RFC: Page Buffer VFD

Kyle Lofthus kyle.lofthus@lifeboat.llc
Aijun Hall aijun.ahhl@lifeboat.llc
John Mainzer john.mainzer@lifeboat.llc

The primary purpose of the Page Buffer VFD is to convert random I/O requests to paged I/O requests. This allows the use of VFDs that require paged I/O lower in the VFD stack.

The current implementation of the page buffer uses a hash table to index pages in the page buffer, and supports the least recently used (LRU) and first in first out (FIFO) replacement policies.

1 Introduction

The Page Buffer VFD was implemented to convert random I/O to paged I/O. This conversion allows the use of VFDs lower in the stack that require page-based I/O. The encryption VFD is the first use case, but it will likely not be the last.

Page buffers are also used to cache frequently used data, thereby reducing I.O traffic and improving performance. While it is possible that the Page Buffer VFD could be beneficial in this way, it was not the primary purpose, and no effort has been made to test what, if any performance benefit it provides.

The current implementations of the allocation and free VFD callbacks cause space allocation failures in the upper library. This will be dealt with in a future iteration, as some VFDs will require it (split and multi VFDs). However, since targeting just the Sec2 VFD is sufficient for the Phase I prototype, we are bypassing the issue for now.

2 Conceptual Overview

The Page Buffer VFD employs two primary data structures: a page header structure, an instance of which is associated with each page in the page buffer, and a main structure, which contains both a hash table and the base pointers of a doubly linked list used in replacement policy management.

The hash table is used to index the valid page headers that currently reside within the page buffer for quick retrieval. The hash table uses a simple hash function. The lower order bits of the page's address are discarded, and the remaining bits are right shifted. The result is taken modulo by the number of buckets in the hash table and this determines which bucket the page header structure is stored in. Hash table collisions are managed via chaining.

The doubly linked list is used by the current replacement policy to select the next candidate for eviction. When the maximum number of pages (and therefore page header structures) has been reached and a new page must be added to the page buffer, the replacement policy selects an eviction candidate, flushes the associated page if dirty, evicts it, and re-uses its page header structure to store the new page in the page buffer.

At present, the page buffer uses the least recently used (LRU) replacement policy as the default, but first-in first-out (FIFO) is also available.

Incoming random I/O requests can be thought of as having three parts – the address or offset from the beginning of the file in bytes, the size of the request in bytes, and the buffer containing the incoming or outgoing data.

To convert random I/O to paged I/O, a random I/O request will be broken into at most three pieces.

- If the I/O request does not start on a page boundary, the head runs from the beginning of the request to the next page boundary or the end of the request – whichever comes first.
- The middle, which runs from the first page boundary in the request to the last page boundary in the request. If these page boundaries are one and the same, the middle does not exist.
- The tail, which runs from the last page boundary in the request to the end of the request.

Any of these three parts may be missing in an I/O request, but at least one must be present.

2.1 Conversion of Random I/O to Paged I/O

During reads and writes the hash table will be searched to check if a page containing the head or tail of the I/O already exists in the page buffer.

If it does, then the request read or write will be applied to this page, with the page being marked dirty in the event of a write.

If the page containing the head or tail is not resident, it is loaded into the page buffer, and processing proceeds as above. Note that this may trigger a flush and eviction.

The middle section of a random I/O request is already paged. However, the management of this section of the request depends on whether any pages that appear in the middle are also resident in the page buffer.

For read requests, the read of any page resident in the page buffer must be satisfied from the version of the page in the page buffer. All other pages are read from file via the underlying VFD stack.

Middle write requests overwrite their constituent pages with new data, thus middle pages in the write request are written straight to the file. However, if the page buffer contains any pages that appear in the write request those pages must be flagged as invalid, since they contain old data. Invalid pages (and their page headers) are removed from the hash table and, regardless of the replacement policy used, selected for eviction prior to any valid pages.

3 VFD Page Buffer Design

While the above discussion of the Page Buffer should provide a good conceptual overview, this section goes into a detailed deep dive of the structures.

3.1 New Data Structures

3.1.1 H5FD_pb_pageheader_t

The page header structure is used by the page buffer to store the contents of the page from the file in the page buffer. Additionally, the page header structure contains necessary information about the page, such as the page's address and the hash code indicating the bucket in which the page is stored.

