# Shared Chunk Cache Internal API

Neil Fortner

John Mainzer

10/17/24

To support the implementation of sparse chunk storage in HDF5, and to improve the performance of raw data I/O in general in HDF5, we intent to implement a new chunk cache that is shared among all datasets in a file. While the initial implementation will only be used by structured chunks, this cache will also support standard chunked and contiguous layouts, and any future layout that breaks the dataset into (hyper)rectangular chunks and stores these chunks separately in the file. It could also be extended to work with external datasets in the future.

Unlike the current chunk cache, which is internal to the chunk layout, and operates at a low level, being managed by the chunk code, we envision the shared chunk cache operating at a high level, managing and coordinating all raw data I/O. This will allow us to implement a more uniform approach to dataset I/O, allow for better coordination in the multi dataset I/O case, and simplify the eventual conversion to concurrent thread-safety in the cache, especially when enforcing ordering of I/O operations from multiple threads. During I/O, the shared chunk cache will be invoked directly from H5D__read() and H5D__write(), immediately after generic I/O initialization routines.

To implement this, there will need to be two API layers for the shared chunk cache, a top level interface that the library calls into, and a layout callback interface for when the shared chunk cache needs to perform layout-specific operations.

This is a work in progress and will almost certainly change.

## Data Formats

Any given piece of raw data can exist in three different formats: the application memory buffer format, the chunk cache memory format, and the disk format. The application memory buffer format is well understood from the existing HDF5 API: simply an array of elements in the memory datatype, with variable length data stored as hvl_t elements containing pointers to the variable length arrays. The chunk cache memory format is determined by the layout, but is generally considered to be stored in a deserialized version of the file format, uncompressed but in the file datatype. For the sparse chunk layout, the chunk cache memory format will likely consist of a structure containing the H5S_t describing the defined values and pointers to buffers for the different sections. For the legacy chunk format, the chunk cache memory format will be identical to the on disk format with the exception of compression. Finally, the on disk format is a serialized version of the chunk memory format and is how the chunk is stored in the file, with any compression, etc. applied. Data in the chunk cache will largely be held in the chunk cache memory format, with the exception that some chunks will be additionally held as buffers containing the on-disk format as chunks are preemptively encoded as they near the conditions for eviction from the cache. Writing data to one of these preemptively encoded chunks before eviction will then invalidate the file format buffer.

# Structures

While the details of the cache's structures are yet to be determined, we can make some broad statements about their general arrangement. The top level H5SC_t struct will contain general cache settings such as the preemption policy, memory limit, and others, as well as the current memory footprint, number of actual bytes used within that footprint, and an LRU (least recently used) list of chunks, that can be in any dataset, or possibly a different structure used for a different preemption method:

```
typedef struct H5SC_t {
        H5SC_preemption_policy_t preemption_policy;
        hsize_t memory_limit;
        <other options>;
        size_t nbytes_alloc;
        size_t nbytes_used;
        H5SC_chunk_t *LRU_head;
        H5SC_chunk_t *LRU_tail;
} H5SC_t;
```

Each dataset will also contain its own hash table (or possibly a different structure) used to index that datasets's cached chunks. This can take the form of a UT_hash_handle placed in the H5D_shared_t struct. The H5SC_chunk_t struct will need to contain the chunk buffer, the chunk's scaled coordinates, address, allocated size on disk, number of bytes allocated and used in memory, and whether it contains only information on the selected elements:

```
typedef struct H5SC_chunk_t {
        void *chunk;
        hsize_t scaled[H5S_MAX_RANK];
        haddr_t addr;
        hsize_t disk_size;
        size_t nbytes_alloc;
        size_t nbytes_used;
        bool contains_values;
} H5SC_chunk_t;
```

# Top Level API

These are the functions that are called by the upper layers of the dataset package in the HDF5 library and serve as the initial entry points to the H5SC (shared chunk cache) package.

```
H5SC_t *H5SC_create(H5F_t *file, hid_t fapl_id);
```

Creates a new, empty shared chunk cache. Will be called at file open time.

```
herr_t H5SC_destroy(H5SC_t *cache);
```

Destroys a shared chunk cache, freeing all data used. Does not flush chunks. Called at file close time.

```
herr_t H5SC_write(H5SC_t *cache, size_t count, H5D_dset_io_info_t *dset_info, H5D_io_type_info_t *io_type_info);
```

Writes raw data through the shared chunk cache. Called by H5D__write() after initial generic setup.

```
herr_t H5SC_read(H5SC_t *cache, size_t count, H5D_dset_io_info_t *dset_info, H5D_io_type_info_t *io_type_info);
```

Reads raw data through the shared chunk cache. Called by H5D__read() after initial generic setup.

```
herr_t H5SC_flush(H5SC_t *cache);
```

Flushes all cached data.

```
herr_t H5SC_flush_dset(H5SC_t *cache, H5D_t *dset, bool evict);
```

Flushes all data cached for a single dataset. If evict is true, also evicts all cached data.

```
herr_t H5SC_set_extent_notify(H5SC_t *cache, H5D_t *dset, hsize_t *old_dims);
```

Called after H5Dset_extent() has been called for a dataset, so the cache can recompute chunk indices, delete chunks, clear unused sections of chunks, etc.

```
herr_t H5SC_direct_chunk_read(H5SC_t *cache, H5D_t *dset, hsize_t *offset, void *buf);
```

Reads the chunk that starts at coordinates give by offset directly from disk to buf, without any decoding or conversion. First flushes that chunk if it is dirty in the cache.

```
herr_t H5SC_direct_chunk_write(H5SC_t *cache, H5D_t *dset, hsize_t *offset, void *buf);
```

Writes the chunk that starts at coordinates given by offset directly from buf to disk, without and encoding or conversion. First evicts that chunk from cache if it is present.

