Performance Report for Free-space Manager

Vailin Choi
The HDF Group
9/15/2008

1. Purpose

The free-space manager is used to track unused space within a heap or an HDF5 file. The trunk and free-space feature branch differ in the way free space for the file memory is handled. This document reports the performance of free-space manager for the two branches.

2. Overview of current Implementation for free-space

- *hdf5 trunk*

File memory is allocated and freed via the virtual file layer, which tracks free-space sections using a free-list. The free-list is an unsorted singly-linked list of address/size pairs. File memory allocation request can be of different types such as super block, b-tree, raw data, global heap and local heap.

When file memory space is requested, the virtual file layer attempts to find space from the corresponding free-list. It tries to find an exact match if possible; otherwise, the best match which is the smallest size that meets the request. If the free-list cannot satisfy the request, the virtual file layer tries to allocate space from the meta/small data aggregators or via individual driver's file allocation.

When unused file memory space is freed, the virtual file layer releases the space to the corresponding free-list. It will either create a new node for the freed block or try to merge the freed block with an existing node in the free-list. The new or adjusted node will be inserted at the beginning of the free-list. The virtual file layer will also handle merging of the freed block with the meta/small data aggregator as well as file shrinking if the freed block is at the end of file.

Since the trunk uses the unsorted linked-list for tracking free blocks, it takes *n* operations to search for a free block and also *n* operations for adding a free block to the linked-list.

- *Free-space feature branch*

File memory is a client of the free-space manager, which uses skip-list to track free spaces. There is free-space manager for each memory allocation request such as super block, b-tree, raw data, global heap and local heap.

All free-space sections are tracked by an array of bins, which is hashed on ranges of section sizes. Each bin points to a skip-list, referred to as *bin-list,* which tracks a range of section sizes for that bin. Each node of a section size in *bin-list* points to a skip-list, referred to as *sect-list*, which tracks sections with ascending addresses for

that size. See Figure 1 *free-space data structures*. There is also a separate skip-list, referred to as *merge-list*, which tracks sections of free-space in address order that might possibly be merged with neighboring sections.

When file memory space is requested, the file memory layer tries to search for large enough space from the free-space manager. The size of the free-space section is hashed to a specific *bin-list*. If there is a large enough size in *bin-list* to fulfill the request, the free-space manager removes the first section with the lowest address from the corresponding *sect-list* as well as the corresponding section from *merge-list*. Otherwise, the search continues to the next bin. Also, if a section is found but its size is greater than the request, the section with the unused space is re-added to free-space pool. However, if the free-space manager cannot satisfy the request in the end, the file memory layer tries to allocate space from the meta/small data aggregators or via individual driver's file allocation.

When unused file memory space is freed, the free-space manager will try merging the returned section with its left and right neighboring sections in *merge-list*. The merging operation continues in this fashion with the new merged section until no more merging is possible. It then handles file shrinking if the returned section is at the end of file and/or absorption of the returned section with the meta/small data aggregator as needed. The shrinking operation is continued with the last section in *merge-list* until no more shrinking is possible. In the end, the free-space manager inserts the returned section, which may possibly be merged, to the appropriate *sect-list*, and then the *merge-list.* Or, no insertion is needed if the section is shrunk away.

Table 1 and Table 2 summarize the steps and the respective operations involved in allocating space from free-space. The following observations are noted:

- **Total** for Table 1 might be less because step #3 is not needed if section size exactly fulfills the request.
- **Total** for Table 2 might be more because step #2 might be repeated to the next bin for up to a maximum of $\log_2$ (*maximum section size*) times if the current bin's *bin-list* does not meet the request.

Table 1

Allocating space from free-space	# of operations
1. Retrieve large enough section from *bin-list* and *sect-list* to fulfill request (see Table 2)	$2 + \log_2 n$
2. Remove the section from *merge-list*	$\log_2 n$
3. Possibly re-add a section of unused space in the section to *merge-list, bin-list & sect-list*	$1 + 4 \log_2 n$
Total	$\mathbf{3 + 6 \log_2 n}$

Table 2

Retrieving a section from *bin-list* and *sect-list*	# of operations
1. Get to the desired hash bin	1
2. Search *bin-list* for large enough size to fulfill request	$\log_2 n$
3. Remove the first section from *sect-list*	1
Total	$\mathbf{2+ \log_2 n}$

Table 3 through Table 6 summarizes the steps and the respective operations involved in adding (freeing) space to free-space. The following observations are noted:

- **Total** for Table 3 might be less because steps #2 and #3 are not needed if the section was shrunk.
- **Total** for Table 4 might be less because step #3 is not needed if the requested section size node already existed.
- **Total** for Table 5 might be less because steps #2, 4 or 6 are not needed if the section is not merged or shrunk. On the other hand, **total** might be more because merging/shrinking might be repeated arbitrary number of times as described above.