The definition of this structure is given below:

```
/******************************************************************************
 *
 * Structure:   H5FD_pb_pageheader_t
 *
 * Description:
 *
 * The pageheader structure is used to store the metadata for a page that is
 * stored in the page buffer, and contains a pointer to the actual contents of
 * the page. The hash table and replacement policy use this structure as nodes
 * keep track of the buffers in use and the order of pages to be evicted.
 *
```

```
 * Fields:
 *
 * magic (int32_t):
 *      Magic number to identify this struct. Must be H5FD_PB_PAGE_HEADER_MAGIC
 *
 * hash_code (uint32_t):
 *      Key used to determine which bucket in the hash table this pageheader
 *      gets stored in. The hash code is calculated by taking the page addr
 *      and cutting off the bottom bits, then right shifting the remaining bits
 *      to get the page number (this is computationally easier than division).
 *      Then the page number is modded by the number of buckets in the hash
 *      table, which results in the bucket number.
 *
 * ht_next_ptr (H5FD_pb_pageheader_t *):
 *      Pointer to the next pageheader in the hash table bucket, or NULL if
 *      this pageheader is the tail.
 *
 * ht_prev_ptr (H5FD_pb_pageheader_t *):
 *      Pointer to the previous pageheader in the hash table bucket, or NULL
 *      if this pageheader is the head.
 *
 * rp_next_ptr (H5FD_pb_pageheader_t *):
 *      Pointer to the next pageheader in the replacement policy list, or NULL
 *      if this pageheader is the tail (i.e. this pageheader is the next one
 *      in the list to be evicted).
 *
 * rp_prev_ptr (H5FD_pb_pageheader_t *):
 *      Pointer to the previous pageheader in the replacement policy list, or
 *      NULL if this pageheader is the head (i.e. this pageheader is the last
 *      one to be evicted next).
 *
 * flags (int32_t):
 *      Integer field used to store various flags that indicate the state of
 *      the pageheader. The flags are stored as bits in the integer field.
 *      The flags are as follows:
 *          - 0b00000001: dirty     (modified since last write)
 *          - 0b00000010: busy      (currently being read/written)
 *          - 0b00000100: read      (queued up to be read)
 *          - 0b00001000: write     (queued up to be written)
 *          - 0b00010000: invalid   (contains old data, page must be discarded)
 *
 * page_addr (haddr_t):
 *      Integer value indicating the addr of the page from the beginning of
 *      the file in bytes. This is used to determine the location of the page,
 *      and to calculate the hash key.
 *
 * type (H5FD_mem_t):
 *      Type of memory in the page.  This is the type associated with the
 *      I/O request that occasioned the load of the page into the page
 *      buffer.
 *
 * page (unsigned char *):
 *      buffer containing the actual data of the page. This is the data that
 *      is read from the file and stored in the page buffer.
 *
 ****************************************************************************
 */
typedef struct H5FD_pb_pageheader_t {
```

```
    int32_t             magic;
    uint32_t            hash_code;
    struct H5FD_pb_pageheader_t * ht_next_ptr;
    struct H5FD_pb_pageheader_t * ht_prev_ptr;
    struct H5FD_pb_pageheader_t * rp_next_ptr;
    struct H5FD_pb_pageheader_t * rp_prev_ptr;
    int32_t             flags;
    haddr_t             page_addr;
    H5FD_mem_t          type;
    unsigned char       page[];
} H5FD_pb_pageheader_t;
```

3.1.2 H5FD_pb_t

The primary structure of the Page Buffer VFD is H5FD_pb_t. This structure is comprised of two other structures, a hash table and a double linked list. The page buffer structure also stores information needed by VFDs above or below it in the VFD stack.

Since I/O below the Page Buffer VFD must be paged, it follows that below the Page Buffer VFD, the End of Allocation (EOA) and End of File (EOF) must be on page boundaries. In contrast, I/O above the page buffer is random, so here the EOA and EOF need not fall on page boundaries. As discussed in the following header comment for H5FD_pb_t, the eoa_up and eoa_down fields are used to manage this discrepancy.