## Contents of H5D_dset_io_info_t

The H5SC_read() and H5SC_write() functions take a pointer to an H5D_dset_io_info_t struct. This is an existing structure that contains information about a single dataset in an I/O operation. The shared chunk cache will not need all of the information contained in this struct, and will need a bit more than is currently there. Here is the current definition of the struct:

```
typedef struct H5D_dset_io_info_t {
   H5D_t              *dset;      /* Pointer to dataset being operated on */
   H5D_storage_t        *store;     /* Dataset storage info */
   H5D_layout_ops_t       layout_ops; /* Dataset layout I/O operation function pointers */
   H5_flexible_const_ptr_t buf;       /* Buffer pointer */

   H5D_io_ops_t io_ops; /* I/O operations for this dataset */

   H5O_layout_t *layout; /* Dataset layout information*/
   hsize_t      nelmts; /* Number of elements selected in file & memory dataspaces */

   H5S_t *file_space; /* Pointer to the file dataspace */
   H5S_t *mem_space;  /* Pointer to the memory dataspace */

   union {
      struct H5D_chunk_map_t *chunk_map;        /* Chunk specific I/O info */
      H5D_piece_info_t      *contig_piece_info; /* Piece info for contiguous dataset */
   } layout_io_info;

   const H5T_t    *mem_type; /* memory datatype */
   H5D_type_info_t type_info;
   bool          skip_io; /* Whether to skip I/O for this dataset */
} H5D_dset_io_info_t;
```

In addition, we plan to initialize type conversion before calling the shared chunk cache. Therefore, we will also need to fill in the type_info field, except for the request_nelemts field within type_info. Here is the definition of H5D_type_info_t:

```
typedef struct H5D_type_info_t {
   /* Initial values */
   const H5T_t *mem_type;  /* Pointer to memory datatype */
   const H5T_t *dset_type; /* Pointer to dataset datatype */
   const H5T_t *src_type;  /* Pointer to source datatype */
   const H5T_t *dst_type;  /* Pointer to destination datatype */
   H5T_path_t  *tpath;     /* Datatype conversion path */

   /* Computed/derived values */
   size_t              src_type_size;  /* Size of source type */
   size_t              dst_type_size;  /* Size of destination type */
   bool                is_conv_noop;   /* Whether the type conversion is a NOOP */
   bool                is_xform_noop;  /* Whether the data transform is a NOOP */
   const H5T_subset_info_t *cmpd_subset;    /* Info related to the compound subset conversion functions */
   H5T_bkg_t              need_bkg;      /* Type of background buf needed */
   size_t              request_nelmts; /* Requested strip mine */
} H5D_type_info_t;
```

Finally, there is some type conversion info that is global to the I/O instead of being specific to a singel dataset. These will need to be passed in a separate struct. This has some overlap with the existing H5D_io_info_t struct so we could use that, but it is probably better to create a new struct:

```
typedef struct H5D_io_type_info_t {
    uint8_t              *tconv_buf;           /* Datatype conv buffer */
    bool                 tconv_buf_allocated;   /* Whether the type conversion buffer was allocated */
    size_t               tconv_buf_size;       /* Size of type conversion buffer */
    uint8_t              *bkg_buf;             /* Background buffer */
    bool                 bkg_buf_allocated;    /* Whether the background buffer was allocated */
    size_t               bkg_buf_size;         /* Size of background buffer */
    H5D_vlen_buf_info_t    vlen_buf_info;        /* Vlen data buffer and info */
    bool must_fill_bkg; /* Whether any datasets need a background buffer filled with destination contents */
    bool may_use_in_place_tconv; /* Whether datasets in this I/O could potentially use in-place type
                              conversion if the type sizes are compatible with it */
} H5D_io_type_info_t;
```

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.

# Layout Callbacks

These are the callback functions that individual layout types implement in order to enable the shared chunk cache to perform these operations in a layout-agnostic manner. Some of these functions accept an hsize_t *parameter called scaled. This is an array containing the scaled coordinates of the chunk, where the coordinates are divided by the chunk dimensions so that adjacent chunks differ in their scaled coordinates by a value of one. In addition, in order for the shared chunk cache to calculate the logical locations of the chunks, the chunk dimensions will be promoted to the H5D_shared_t struct and made available for all layout types where it is valid (contiguous datasets will simply use the dataset dimensions). The layout will also need to specify whether it uses the legacy vlen/reference storage where data is stored outside the dataset, or whether the variable length data is stored in a separate section of the chunk.

```
typedef herr_t (*H5SC_chunk_lookup_t)(H5D_t *dset, hsize_t *scaled /*in*/, haddr_t *addr /*out*/, hsize_t *size
/*out*/, hsize_t *defined_values_size /*out*/, size_t *size_hint /*out*/, size_t *defined_values_size_hint /*out*/, void
**udata /*out*/);
```

Looks up chunk address and size on disk. defined_values_size is the number of bytes to read if only the list of defined values is needed. size_hint is the suggested allocation size for the chunk (could be larger if the chunk might expand when decoded). defined_values_size_hint is the suggested allocation size if only the list of defined values is needed. If *defined_values_size is returned as 0, then all values are defined for the chunk. In this case, the chunk may still be decoded without reading from disk, by allocating a buffer of size defined_valued_size_hint and passing it to H5SC_chunk_decode_t with *nbytes_used set to 0. *udata can be set to anything and will be passed through to H5SC_chunk_decode_t and/or the selection or vector I/O routines, then freed with free() (we will create an H5SC_free_udata_t callback if necessary).

```
typedef herr_t (*H5SC_chunk_decode_t)(H5D_t *dset, size_t *nbytes /*in,out*/, size_t *alloc_size /*in,out*/, void
**chunk /*in,out*/, void *udata);
```