Table 3

Adding (freeing) space to free-space	# of operations
1. Merge/shrink the section as needed (see Table 5)	$5 + 7 \log_2 n$
2. Insert the section to *bin-list* and *sect-list* if not shrunk (see Table 4)	$1 + 3 \log_2 n$
3. Insert the section to *merge-list* if not shrunk	$\log_2 n$
Total	$\mathbf{6 + 11 \log_2 n}$

Table 4

Inserting a section to *bin-list* & *sect-list*	# of operations
1. Get to the desired hash bin	1
2. Search *bin-list* for section size	$\log_2 n$
3. Possibly insert new section size to *bin-list*	$\log_2 n$
4. Insert the section to *sect-list*	$\log_2 n$
Total	$\mathbf{1+ 3 \log_2 n}$

Table 5

Merging/shrinking a section	# of operations
1. Find a *less than* section to be merged from *merge-list*	$\log_2 n$
2. Possibly remove *less than* section being merged from *bin-list* & *sect-list* (see Table 6)	$1+2\log_2 n$
3. Find a *greater than* section to be merged from *merge-list*	1
4. Possibly remove *greater than* section being merged from *bin-list* & *sect-list* (see Table 6)	$1+ 2\log_2 n$
5. Find *last* section to be shrunk from *merge-list*	1
6. Possibly remove *last* section to be shrunk from *bin-list* & *sect-list* (see Table 6)	$1+ 2\log_2 n$
Total	$\mathbf{5 + 7 \log_2 n}$

Table 6

Removing a section from *bin-list* and *sect-list*	# of operations
1. Get to the desired hash bin	1
2. Search *bin-list* for section size	$\log_2 n$
3. Remove the section from *sect-list*	$\log_2 n$
Total	$\mathbf{1+ 2 \log_2 n}$

3. Timing Methodology

The library's internal timer routines, *H5_timer_begin()* and *H5_timer_end()*, are used to measure time. The timer, which calls *getrusage(),* reports three types of times: user time, system time and elapsed time. User time is "the amount of time spent executing in user mode", while system time is "the amount of time the system (or the processor) spends on behalf of the current process". Elapsed time is the wall-clock time, which is "the amount of time that passes on your wall clock while your process runs." Since we are measuring the performance of the free-space manager in the library, user and system times are added together to get cpu time for this benchmark. The timer is turned on at the beginning of each test when creating the HDF5 file and turns off when the file is closed at the end of the test.

4. Testing

Tests are written with the purpose of exercising the usage of the free-space manager. When creating groups/datasets/attributes, space will be allocated from available space tracked by the free-space manager. When deleting groups/datasets/attributes, unused space will be added to the free-space pool via the free-space manager, which also handles merging/shrinking of free-space sections. Also, all the datasets are created with H5D_ALLOC_TIME_EARLY so that storage is allocated on creation of the datasets and no I/O is involved in reading/writing the datasets.

- Test 1: Creating/deleting groups
  The following test is repeated with sets of 500, 1,000, 5,000, 10,000 and 50,000 groups.

  Create first set of groups;
  Delete odd-numbered groups from the first set
  Create second set of groups
  Delete all groups from the second set

- Test 2: Creating/deleting datasets
  The following test is repeated with sets of 500, 1,000, 5,000, 10,000 and 50,000 datasets.

  Create one set of big datasets
  Delete odd-numbered big datasets
  Create one huge dataset
  Delete the huge dataset
  Create one set of medium datasets
  Delete odd-numbered medium datasets
  Create one set of small datasets

- Test 3: Creating/deleting groups with datasets
  The following test is repeated for sets of 500 and 1,000 groups with the corresponding number of datasets. This test is not done for sets of 5,000, 10,000 and 50,000 groups/datasets because it takes too long.

Create first set of groups with medium datasets
Delete odd-numbered groups and their datasets in the first set
Create second set of groups with small datasets

- Test 4: Creating/deleting attributes
  (Note that this test cannot be done at this time due to abort problem #2 below)

  Create first set of attributes
  Delete odd-numbered attributes from the first set
  Create second set of attributes
  Delete all attributes from the second set

3. Existing problems

The following two problems are encountered when running the performance tests and are reported to be fixed:

1. *abort* for overlapping of raw data with meta data accumulator
2. *abort* for exceeding maximum object header message size

5. Results

The following tables list the result for test #1, 2 and 3.

