The 'getAttr' method is a frequently used function in FUSE. It retrieves information about a file including access times, permissions, and ID parameters. Any method that modifies a file is likely to call 'getAttr' first to retrieve the necessary information.

In a client-server implementation of FUSE, where the methods reside on a server and are called by a client, a large number of calls made are 'getAttr' calls on files. However, calling 'getAttr' on the same file repeatedly can result in multiple calls to the server, which may return similar parameters to the client. By implementing caching for file attributes within our FUSE implementation, we can avoid making unnecessary calls to the server and optimize 'getAttr' to only be called when necessary.

My caching implementation involves saving file attributes in a cache every time 'getAttr' is called. If 'getAttr' is called on a new file or a file that has not been accessed recently, the file's attributes are added to the cache in case they are needed later. If 'getAttr' is called on the same file again, we check the cache for its existence, and, if found, return the attributes from the cache, saving a call to the server.

For each entry in the cache, we save the file's path along with its mode, user ID, and size. When 'getAttr' is called, these parameters are returned to the caller if the file is found in the cache. If 'getAttr' is called on the server, the new file's attributes are also added to the cache. The parameters of every cache entry are briefly described below.

path - the full path of the file
mode - the mode of the file
uid - the user id of the file
size - size of the file in bytes

Cache Implementation:

Our cache is implemented as a circular queue, where the most recently added entry becomes the new start of the queue. When adding a new file to the cache, the 'addToCache' method is called with the file's path and attributes as parameters. If the cache has reached its maximum capacity, 'addToCache' will handle this by removing the oldest entry in the cache to make room for the new file.

If a file needs to be removed from the cache, the 'removeFromCache' method is called with the file's pathname as a parameter. In addition, we have provided methods 'updateModeOfFileOnCache' and 'updateSizeOfFileOnCache' to update the file attributes if necessary.To ensure the consistency of the cache, these update methods may need to be called in response to certain calls made through FUSE.

The full set of cache management methods is listed on the following page.

Caching Methods:

```
/**
* Adds a file into the cache.
*
* @param head - Reference to the head of the cache
* @param path - the path of the file to add
* @param mode - the mode of the file
* @param uid  - the user id of the file
* @param size - the size of the file
*/
void addToCache(struct cacheEntry** head, char *path, mode_t mode, uid_t uid, off_t size);

/**
* Removes a file from the cache
*
* @param head - Reference to the head of the cache
* @param path - the path of the file to remove
*/
void removeFromCache(struct cacheEntry** head, char *path);

/**
* Removes the oldest file in the cache. Method Utilizes
* First in First out
*
* @param head - Reference to the head of the cache
*/
void removeOldestFile(struct cacheEntry **head);
```

```
/**
* Returns the specific attributes of a file identified by a path
*
* @param head - Reference to the head of the cache
* @param path - the path of the file
* @param mode - the mode of the file
* @param uid  - the user id of the file
* @param size - the size of the file
*/
void getFileAttrFromCache(struct cacheEntry** head, const char *path, mode_t *mode, uid_t
*uid, off_t *size);

/**
* Identifies if a specific file is on the cache.
*
* @param head - Reference to the head of the cache
* @param path - the path of the file to find
*/
int pathOnCache(struct cacheEntry *head, const char *path);


/**
* Prints the cache starting from the head
*
* @param head - Reference to the head of the cache
*/
void printCache(struct cacheEntry *head);
```

```
/**
* Counts the number of parameters on the cache
*
* @param head - Reference to the head of the cache
*
* @return the size of the cache
*/
int countCache(struct cacheEntry *head);


/**
* Updates a file size
*
* @param head - Reference to the head of the cache
* @param path - the path of the file
* @param s    - the new size of the file
*/
void updateSizeOfFileOnCache(struct cacheEntry *head, char *path, off_t s);


/**
* Returns the specific attributes of a file identified by a path
*
* @param head - Reference to the head of the cache
* @param path - the path of the file
* @param mode - the new mode of the file
*/
void updateModeOfFileOnCache(struct cacheEntry *head, char *path, mode_t m);
```

Testing:

To test the effectiveness of our cache, we want to see the behavior of the FUSE client server implementation when we access the same file more than once. Here is a sample bash script that fuse will call on itself titled stresshoofs.sh

```
touch d
echo "this" > d
echo "is" > d
echo "my" > d
echo "test" > d

chmod 777 d
chmod 646 d
chmod 777 d
```

In this sample script, we create a file using the 'touch' command, and then modify the file's attributes several times. Without caching, the attributes of the file would be repeatedly accessed from the server. However, our caching implementation stores the file attributes locally, allowing for subsequent access without the need for additional server calls.

To verify the effectiveness of our caching implementation, we can utilize the XMLRPC environment variable XMLRPC_TRACE_XML to track the number of times the 'getAttr' method is called on the server. This allows us to determine the number of server calls that are being made and evaluate the efficiency of our caching approach

Result:

The sample script was run with caching both enabled and disabled. When caching was disabled, the script resulted in 18 calls to the function 'getAttr' on the server. With caching enabled, there were only two calls to 'getAttr' on the server - one initial call when the file was not found, and a second call after the file was created. The results of these tests are documented in the files 'cachingon.txt' and 'cachingoff.txt' within the submission.

Caching significantly reduced the number of calls made to the server, by approximately 88% in this instance. This improvement is significant, particularly when considering the potential for multiple frequently accessed files. By caching file attributes, we can optimize the process of retrieving these attributes and only retrieve them when necessary