As mentioned in the introduction, the current implementations of the allocation and free VFD callbacks cause space allocation failures in the upper library. This must be dealt with in a future iteration, as some VFDs will require it (split and multi VFDs). However, since targeting just the Sec2 VFD is sufficient for the Phase I prototype we are bypassing the issue for now.

The definition of H5FD_pb_t is given below, minus the variables used for statistical tracking. They were removed for brevity.

```
/****************************************************************************
 *
 * Structure:   H5FD_pb_t
 *
 * Description:
 *
 * Root structure used to store all information required to manage the page
 * buffer. An instance of this structure is created when the file is "opened"
 * and is discarded when the file is "closed".
 *
 * Fields:
 *
 * pub: An instance of H5FD_t which contains fields common to all VFDs.
 *      It must be the first item in this structure, since at higher levels,
 *      this structure will be treated as an instance of H5FD_t.
 *
 * magic (int32_t):
 *      Magic number to identify this struct. Must be H5FD_PB_MAGIC.
 *
```

```
 * fa:  An instance of H5FD_pb_vfd_config_t containing all configuration data
 *      needed to setup and run the page buffer.  This data is contained in
 *      an instance of H5FD_pb_vfd_config_t for convenience in the get and
 *      set FAPL calls.
 *
 * file: Pointer to the instance of H5FD_t used to manage the underlying
 *  VFD.  Note that this VFD may or may not be terminal (i.e. perform
 *  actual I/O on a file).
 *
 * Hash Table Description:
 *      A structure that contains an array of doubly linked lists used to store and
 *      index instances of H5FD_pb_pageheader_t associated with active pages in
 *      the buffer.
 *
 *      Random I/O requests come in and the page buffer turns them into paged I/O
 *      requests.  When there is a partial page in the request, the full page that
 *      contains the partial page must be loaded and those full pages are kept
 *      here in the hash table. The number of buckets must be a power of 2.
 *
 *      NOTE: The number of buckets in the hash table is currently fixed, but
 *      will be made configurable in future versions.
 *
 * Hast Table Fields:
 *      index:
 *          An integer value used to determine which bucket in the hash table
 *          an instance of H5FD_pb_pagedheader_t should be stored in based on
 *          its hash code.
 *
 *      num_pages_in_bucket:
 *          An integer value used to keep track of the number of pages that are
 *          stored in the bucket. This is a statistic used for debugging and
 *          performance analysis.
 *
 *      ht_head_ptr:
 *          Pointer to the head of the doubly linked list of pageheaders in
 *          the bucket. The head is where pageheaders are inserted into the
 *          bucket.
 *
 * Replacement Policy Description:
 *      A doubly linked list data structure used to store the pageheaders and
 *      determine which pageheader will evict its page to store a new page
 *      when a page must be read into the buffer. It will support multiple
 *      replacement policies, such as LRU, FIFO, etc.
 *      LRU is the default replacement policy.
 *      NOTE: Currenlty LRU is the only replacement policy implemented.
 *
 * Replacement Policy Fields:
 *      rp_policy:
 *          An integer value used to determine which replacement policy will be
 *          used. 0 is for LRU, 1 is for FIFO, etc.
 *
 *      rp_head_ptr:
 *          Pointer to the head of the doubly linked list of pageheaders in
 *          the replacement policy list. The head is the location that page
 *          headers are added to the list when created. Depending on the
 *          selected replacement policy, pageheaders may be moved to the head
 *          of the list when accessed again (such as Least Recently Used (LRU)
 *          policy).
```

```
 *
 *      rp_tail_ptr:
 *          Pointer to the tail of the doubly linked list of pageheaders in
 *          the replacement policy list. The tail is the location that page-
 *          headers are selected to have their pages evicted from the page
 *          buffer, when a new page must be read into the page buffer.
 *
 * EOA management:
 *
 * The page buffer VFD introduces an issue with respect to EOA management.
 *
 * Specifically, the page buffer converts random I/O to paged I/O.  As
 * a result, when it receives a set EOA directive, it must extend the
 * supplied EOA to the next page boundary lest the write of data in the
 * final page in the file fail.
 *
 * Similarly, when the current EOA is requested, the page buffer must
 * return the most recent EOA set from above, not the EOA returned by
 * the underlying VFD.
 *
 * Righly or wrongly, we make no attempt to adjust the reported EOF.
 * This may result in waste space in files.  If this becomes excessive,
 * we will have to re-visit this issue.
 *
 * eoa_up: The current EOA as seen by the VFD directly above the encryption
 *      VFD in the VFD stack.  This value is set to zero at file open time,
 *      and retains that value until the first set eoa call.
 *
 * eoa_down: The current EOA as seen by the VFD directly below the encryption
 *      VFD in the VFD stack. This field is set to eoa_up extended to the
 *      next page boundary.  As with eoa_up, this alue is set to zero at
 *      file open time, and retains that value until the first set eoa call.
 *
 ******************************************************************************
 */

typedef struct H5FD_pb_t {

    H5FD_t               pub;

    int32_t              magic;
    H5FD_pb_vfd_config_t fa;
    H5FD_t              *file;

    /* hash table fields */
    struct {
        int32_t               index;
        int32_t               num_pages_in_bucket;
        H5FD_pb_pageheader_t *ht_head_ptr;
    } ht_bucket[H5FD_PB_DEFAULT_NUM_HASH_BUCKETS];

    /* replacement policy fields */
    int32_t                   rp_policy;
    H5FD_pb_pageheader_t     *rp_head_ptr;
    H5FD_pb_pageheader_t     *rp_tail_ptr;
    int64_t                   rp_pageheader_count;

    /* eoa management fields */
```

```
    haddr_t                 eoa_up;
    haddr_t                 eoa_down;

} H5FD_pb_t;
```