Decompresses/decodes the chunk from file format to memory cache format if necessary. Reallocs chunk buffer if necessary. On entry, nbytes is the number of bytes used in the chunk buffer. On exit, it shall be set to the total number of bytes used (not allocated) across all buffers for this chunk. On entry, alloc_size is the size of the chunk buffer. On exit, it shall be set to the total number of bytes allocated across all buffers for this chunk. Optional, if not present, chunk is the same in cache as on disk.

```
typedef herr_t (*H5SC_chunk_decode_defined_values_t)(H5D_t *dset, size_t *nbytes /*in,out*/, size_t *alloc_size
/*in,out*/, void **chunk /*in,out*/);
```

The same as H5SC_chunk_decode_t but only decodes the defined values. Optional, if not present, all values are defined.

```
typedef herr_t (*H5SC_new_chunk_t)(H5D_t *dset, bool fill, size_t *nbytes /*out*/, size_t *buf_size /*out*/, void
**chunk /*chunk*/);
```

Creates a new empty chunk. Does not insert into on disk chunk index. If fill is true, writes the fill value to the chunk (unless this is a sparse chunk). The number of bytes used is returned in *nbytes and the size of the chunk buffer is returned in *buf_size.

```
typedef herr_t (*H5SC_chunk_condense_t)(H5D_t *dset, size_t *nbytes /*in, out*/. void **chunk /*in, out*/);
```

Reallocates buffers as necessary so the total allocated size of buffers for the chunk (alloc_size) is equal to the total number of bytes used (nbytes). Optional, if not present the chunk cache will be more likely to evict chunks if there is wasted space in the buffers.

```
typedef herr_t (*H5SC_chunk_encode_t)(H5D_t *dset, hsize_t *write_size /*out*/, hsize_t *write_buf_alloc /*out*/,
const void *chunk, void **write_buf /*out*/);
```

Compresses/encodes the chunk as necessary. If chunk is the same as cache_buf, leaves *write_buf as NULL. This function leaves chunk alone and allocates write_buf if necessary to hold compressed data, sets *write_size to the size of the data in write_buf, and sets *write_size_alloc to the size of write_buf, if it was allocated.

```
typedef herr_t (*H5SC_chunk_evict_t)(H5D_t *dset, void *chunk);
```

Frees chunk and all memory referenced by it. Optional, if not present free() is simply used.

```
typedef herr_t (*H5SC_chunk_encode_in_place_t)(H5D_t *dset, size_t *write_size /*out*/, void **chunk /*in,out*/);
```

The same as H5SC_chunk_encode_t but does not preserve chunk buffer, encoding is performed in-place. Must free all other data used.

```
typedef herr_t (*H5SC_chunk_insert_t)(H5D_t *dset, size_t count, hsize_t *scaled[] /*in*/, haddr_t *addr[] /*in,out*/,
hsize_t old_disk_size[], hsize_t new_disk_size[], void *chunk[] /*in*/);
```

Inserts (or reinserts) count chunks into the chunk index if necessary. Old address and size (if any) of the chunks on disk are passed as addr and old_disk_size, the new size is passed in as new_disk_size. This function resizes and reallocates on disk if necessary, returning the address of the chunks on disk in *addr. If an element in chunk is passed as NULL then this function shall insert a chunk large enough and with properties set to (initially) hold only fill values.

```
typedef herr_t (*H5SC_chunk_selection_read_t)(H5D_t *dset, H5S_t *file_space_in, void *chunk /*in*/, H5S_t
**file_space_out /*out*/, bool *select_possible /*out*/, void *udata);
```

For when the chunk cache wants to read data directly from the disk to the user buffer. If not possible due to compression, etc, returns select_possible=false. Otherwise transforms the file space if necessary to describe the selection in the on disk format (returns transformed space in file_space_out). If no transformation is necessary, leaves *file_space_out as NULL. chunk may be passed as NULL, and may also be an in-cache chunk that only contains information on selected elements. Optional, if not present, chunk I/O is only performed on entire chunks or with vector I/O. The H5SC code checks for type conversion before calling this.

```
typedef herr_t (*H5SC_chunk_vector_read_t)(H5D_t *dset, haddr_t addr, H5S_t *file_space_in, void *chunk /*in*/, size_t
*vec_count /*out*/, haddr_t **offsets /*out*/, size_t **sizes /*out*/, bool *vector_possible /*out*/, void *udata);
```

For when the chunk cache wants to read data directly from the disk to the user buffer, using vector I/O. If not possible due to compression, etc, returns vector_possible=false. Otherwise returns the vector of selected elements in offsets (within the file, not the chunk, this is why addr is passed in) and sizes, with the number of vectors returned in vec_count. chunk may be passed as NULL, and may also be an in-cache chunk that only contains information on selected elements. Optional, if not present, chunk I/O is only performed on entire chunks or with selection I/O. The H5SC code checks for type conversion before calling this.

```
typedef herr_t (*H5SC_chunk_selection_write_t)(H5D_t *dset, H5S_t *file_space_in, void *chunk /*in*/, H5S_t
*file_space_out /*out*/, bool *select_possible /*out*/, void *udata);
```

For when the chunk cache wants to write data directly from the user buffer to the cache. If not possible due to compression, etc, returns select_possible=false. Otherwise transforms the file space if necessary to describe the selection in the on disk format (returns transformed space in file_space_out). If no transformation is necessary, leaves *file_space_out as NULL. chunk may be passed as NULL, and may also be an in-cache chunk that only contains information on selected elements. Optional, if not present, chunk I/O is only performed on entire chunks or with vector I/O. The H5SC code checks for type conversion before calling this.

```
typedef herr_t (*H5SC_chunk_vector_write_t)(H5D_t *dset, haddr_t addr, H5S_t *file_space_in, void *chunk /*in*/, size_t *vec_count /*out*/, haddr_t **offsets /*out*/, size_t **sizes /*out*/, bool *vector_possible /*out*/, void *udata);
```

