:
    - Create a new CacheEntry structure for the loaded chunk of data
    - The newly minted Cache Entry is then added to:
        - The global hash table
        - The DatasetLRUList for dataset 3

4.2 Reading a Chunk from Within the SCC

Suppose we have a chunk of data identified by a computed chunk key value of `4321`, and is a member of dataset number 2 which is currently within the SCC. The procedural steps are as follows:

1. Data Request:
    - The system makes a request to read a data chunk with the computed chunk key `4321`
2. Global Key Lookup:
    - The system queries the hash_table in GlobalCache using the chunk key `1234`
    - If the chunk is found, resulting in a cache hit:
        - Update the LRU order:
            * The chunk is moved to the head of the DatasetLRUList for dataset 2
            * Move the DatasetLRUList for dataset 2 to the head of the global dataset LRU list in the GlobalCache
        - Return the cached data
    - If the chunk is not in the hash table, resulting in a cache miss:
        - Attempt to read the chunk from file

4.3 Writing a Dirty Chunk from within the SCC

Suppose we have a dirty chunk of data identified by a computed chunk key value of `2211`, and is a member of dataset number 8 which is currently within the SCC. The procedural steps are as follows:

1. Data Request
    - The system request to write a data chunk with the computed chunk key of 2211
2. Global Key Lookup
    - The system quires the `GlobalKeyMapping` hash table using the chunk key 2211.
    - The dirty_flag is noted to be set as 1, requiring the chunk to be flushed

[14]Note: This does not account for any decoding or conversion. These details will be added at a later date.

3. Initialize the Write Request
4. The system requests to write to the on-file chunk associated with ChunkKey 2211 within dataset 8
5. Flush the Chunk to File
    - Using the appropriate callback functions, the cached_chunk is processed, then written to the file
    - The dirty_flag is cleared, marking the chunk as clean
6. Update LRU Orderings
7. Move the chunk to the head of the DatasetLRUList associated with dataset 8
8. Move dataset 8 to the head of the global LRU list

4.4 Writing a Newly Created Chunk from within the SCC

Suppose we want to create a chunk of data, identified by a computed chunk key value of 1122, as a member of dataset number 4 which is currently within the SCC. Assume it is not present within the file. Additionally, assume that the indexing of this chunk does not conflict with the chunks in the file and the necessary on-disk space is available. Assume the SCC has sufficiency memory allocation available to store the requested chunk without triggering the need for eviction. The procedural steps are as follows:

1. Chunk Creation Within the Cache
    - Create a new CacheEntry for the chunk:
        - Assign a ChunkKey with a value of 1122
        - Allocate memory for the in-cache chunk data
        - Set the dirty_flag to 1 to indicate the chunk will need to be flushed
    - Add the newly created CacheEntry to:
        - The hash_table present in the GlobalCache
        - The DatasetLRUList for dataset 4
2. Chunk Lookup
    - Should there be any delay between creating and the write request, the system should verify that the CacheEntry with ChunkKey of 2211 is still present in the cache before proceeding
3. Initialize Write Request
    - The system requests to write a dirty data chunk with the computed chunk key of 1122 to dataset 4
        - Since this chunk is not yet present in the file, an appropriate offset will be computed
        - Other HDF5-specific allocations will be completed in this step, such as computing the in-file format size based on the required encoding/compression; more detail will be added soon

4. Flush the Chunk to File
    - Using the appropriate callback functions, the cached_chunk is processed, then written to the file
    - The pointer to the on_disk_chunk is updated from NULL to reflect the in-file location of the file formatted chunk
    - The dirty_flag is cleared, marking the chunk as clean
5. Update File Metadata
    - Complete any necessary file metadata to reflect the newly created chunk
6. Update LRU Orderings
7. Move the chunk to the head of the DatasetLRUList associated with dataset 4
8. Move dataset 4 to the head of the global LRU list

4.5 The LRU Eviction Cycle

Note: This section will be updated to utilize Figure 2 in a future update to this document.

Note: This example has not yet fully been updated to reflect the alterations made to the initial redesign.

