Shared Chunk Cache Design

Dr. Clay Carper
John Mainzer

December 19, 2024

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 near-essential at this point. Using Version 5 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].

2 Design Requirements and Considerations

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 a 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 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 considering data contiguousness.
- Considering the need for supporting variable-length data.
- 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 1 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 indexing of datasets (to my knowledge) is fixed, while the

[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.

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[3] 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[4]. 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, as opposed to eviction-upon-collision.

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[5]. 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[6]. 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, 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:* 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*: At the cost of increasing computational overhead, allows for validation of partial chunks though utilizing a checksum or data hash. Works to reduce the

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

[4] 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.

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

[6] 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.

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[7].

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.

With respect to supporting variable-length data, the value field associated with a cache entry will be a void pointer. 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[8].

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.

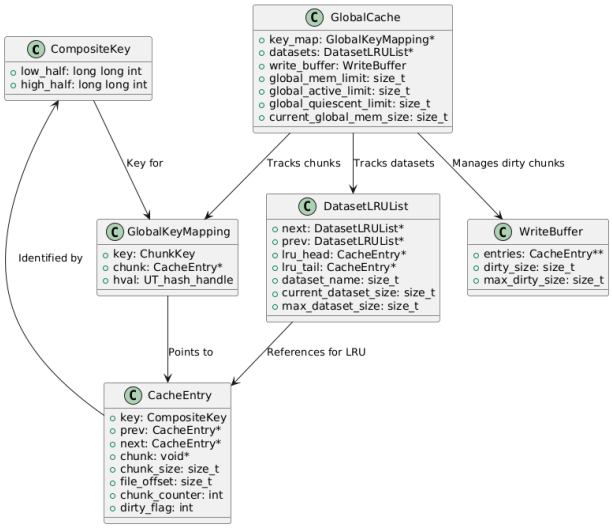

Figure 1: UML diagram illustrating the relationship between the proposed structures in the Shared Chunk Cache.

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

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

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.

```
typedef structure ChunkKey {
    long long int low_half;  /* Bottom half of the 128-bit int */

    long long int high_half; /* Top half 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`[9]. 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. Additionally, by operating on a 128-bit key using UTHash, it is highly unlikely collisions will occur, allowing for the hash table to be maintained at a minimal size.

3.2 Individual Cached 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 key mapping and linked to the appropriate dataset-specific LRU list.

```
typedef struct CacheEntry {
    ChunkKey data_key;       /* Key for the cache entry */

    struct CacheEntry *prev; /* Pointer to the previous node in
    the linked list */

    struct CacheEntry *next; /* Pointer to the next node in the
```

[9]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.

```
    linked list */

    void *chunk;              /* Pointer to in-memory data (NULL if not loaded) */

    size_t chunk_size;        /* Size of the cached data (post
    compression/conversion, check Neil's docs)*/

    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) */
} 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[10], 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.

3.3 Global Chunk Management with `GlobalKeyMapping`

- **Purpose:** Provides the global mechanism for constant time lookup, insertion, and removal of chunks.
- **Basic Data Type:** Hash Table (provided by UTHash)
- **Usage:** Centralized registry for all cached chunks, providing isolation from dataset-centered queries.

```
typedef  struct GlobalKeyMapping{
    ChunkKey data_key;   /* Unique key for the cached chunk */

    CacheEntry *chunk;   /* Pointer to the cached data chunk */

    UT_hash_handle hval; /* Handle for UTHash hash table */
} GlobalKeyMapping;
```

Further Commentary: To manage lookups across all current instances of `DatasetCache`, a global hash table will be utilized. This structure enables constant-time lookups while also enabling an association between specific `ChunkKey` and the corresponding `DatasetCache`, avoiding the need to transverse each cached dataset to determine which chunks are present within the cache. Further, having a centralized key management solution provides a consistent, single structure for evaluating what data is currently loaded.

[10] Based on past discussions during Technical Meetings and Neil's design document, all data conversions occur before adding data into the cache. Should this assumption be faulty, a distinction will need to be made to ensure the correct data is pointed to, with the correct sizing (i.e. both will need to be associated with the chunk cache memory format.

3.4 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.

```
typedef structure DatasetLRUList {
    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 */

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

    CacheEntry *lru_tail;        /* Pointer to the tail of the LRU list (least recently used c

    size_t dataset_name;         /* Identifier for the dataset associated with this LRU list *

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

    size_t max_dataset_size;       /* Maximum available size allocated for this  dataset */

} 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. 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.

3.5 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 Lists and a Buffer
- **Usage:** Provides the framework for enforcing global memory limits, coordination of chunk eviction, and managing dirty chunk writes.

```
typedef struct GlobalCache {

    GlobalKeyMapping *key_map;       /* Pointer to the hash table for global chunk tracking */
```

```
    DatasetLRUList *datasets;        /* Pointer to the head of the dataset-specific LRU lists

    WriteBuffer write_buffer;        /* Buffer for batching dirty chunks */

    size_t global_mem_limit;         /* Global cache memory limit */

    size_t global_active_limit;      /* Maximum size for active data
    (global) */

    size_t global_quiescent_limit;   /* Maximum size for inactive
    data (global) */

    size_t current_global_mem_usage; /* Current total cache size
    across all datasets */

} 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, `GlobalCache` is responsible for enforcing global memory limits, maintaining lists of all datasets, and the coordination of chunk eviction. In contrast to `GlobalKeyMapping`, which enables low-level chunk tracking though the hash table; serving as a light weight, focused data structure. This separation of responsibility is a key reason for the existence of two global data structures. Scalability is another key point of consideration for maintaining a separation between these two components. Should a more complex hash table be necessary for future extensions, the global key mapping could be replaced without a needing to alter the functionality of the global cache. Continuing this intention, when developing multi-thread safe extensions to this redesign, maintaining an isolation between the global cache and the chunk mapping structures will reduce the complexity of introducing thread pools for global operations and lock-free hash tables.