For when the chunk cache wants to write data directly from the user buffer to the cache. If not possible due to compression, etc, returns vector_possible=false. Otherwise returns the vector of selected elements in offsets (within the file, not the chunk, this is why addr is passed in) and sizes, with the number of vectors returned in vec_count. chunk may be passed as NULL, and may also be an in-cache chunk that only contains information on selected elements. Optional, if not present, chunk I/O is only performed on entire chunks or with selection I/O. The H5SC code checks for type conversion before calling this.

```
typedef herr_t (*H5SC_chunk_scatter_mem_t)(H5D_io_info_t *io_info, H5D_dset_io_info_t *dset_info, H5S_t *mem_space, H5S_t *file_space, const void *chunk);
```

Scatters data from the chunk buffer into the memory buffer (in dset_info), performing type conversion if necessary. file_space's extent matches the chunk dimensions and the selection is within the chunk. mem_space's extent matches the entire memory buffer's and the selection within it is the selected values within the chunk, offset appropriately within the full extent. Optional, if not present, chunk is the same in memory as it is in cache, with the exception of type conversion (which will be handled by the H5SC layer). If the layout stores variable length data within the chunk this callback must be defined.

```
typedef herr_t (*H5SC_chunk_gather_mem_t)(H5D_io_info_t *io_info, H5D_dset_io_info_t *dset_info, H5S_t *mem_space, H5S_t *file_space, size_t *nbytes /*in,out*/, size_t *alloc_size /*in,out*/, size_t *buf_size_total /*in,out*/, void *chunk);
```

Gathers data from the memory buffer (in dset_info) into the chunk buffer, performing type conversion if necessary. file_space's extent matches the chunk dimensions and the selection is within the chunk. mem_space's extent matches the entire memory buffer's and the selection within it is the selected values within the chunk, offset appropriately within the full extent. Defines selected values in the chunk. Optional, if not present, chunk is the same in memory as it is in cache, with the exception of type conversion (which will be handled by H5SC layer). If the layout stores variable length data within the chunk this callback must be defined.

```
typedef herr_t (*H5SC_chunk_fill_t)(H5D_io_info_t *io_info, H5D_dset_io_info_t *dset_info, H5S_t *space, size_t *nbytes /*in,out*/, size_t *alloc_size /*in,out*/, size_t *buf_size_total /*in,out*/, void *chunk);
```

Propagates the fill value into the selected elements of the chunk buffer, performing type conversion if necessary. space's extent matches the chunk dimensions and the selection is within the chunk. Optional, if not present, chunk is the same in memory as it is in cache, with the exception of type conversion (which will be handled by H5SC layer). If the layout stores variable length data within the chunk this callback must be defined.

```
typedef herr_t (*H5SC_chunk_defined_values_t)(H5D_t *dset, H5S_t *selection, void *chunk, H5S_t **defined_values /*out*/);
```

Queries the defined elements in the chunk. selection may be passed as H5S_ALL. These selections are within the logical chunk. Optional, if not present, all values are defined.

```
typedef herr_t (*H5SC_chunk_erase_values_t)(H5D_t *dset, H5S_t *selection, size_t *nbytes /*in,out*/, size_t *alloc_size /*in,out*/, void *chunk, bool *delete_chunk /*out*/);
```

Erases the selected elements in the chunk, causing them to no longer be defined. If all values in the chunk are erased and the chunk should be deleted, sets *delete_chunk to true, causing the cache to delete the chunk from cache, free it in memory using H5SC_chunk_evict_t, and delete it on disk using H5SC_chunk_delete_t. These selections are within the logical chunk. Optional, if not present, the fill value will be written to the selection using H5SC_chunk_fill_t.

```
typedef herr_t (*H5SC_chunk_evict_values_t)(H5D_t *dset, size_t *nbytes /*in,out*/, size_t *alloc_size /*in,out*/, void *chunk);
```

Frees the data values in the cached chunk and memory used by them (but does not reallocate - see H5SC_chunk_condense_t), but leaves the defined values intact. Optional, if not present the entire chunk will be evicted.

```
typedef herr_t (*H5SC_chunk_delete_t)(H5D_t *dset, hsize_t *scaled /*in*/, haddr_t addr, hsize_t disk_size);
```

Removes the chunk from the index and deletes it on disk. Only called if a chunk goes out of scope due to H5Dset_extent() or of H5SC_chunk_erase_values_t returns *delete_chunk == true.

## Callback struct

The final callback structure for each layout class is therefore:

```
typedef H5SC_layout_ops_t {
    H5SC_chunk_lookup_t                  lookup;
    H5SC_chunk_decode_t                  decode;
    H5SC_chunk_decode_defined_values_t   decode_defined_values;
    H5SC_new_chunk_t                              new_chunk;
    H5SC_chunk_condense_t                condense;
    H5SC_chunk_encode_t                  encode;
    H5SC_chunk_evict_t                   evict;
```

```
        H5SC_chunk_encode_in_place_t        encode_in_place;
        H5SC_chunk_insert_t                 insert;
        H5SC_chunk_selection_read_t         selection_read;
        H5SC_chunk_vector_read_t            vector_read;
        H5SC_chunk_selection_write_t        selection_write;
        H5SC_chunk_vector_write_t           vector_write;
        H5SC_chunk_scatter_mem_t            scatter_mem;
        H5SC_chunk_gather_mem_t                     gather_mem;
        H5SC_chunk_fill_t                           fill;
        H5SC_chunk_defined_values_t         defined_values;
        H5SC_chunk_erase_values_t           erase_values;
        H5SC_chunk_evict_values_t           evict_values;
        H5SC_chunk_delete_t                 delete;
} H5SC_layout_ops_t;
```

# Code Flow Examples

## Raw Data Write

H5D__write() will perform initial setup, then call H5SC_write(). The chunk cache will, for each chunk, check if it is in cache, if not it will look up the chunk with H5SC_chunk_lookup_t. If the lookup finds the chunk on disk, and the full chunk is not being overwritten, the cache will read the chunk from disk then decode it to chunk cache memory format with H5SC_chunk_decode_t. Multiple chunks could be loaded at once using vector I/O. If the lookup does not find the chunk or it is being fully overwritten, a new chunk will be created with H5SC_new_chunk_t, with fill set to false if it is being fully overwritten and true otherwise. The data will then be written to the in cache chunk using H5SC_chunk_gather_mem_t, performing any type conversion necessary.