Suppose the SCC contains three datasets, labeled 0, 1, and 2, each with four chunks, with the 1 at the head, 2 at the tail, and 0 between the other two nodes. Without loss of generality, assume that two of the chunks in each LRU list are partially written clean chunks (PWCC), one chunk is fully written clean chunk (FWCC), and the final chunk is a fully written dirty chunk (FWDC), in that order, respectively[15]. The eviction priority, accounting for the need to flush some chunks, will first evict fully written chunks, then partially written chunks, and finally the dirty chunks. Assume the `current_mem_usage` in the `GlobalCache` exceeds the `global_mem_limit`, triggering the eviction process. The procedural steps are as follows:

1. Eviction is Triggered
    - After processing chunks for the `DatasetLRUList` for dataset 1, `current_mem_usage` in `GlobalCache` exceeds `global_mem_limit`
2. Dataset Eviction Nomination
    - The element at the tail of `DatasetLRUList`, the LRU list associated with dataset 2, is nominated since it is the least recently used dataset
3. Dataset 2 Chunk Eviction Nomination Process
    - **Note:** The ordering given herein is an approximation and not an accurate illustration of the underlying doubly linked list (Initial ordering: PWCC $\rightarrow$ PWCC $\rightarrow$ FWCC $\rightarrow$ FWDC)
    - Beginning with the chunk cache element at the tail, the eviction priority is assessed:

[15]For simplicity, assume all three caches have the same ordering of chunks.

- The first chunk examined is a fully written dirty chunk (FWDC), which isn't the highest priority eviction. A separator value is first added to the head of the LL, then the examined chunk (order is now: FWDC → `seperator` → PWCC → PWCC → FWCC)
- No eviction occurred, and the item at the tail is not a separator, so nomination continues within this dataset LRU
- The next chunk is a fully written, clean chunk, which is the highest priority for eviction. This chunk is flagged for eviction.

4. Evicting the Fully Written Clean Chunk from Dataset 2
   - Within `DatasetLRUList` for dataset 2, the chunk is unlinked from the LRU list
     - This list's `current_cache_size` is decremented to reflect this eviction
   - In the GlobalCache, the CacheEntry associated with the evicted chunk is removed from the hash table, removing the global reference
   - The `GlobalCache` frees the memory associated with the evicted chunk and updates the `current_mem_usage`
5. Post-Eviction Memory Check
   - If the current memory usage is still too high, the eviction process is continued (assume, for the sake of the exercise, this is the case)
6. Second-Round Eviction Nomination (`DatasetLRUList` 2 Cont.)
   - After eviction, the ordering of the LRU list for dataset 2 is now: FWDC → `seperator` → PWCC → PWCC
   - Maintaining the eviction priority, the next two chunks fall into a lower tier of priority since they are partially written, clean chunks, with each of them being moved to the head of the list, resulting in an ordering of: PWCC → PWCC → FWDC → `seperator`
   - With a separator at the head, the indication is that there are no longer fully written, clean chunks present in this LRU. The separator is moved to the head, with the addition of a second separator, resulting in an ordering of:`seperator` → `seperator` → PWCC → PWCC → FWDC
   - When one or more separators is detected at the head, the eviction algorithm is signaled to move onto the next dataset in the list of dataset LURs
7. Moving on to Dataset LRU 0
   - In an identical process to Dataset LRU 2, items in the LRU list are examined until either a candidate is nominated for eviction, or at least one separator is hit
   - In an identical fashion, assume the fully written, clean chunk is found, nominated, and evicted
8. Repeat if Necessary
   - After evicting the second chunk, the global parameters are updated, and should further eviction be necessary, the process will be repeated until the necessary amount of memory has been freed.

5 Loose Ends

5.1 Transitioning to Full Support of Multi-Threaded Computation

Note: This section will be updated and merged into Section 2 of the document in a future update.

Note: This content has not yet been updated to reflect the alterations made to the initial redesign.

When considering the transition to supporting multi-threaded computation, two key considerations have been avoiding lock-based strategies and utilizing atomic operations. Focusing building around lock-free data structures will aid in avoiding contention when accessing chunks, reducing blocking, and deadlocks. Further, lock-free structures promote thread independence and system scalability. The content in this subsection will provide an overview of necessary modifications to data structures proposed for the serial implementation of the SCC, how global memory management components could utilize atomics, and how the proposed eviction policy could be modified to support multi-threaded usage. Additionally, a discussion is provided concerning the challenges presented by I/O coordination under the proposed SSC redesign.

