James Hong

Kuang Han

Time spent: 2 hours

The DeFiler file system was implemented with three layers of abstraction: the file system layer (DFS), the caching and memory buffering layer (DBufferCache and DBuffer), and the block device layer (VirtualDisk).

The file system layer manages the data in a high-level abstraction. This layer was implemented as a single class called DFS, and contains methods that allow the creation/deletion/listing of files, as well as read and write operations.

The caching and memory buffering layer provides an interface between the raw block device and the file system abstraction. The buffers should reside completely in main memory, allowing for performance improvements since the number of necessary I/O operations will be reduced. In order to keep a constant size buffer with LRU replacement policy, two data structures were considered: an array with a timestamp field for each entry and a linked list sorted by recent usage (most recent on top of the list). The array structure could provide O(1) access to each buffer and O(n) time for replacement of a single element (due to necessity to scan entire array for oldest timestamp). The linked list could provide O(n) access to buffers and O(1) time for replacement. It was decided that the linked list would be the best data structure to use in this case because the potential worst case scenario for replacing all elements in the array (such as when using the buffer for data streaming) is O(nˆ2), which could cause bottlenecks in performance.

An LRU replacement policy was decided as the best option in this case because it seems to have high performance for the majority of use cases. It only performs badly when performing large sequential accesses but then, those types of accesses are hard to cache with any replacement policy. Only prefetching would improve it.

The raw block device layer is implemented in the class VirtualDisk. It interfaces directly with a file. There is a queue for I/O requests (between the DBuffer and VirtualDisk layers) and a worker thread that takes requests off the queue and executes them sequentially. Therefore, no synchronization procedures are needed at the VirtualDisk level.

The thread model in this file system is that every concurrent operation on a distinct DFile uses a distinct thread. A thread that originates somewhere in the application and wants to perform some I/O on a DFile will call a method at the top layer of the file system (the DFS), which will in turn call the middle layer of buffers and cache. Depending on the cache status and eviction policy, the I/O operation might go to disk and call the lower layer (VirtualDisk). The methods that are actually called depend on the operation performed, but the flow is intuitive: a read() on the top layer will invariably call read() on the middle layer (if the buffer is present and valid), otherwise, the startRequest() asynchronous method in the VirtualDisk gets called and the data contents get loaded to a DBuffer. Notet tha the interface between the DBuffers and the VirtualDisk is asynchronous: the method returns shortly after being called and the completion of the task is signaled by callbacks to the middle layer. Some methods are special for bookkeeping and should be called in some circumstances. For instance, sync() in the DBufferCache should be called before exiting an application in order to flush the buffers to disk.

On the DFS level, we create an array of DFile metadata to make a memory copy of the disk inode regions, so that each update to inode can be cached like a normal data block to avoid the overheads in the disk IO. Aside from the DFile metadata, a freemap of disk block is also used by DFS to track the block usage. When DFS answers write requests, this freemap is used to identify the disk block usable by DFiles. As for the synchronization, we used a ReadWriteLock inside each DFile metadata to make sure that threads that write to each DFile are exclusive while reads are shared.

In terms of cache level synchronization, DBufferCache will have a lock on its DBuffer list (or set depending on the implementation) so that access to the DBuffer queue will be atomic.. Also, each DBuffer has its own status method isBusy() to identify when the buffer is processing a I/O or when it is being held. Only one thread can have access to the buffer when it is busy. This is to prevent the case that eviction and access calls happen on the same buffer.

The tests are all written at the DFS level. They test common functionality, such as creating and deleting files, and reading and writing data from a file. Some tests only perform those operations on a single file, and some stress test the file system by creating as many files as possible and writing data to them, as well as randomly interleaving operations such as creating, deleting and content-checking files.