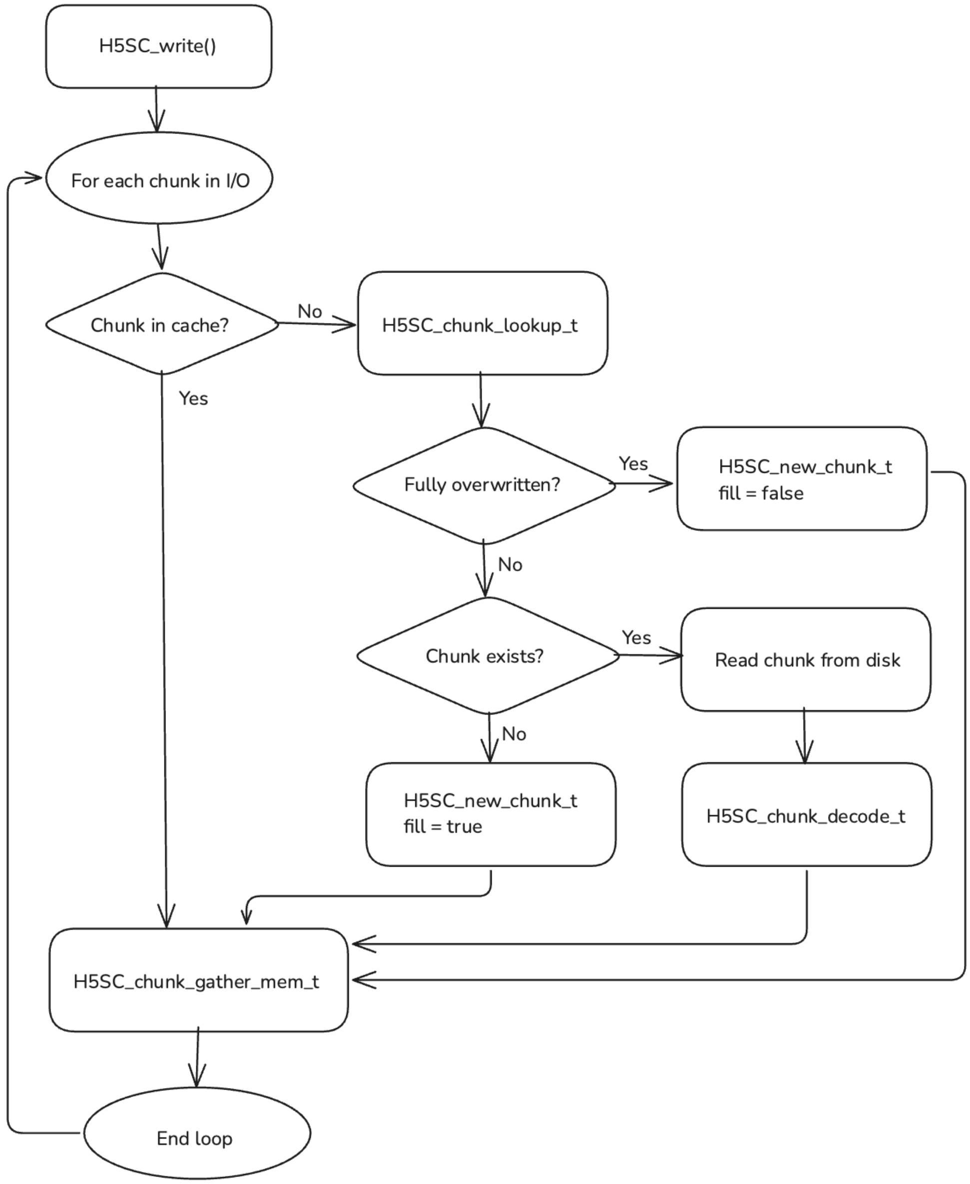


*Figure 1: H5SC_write()*

As the cache starts to get full, it will intelligently assign some dirty chunks to preemptively encode the on-disk format (including compression if specified) using H5SC_chunk_encode_t. Once the cache is full, it will pick chunks to evict. For any such chunks that are dirty, it will, if H5SC_chunk_encode_t was called, evict the chunk with H5SC_chunk_evict_t, (re)insert the chunk into the index with H5SC_chunk_insert_t, write the data from the previously encoded write buffer to disk, then free the write buffer. If H5SC_chunk_pre_flush_t was not called, the cache will call H5SC_encode_in_place_t, (re)insert the chunk into the index with H5SC_chunk_insert_t, write the write buffer to disk, then free the write buffer. Clean chunks will simply be evicted with H5SC_chunk_evict_t.