4 Testing

Due to time constraints, only a subset of unit tests has been implemented so far. More tests will be implemented in the next version and are discussed in section 4.2 Needed Unit Tests.

4.1 Current Unit Tests

Unit tests are intended to verify correct behavior of the Page Buffer VFD. This section will be expanded in the future as more unit tests are written.

- Test read-only access, including when the file does not exist.
- Test file creation, utilizing different child FAPLs (default vs. specified), logfile, and Write Channel error ignoring behavior.
- Test write and read of each piece, head, middle, and tail, when the page does not exist in the page buffer and when the page does exist in the page buffer.
- Tests write and read of all possible combinations of head, middle, and tail:
    - Head and tail
    - Head and middle
    - Middle and tail
    - Head, middle, and tail.
- Test the eviction of pages from the replacement policy's doubly linked list, making sure the correct instance of H5FD_pb_pageheader_t is evicted, for both LRU and FIFO replacement policies.
- Test invalidating a page in the page buffer ensuring that the invalidated page is next to be evicted and that the invalidated page will not be found when searching the page buffer.
- Test specific middle cases when the middle contains multiple pages and only some of them are in the page buffer.
    - Only the first page is in the page buffer
    - Only the last page is in the page buffer
    - The first and last page are not in the page buffer, but consecutive pages between them are.
    - Alternating pages of the middle are in the page buffer.

4.2 Needed Unit Tests

These unit tests are tests that will be added in the future but were left out due to time constraints.

- Tests verifying that an I/O request will be rejected if the address, size, or buffer of the request are invalid (i.e. address outside of the allocated file memory space).
- Tests that verify very small heads and tails (i.e. 1, 2, or 3 bytes of data) are handled correctly.
- Tests with different page sizes. Currently all tests are done with the default page size of 4096 bytes.
- Test the FIFO replacement policy to verify that touching a page is a NO-OP as expected.

4.3 Needed Integration Tests

At present, no automated integration tests exist. The exact integration tests needed will be determined as part of future work. However, the following seem obvious:

- Update “make check vfd” to include the page buffer VFD.
- Add the page buffer VFD to the tools tests.

Acknowledgement

This work is supported by the U.S. Department of Energy, Office of Science under award number DE-SC0024823 for Phase I SBIR project “Protecting the confidentiality and integrity of data stored in HDF5"

Revision History

September 30, 2024:	Version 1 circulated for comment within Lifeboat, LLC.
October 17, 2024:	Version 2 added copyright symbol to footnotes.