Test 1

	free-space branch (A)	hdf5 trunk (B)	A versus B B/A
objects	*cpu time (seconds)*	*cpu time (seconds)*	*# times faster*
500	0.107983	0.118982	1.101859
1,000	0.230965	0.232964	1.008655
5,000	1.824721	2.511614	1.376437
10,000	5.655142	10.02247	1.772276
50,000	161.09337	643.3811	3.99384

Test 2

	free-space branch (A)	hdf5 trunk (B)	A versus B B/A
objects	*cpu time (seconds)*	*cpu time (seconds)*	*# times faster*
500	0.189971	0.301954	1.589474
1,000	0.39194	0.651901	1.663267
5,000	2.251661	9.32158	4.139868
10,000	5.684133	44.17526	7.77168
50,000	102.67842	1,431.526	13.94184

Test 3

objects	free-space branch (A) *cpu time (seconds)*	hdf5 trunk (B) *cpu time (seconds)*	A versus B B/A *# times faster*
500	68.17364	786.497	11.53667
1,000	288.9866	1,0998.77	38.0598

6. Conclusion

As can be seen from the last column of the above tables, the performance of the free-space feature branch is significantly better than the trunk as the number of objects increases. The result is consistent with the implementation of the free-space manager for the two branches. For n allocation or freeing of free-space, the hdf5 trunk is in the order n^2 but the feature branch is $n\log_2 n$.

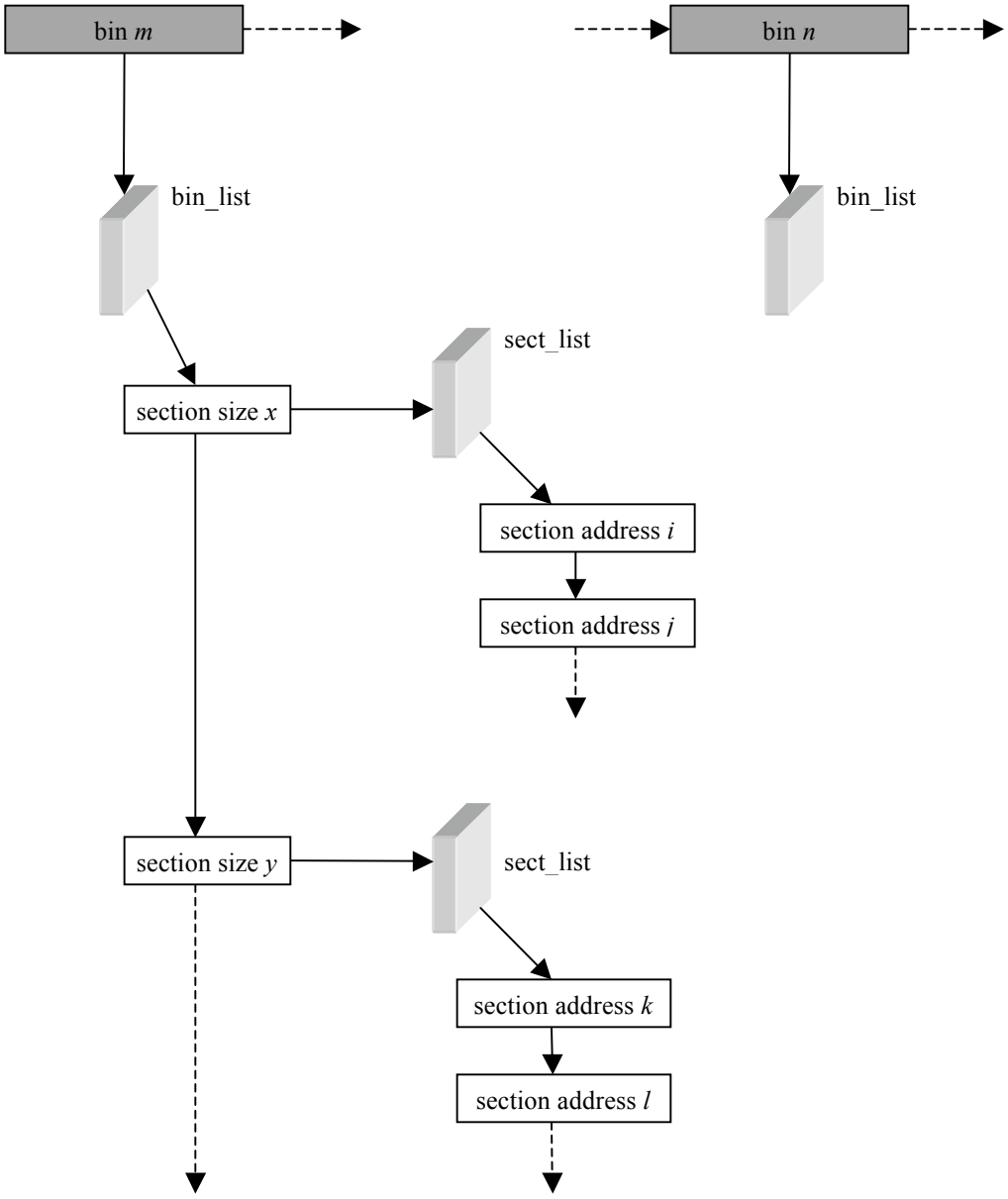

Figure 1 ***free-space data structures***