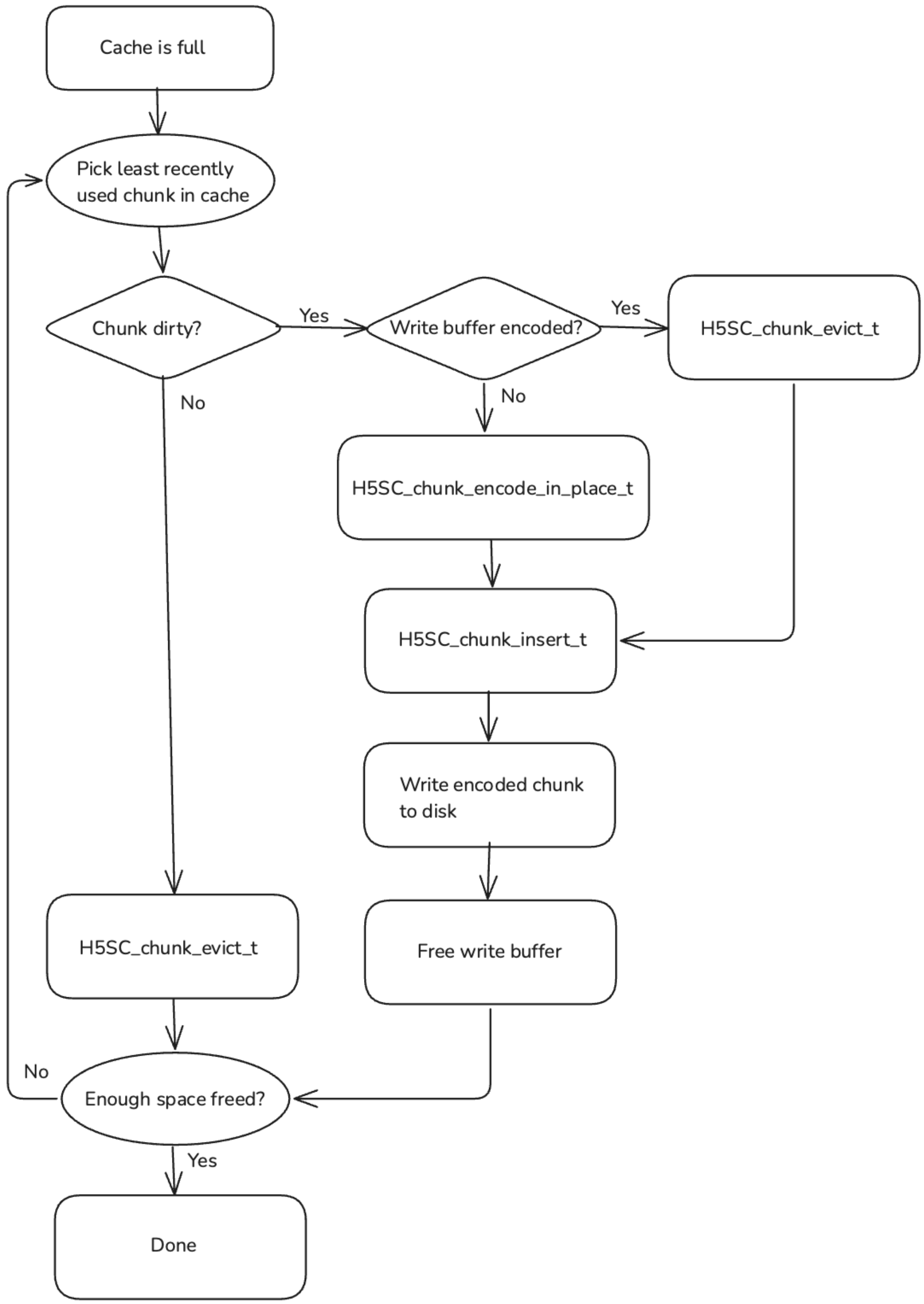


*Figure 2: Chunk cache eviction*

# Raw Data Read

H5D__read() will perform initial setup, then call H5SC_read(). The chunk cache will, for each chunk, check if it is in cache, if not look up the chunk with H5SC_chunk_lookup_t. If the lookup finds the chunk on disk, the cache will read the chunk from disk then decode it to chunk cache memory format with H5SC_chunk_decode_t. Multiple chunks could be loaded at once using vector I/O. The data from the chunk in cache will be scattered to the memory buffer using H5SC_chunk_scatter_mem_t. If the lookup does not find the chunk, the cache will propagate the fill value to the selected elements in the memory buffer.

## Raw Data Write (Skip Cache)

If a raw data write operation will skip the cache for one or more chunks involved in I/O, either due to a user request, the chunk being too big, or if the cache decides it's best for some other reason, the shared chunk cache code will, for each dataset involved, check if H5SC_chunk_selection_write_t and/or H5SC_chunk_vector_write_t is defined. The cache will then iterate over chunks involved in the I/O that will skip the cache. For each chunk the cache will first look up the chunk's address with H5SC_chunk_lookup_t. If the chunk does not exist on disk or neither H5SC_chunk_selection_write_t nor H5SC_chunk_vector_write_t are defined, the shared chunk cache will take the same actions as if the chunk cache were not being skipped except it will flush and evict the chunk immediately before moving on to the next chunk. Otherwise, if H5SC_chunk_selection_write_t is defined the shared chunk cache will invoke it with the correct file selection, and, if select_possible is returned as true, issue a low level selection I/O request with the file selection returned and previously calculated memory selection (or add to a larger selection I/O op to issue later to cover all chunks or datasets). Otherwise, the shared chunk cache will similarly invoke H5SC_chunk_vector_write_t with the correct file selection, and, if vector_possible is returned as true, calculate memory vectors to match the returned file vectors and issue a low level vector I/O call with these vectors (or add to a larger vector I/O op to issue later to cover all chunks or datasets).

If type conversion is required, the shared chunk cache will first check if the entire selection can fit in the type conversion buffer. If it can, it will proceed as above except the contents of the buffer will be gathered to the type conversion buffer, converted, and then this type conversion buffer will be passed as a contiguous source buffer to the low level selection or vector write routine (or add to a larger selection or vector I/O op if there is enough room in the type conversion buffer). If the selection cannot fit in the type conversion buffer, the shared chunk cache will only use the vector write callback, and if available it will process elements in batches up to the type conversion buffer size using a similar algorithm to the existing routine H5D__scatgath_write().

We could add a memory usage optimization to, if the chunk does not exist yet, only fill part of the chunk at a time. This will not be necessary to duplicate any existing behavior though, since contiguous datasets will allocate and fill the data iteratively using an existing separate pathway (H5D__alloc_storage) before the code reaches the shared chunk cache, and chunked datasets allocate and fill the entire chunk at once.