The transition from the serial SCC into a multi-thread safe SCC will likely begin by replacing the existing data structures with lock-free equivalents. For example, the `GlobalKeyMapping` structure, which currently utilizes a UTHash-based hash table, could be transitioned to use a lock-free hash map to maintain constant time lookups, insertions, and deletions without the need for adding locks. This transition would likely require a custom implementation that encompasses the use of open addressing with an atomic compare-and-swap for updating the hash map buckets. To transition `DatasetLRUList` into supporting multi-threaded applications, a lock-free doubly linked list coupled with an atomic access flag would allow for per-dataset LRU ordering to be maintained. This would likely require altering the content of the current `DatasetLRUList` to reflect the need to restrict access to an individual dataset LRU list to a single thread. As for the `WriteBuffer` component, a lock-free ring buffer or lock-free circular queue would maintain the current functionality while allowing for concurrent addition and flush operations. This would likely require a set of atomic counters, allowing for management of the head and tail pointers.

5.2 SCC Version 0 Development Summary

The Shared Chunk Cache, Version 0 (V0), is a minimally functional representation of how structured chunk data will be processed through the cache. This initial version focuses on demonstrating the connection between the high-level API (through `H5D__write()`) and the low-level callbacks (which are invoked by the SCC). To maintain a simplistic, verifiable output, a single chunk within a single dataset was the sole focus of this development. To illustrate proper cache function, data would first need to be written to a file and then read back, with the cache simply passing the data buffer between the high-level API and the low-level callbacks, effectively simulating the action of caching the data (i.e. it is assumed that the cache can only hold a single chunk; any additional operations involving the cache would require eviction). This approach serves two primary purposes. First, it was necessary to test both the high-level and low-level code integration, which up to this point, remained untested (by necessity). Second, the V0 cache would provide exposure to the necessary internal HDF5 components that would further inform this redesign. With these goals in mind, the V0 project focused on a 1-dimensional dataset with five elements and a single chunk of the same size. This chunk would contain non-fill values at indexes 1, 2, and 3 to provide a quick, consistent point of reference.

To facilitate testing the high- and low-level code integration, the write process was the first to be focused on, through the development of the `H5SC_write()` function. Using chunk informa-

tion derived from first calling `H5SC__io_info_init()`, two key tasks are done; first the caching action for a dirty chunk is emulated, followed by a simulation of the eviction process. Upon calling `H5SC_write()`, `H5SC__io_info_init` is called, which sets up selections for each chunk in the I/O request. Once this preliminary chunk information is collected, we then look up the chunk on disk (which, by assumption, will result in a failure to find the chunk). Since the chunk is not found on disk, it should be fully overwritten, which leads to invoking the new chunk callback to initialize a new chunk. After the new chunk is created, the gather memory callback is called to collect the chunk data into a single buffer. Next, the "eviction" process begins. First, the chunk is encoded to ensure that it is transitioned from the in-memory format into the on-disk format. Next, the chunk is inserted into the file indexing using the insert callback function. Finally, the encoded chunk buffer is written to disk using the `H5F_block_write()` function. On-disk data verification was done using the Unix octal dump (od) tool.

After the write process was debugged and verified, the next step was to read the data back into a user buffer through developing the `H5SC_write()` function. Similar to the procedure used to write this initial chunk, we first collect information about the requested chunk through utilizing `H5SC__io_info_init()`. Using this information, the chunk is looked up (and this time around, it is expected to be present). If the chunk is found on disk, the chunk read process is initiated. Using the information collected with the lookup callback, `H5F_block_read()` is invoked to load the on-disk formatted chunk into a buffer. This buffer is then run through the decode callback, which translates the data into the in-memory format. Finally, this buffer is scattered into the available memory using the scatter memory callback.

As of the time of writing, the V0 cache supports reading and writing a single chunk of data. Currently, this V0 is being extended to support I/O that involves multiple chunks within a single dataset; however, considerations for how vector I/O will be supported within the full-feature version of the SCC have lead to minor delays in a more functional iteration of the V0 cache being made available. Additionally, there is currently a segmentation fault occurring when the library is closed which will need to be addressed sooner rather than not.