3.5.1 Write Buffer

- **Purpose:** Temporarily stores dirty chunks for batching write operations.
- **Basic Data Type:** Dynamic Array of Pointers
- **Usage:** Provides a method for minimizing to-disk write I/O by consolidating multiple dirty chunks.

```
typedef struct WriteBuffer {
    CacheEntry **entries;  /* Array of dirty chunks (to avoid
    duplication, we point directly to the existing object) */

    size_t dirty_size;     /* Size of the dirty chunks in the buffer */

    size_t max_dirty_size; /* Maximum size allocated to the dirty chunk buffer */
}WriteBuffer;
```

Further Commentary: Introducing a write buffer through the utilization of a dynamic array of pointers provides an in-cache method for isolating dirty chunks. This isolation will enable batched write-back operations without requiring transversal of the full list of cached items. Should this isolation be deemed unnecessary by the user, the flag indicating a chunk's cleanliness could simply be ignored.

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:

- A chunk read from file to the SCC
- Reading a chunk from within the SCC
- Writing a clean cached chunk to file from the SCC
- Writing a dirty cached chunk to file from the SCC
- The LRU eviction cycle

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[11].

4.1 From File Chunk Read

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

[11] 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. Cache Entry Creation:
    - A new `CacheEntry` is created for the chunk, using the specified key.
5. Adding to Dataset LRU:
    - A reference to the newly minted `CacheEntry` is added to the head of the LRU list associated dataset number 3 (an instance of `DatasetLRUList`). The current cache size parameter associated with this LRU list is updated, which is cascaded to the appropriate global cache size variables
6. Global Key Mapping Update:
    - The `GlobalKeyMapping` hash table is updated to reflect the newly added chunk via the associated `CacheEntry`
7. Data Retrieval from Source:
    - Disk I/O is invoked by the system to read the chunk from the file using the `file_offset` (using the avaialbe call backs)
    - At this stage, the chunk is loaded into the memory allocated for the associated `CacheEntry`

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. 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 `4321`
2. Global Key Lookup:
    - The system queries the `GlobalKeyMapping` using the chunk key `1234`
    - The key is found, resulting in a cache hit
3. Access the Entry:
    - The `CacheEntry` pointer is retrieved from the `GlobalKeyMapping`
4. Updating to Dataset LRU:
    - A reference to the accessed `CacheEntry` is moved to the head of the `DatasetLRUList` associated dataset number 2.
5. Return Data:
    - The cached data chunk is returned to the caller

4.3 Writing a Clean Chunk from within the SCC

Suppose we have a chunk of data identified by a computed chunk key value of 1122, and is a member of dataset number 4 which is currently within the SCC. 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. Data Request
    - The system request to write a data chunk with the computed chunk key of 1122
2. Global Key Lookup
    - The system quires the `GlobalKeyMapping` hash table using the chunk key 1122.
    - The key is found, resulting in a cache hit
3. Access the Entry
    - The `CacheEntry` pointer is retrieved from the `GlobalKeyMapping`
4. Modify the Chunk
    - The system modifies the data stored in the chunk associated with the `CacheEntry` using UTHash Macros as needed
5. Mark as Dirty
    - The `dirty` flag in the altered `CacheEntry` is set to 1 to indicate the in-memory chunk has been altered
    - A reference to the chunk is then added to the `WriteBuffer` in the `GlobalCache` to be batch written in the future
6. Write Buffer Management
    - When the `WriteBuffer` exceeds the count limit (or when triggered during eviction), the dirty chunk(s) are written back to disk in a single batched operation
    - After the batched write operation is completed, the `dirty` flag is cleared for each chunk and the write buffer is cleared
7. Update the LRU List(s):
    - For each chunk cleared, the `DatasetLRUList` instance for dataset 4 is updated by moving the flushed chunk(s) to the head of the linked list.

4.4 Writing a Dirty Chunk from within the SCC

Suppose we have a 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. 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. 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 key is found, resulting in a cache hit
3. Access the Entry
   - The `CacheEntry` pointer is retrieved from the `GlobalKeyMapping`
4. Modify the Chunk
   - The system modifies the data stored in the chunk associated with the `CacheEntry` using UTHash Macros as needed
5. Preserve Dirty Status
   - Since the dirty flag is already set, no additional action is taken (i.e. this chunk is already in the `WriteBuffer`)
6. Write Buffer Management
   - When the `WriteBuffer` exceeds the count limit (or when triggered during eviction), the dirty chunk(s) are written back to disk in a single batched operation
   - After the batched write operation is completed, the `dirty` flag is cleared for each chunk and the write buffer is cleared
7. Update the LRU List(s):
   - For the `CacheEntry` with chunk key 2211, the `DatasetLRUList` instance for dataset 8 is updated by moving the flushed chunk(s) to the head of the linked list.

4.5 The LRU Eviction Cycle

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[12]. 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

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

- 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 → PWCC → FWCC → FWDC)
   - Beginning with the chunk cache element at the tail, the eviction priority is assessed:
     - 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 `GlobalKey Mapping`, 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

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.