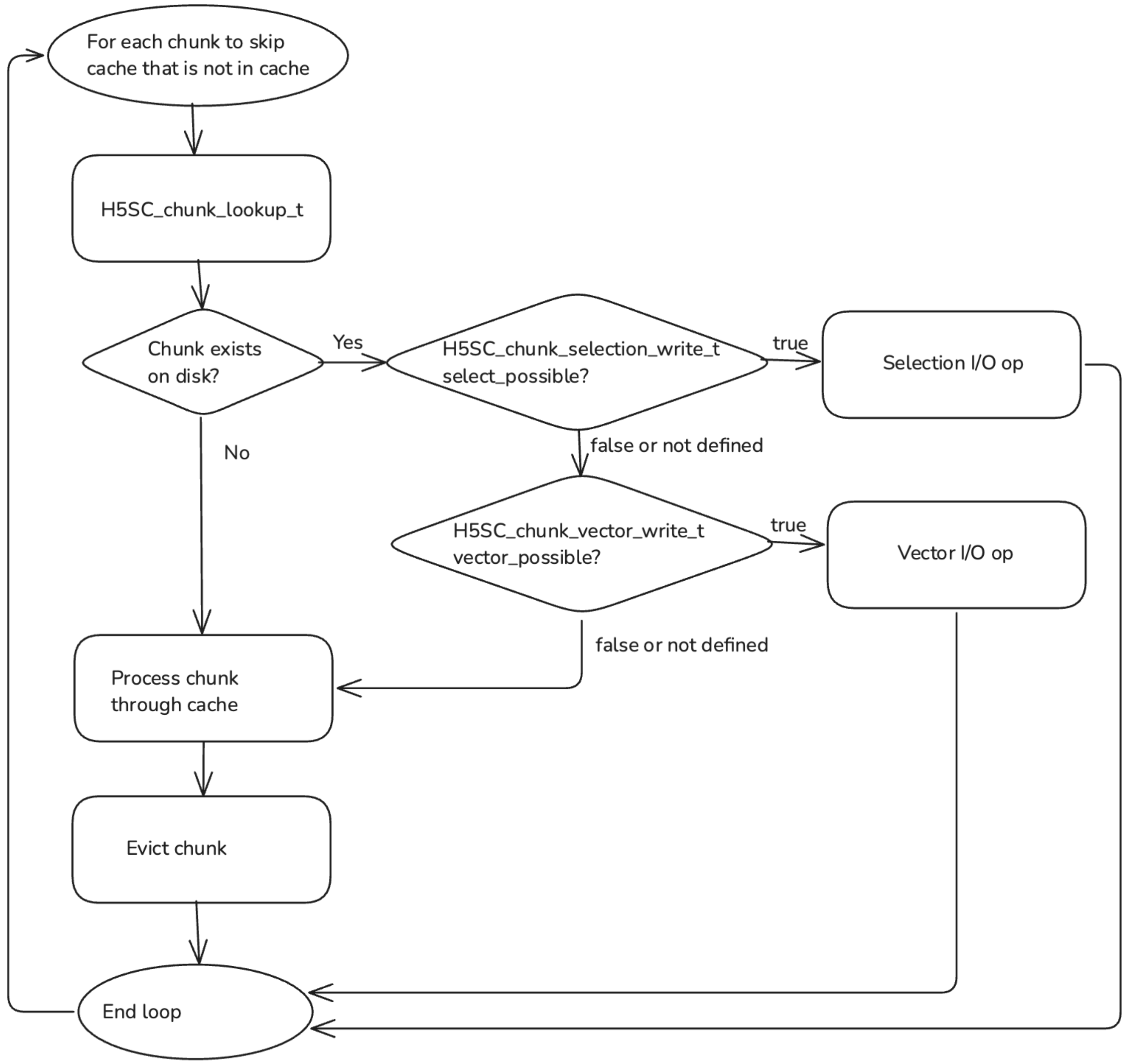


*Figure 3: Raw data write bypassing cache*

## Raw Data Read (Skip Cache)

If a raw data read operation will skip the cache for one or more chunks involved in I/O, either due to a user request, the chunk being too big, or if the cache decides it's best for some other reason, the shared chunk cache code will, for each dataset involved, check if H5SC_chunk_selection_read_t and/or H5SC_chunk_vector_read_t is defined. The cache will then iterate over chunks involved in the I/O that will skip the cache. For each chunk the cache will first look up the chunk's address with H5SC_chunk_lookup_t. If the chunk does not exist on disk the shared chunk cache will propagate the fill value to the matching selected parts of the user buffer. Otherwise, if H5SC_chunk_selection_read_t is defined the shared chunk cache will invoke it with the correct file selection, and, if select_possible is returned as true, issue a low level selection I/O request with the file selection returned and previously calculated memory selection (or add to a larger selection I/O op to issue later to cover all chunks or datasets). Otherwise, if H5SC_chunk_vector_read_t is defined the shared chunk cache will similarly invoke it with the correct file selection, and, if vector_possible is returned as true, calculate memory vectors to match the returned file vectors and issue a low level vector I/O call with these vectors (or add to a larger vector I/O op to issue later to cover all chunks or datasets). Otherwise, the shared chunk cache will take the same actions as if the chunk cache were not being skipped except it will flush and evict the chunk immediately before moving on to the next chunk.

If type conversion is required, the shared chunk cache will first check if the entire selection can fit in the type conversion buffer. If it can, it will proceed as above except the type conversion buffer will be passed as a contiguous destination buffer to the low level selection or vector read routine (or add to a larger selection or vector I/O op if there is enough room in the type conversion buffer), then the type conversion buffer will be converted and the contents will be scattered to the application memory buffer buffer. If the selection cannot fit in the type conversion buffer, the shared chunk cache will only use the vector read callback, and if available it will process elements in batches up to the type conversion buffer size using a similar algorithm to the existing routine H5D__scatgath_read().

## H5Dset_extent()

When performing a set extent operation, the upper levels of the library will first change the dataset struct, then make the H5SC_set_extent_notify() call. The shared chunk cache will then iterate over all of that dataset's chunks in cache, and recompute their chunk index. If the chunk has been completely removed from the extent, it will then be deleted on disk with the H5SC_chunk_delete_t callback and evicted from cache using the H5SC_chunk_evict_t callback and internal cache code to manage the cache structure. If the chunk is not completely removed from the extent but contains elements that are no longer in the extent, then, if H5SC_chunk_erase_values_t is defined, it will be called with the newly out of bounds elements as the selection. If H5SC_chunk_erase_values_t is not defined, then, if appropriate for the fill value/time settings, the fill value (or zero) will be written to the newly out of bounds elements using the H5SC_chunk_scatter_mem_t callback.

Next, the cache will iterate over all chunks that contain elements that are no longer in the extent (see H5D__chunk_prune_by_extent() for an example). If the chunk is in cache it is skipped, since it was already handled. Otherwise, if the chunk is still partly within the extent and (H5SC_chunk_erase_values_t is defined or (both H5SC_chunk_selection_write_t and H5SC_chunk_vector_write_t are not defined)), the chunk will be saved to a linked list of chunks to process (since the processing may cause chunks to be evicted from cache and subvert the selection criteria). It will then initiate a second pass over this linked list and, for each chunk, check if it exists on disk with H5SC_chunk_lookup_t. If it does not exist, processing can move to the next chunk, otherwise, it will be read and decoded using H5SC_chunk_decode_t. Next, if H5SC_chunk_erase_values_t is defined it will be invoked using the selection of newly out of bounds elements, otherwise the fill value will be written using H5SC_chunk_fill_t. If H5SC_chunk_erase_values_t is not defined and H5SC_chunk_selection_write_t or H5SC_chunk_vector_write_t is defined, only a single pass is necessary, and the fill value will be written directly to the newly out of bounds elements of each chunk using the procedure outlined above for raw data write (skip cache), or if the chunk is no longer within the extent it will simply be deleted with H5SC_chunk_delete_t.

Finally, if early allocation is enabled, the cache will, if appropriate for the fill value settings, create a new chunk with the fill value using H5SC_new_chunk_t and encode it to on disk format using H5SC_chunk_encode_in_place_t. Next, the cache will iterate over all chunks that are newly within the extent, and allocate and insert each on disk using H5SC_chunk_insert_t, then, if the fill value chunk was created, write it to disk at the address returned by H5SC_chunk_insert_t. In the case of legacy vlen or reference types that store data elsewhere, the cache will need to avoid calling H5SC_chunk_encode_in_place_t at the start, and instead maintain a buffer of a memory type fill value chunk and convert and encode it anew for each chunk to be created, possibly using something like H5D__fill_refill_vl().

## Early Allocation

Early allocation will be not be handled by the shared chunk cache, each layout type will implement it separately and no chunks will be cached by this operation.

# Appendix

## External Datasets

We would like to be able to extend the shared chunk cache to support external datasets. To do this, we recommend first modifying the external dataset code to use the H5FD layer to interact with external datasets. We may also want to add public API functions to allow the user to specify the file driver to use for the external data file. This change will cause an H5FD_t * to be stored within the H5D_shared_t struct (possibly in a nested struct), which can then be made visible to the shared chunk cache, either by placing the H5FD_t * in a uniform place or it can be returned through the low level API, possibly by adding an H5FD_t ** to H5SC_chunk_lookup_t (non-external datasets would return NULL). The shared chunk cache will then proceed as normal, and whenever it needs to perform I/O to or from the disk it will simply use this H5FD_t * instead of the one associated with the dataset's (and cache's) file.

## Type Conversion

With the new sparse chunk format introducing a new way to store variable length and reference data types, we must reconfigure the internal datatype conversion interface to be able to handle this. To do this, we can add a new value to the H5T_loc_t enum so it looks something like:

```
typedef enum {
   H5T_LOC_BADLOC = 0,      /* invalid datatype location */
   H5T_LOC_MEMORY,          /* data stored in memory */
   H5T_LOC_DISK_VL_GHEAP,   /* data stored on disk, with variable length data in a global heap */
   H5T_LOC_DISK_VL_INPLACE, /* data stored on disk, with variable length data co-located */
   H5T_LOC_MAXLOC           /* highest value (Invalid as true value) */
} H5T_loc_t;
```

The new sparse chunk format will then use H5T_LOC_DISK_VL_INPLACE. At least initially, we will not allow conversion from disk to disk. Data conversion between disk_vl_inplace and memory will then involve an additional buffer, which will be used to store the variable length data for the data in disk (cache) format. This buffer will then need to be passed to H5T_convert, conversion callbacks, and made visible to the public API. To keep track of this buffer and information related to it, we can introduce a new struct:

```
typedef struct H5T_vlen_buf_info_t {
   void  *buf;
   size_t nybtes_alloc;
   size_t nbytes_used;
} H5T_vlen_buf_info_t;
```

This will allow the type conversion code to reallocate the buffer if needed to fit more variable length data. Since the reallocation will only happen when converting from memory to cache format, the variable length data will already be in memory and excessive memory usage should not be a major issue. We may implement a first pass in the conversion step to determine the needed size of the vlen buffer. Here is the proposed signature for H5T_convert():

```
herr_t H5T_convert(H5T_path_t *tpath, const H5T_t *src_type, const H5T_t *dst_type, size_t nelmts, size_t buf_stride,
size_t bkg_stride, void *buf, void *bkg, H5T_vlen_buf_info_t *vlen_buf_info);
```

The need to potentially reallocate the covnersion buffer makes adding a public API function analogous to H5Pset_buffer() more complicated, but we could do so by passing the address of a buffer and an optional realloc callback.