Cache.java Specification

Cache.java simulates a cache that utilizes the enhanced second chance algorithm. Within the class Cache there is a custom CacheEntry data structure that is used to represent an entry in a page table. The Cache class itself then maintains an array of CacheEntry objects that constitute said page table.

The implementation of Cache's main methods -- read, write, sync, and flush -- is fairly straightforward. The read method first looks for a frame that matches a given block ID. If a match exists in the cache, it proceeds to read the cache block into the buffer. If it can't find the frame in the cache, it looks for an empty frame to fill and if it finds one, then it loads the required data into it from memory. However if it cannot find an empty frame to fill, then the Cache finds a victim page to overwrite. And finally before overwriting the victim page, the Cache first checks to see if it has unsynchronized data in it, according to its dirty bit being set, and if so then it commits that data to memory.

The write method works in a similar way as the read method. It first checks the cache for a particular frame to write to. However, a notable difference between write and read is that the write method will set the dirty bit to true after it changes the cache's contents.

The sync and flush methods are both similar in function. Both of these methods commit the entire cache to memory. However, the flush method differs from sync in that it also "clears" the cache by virtue of invalidating all of its frames and setting all reference bits to false.

Below are the results of running Test4 against Cache.java. Test4 ran a total of 8 tests. Four tests ran with the cache enabled, and four ran with the cache disabled. To understand these results, it is important to only compare the same types of tests together across enabled cache vs disabled cache. For example, it makes sense to compare Random accesses with enabled cache vs Random accesses with disabled cache, but it does not make sense to compare Random accesses vs Localized accesses. This is because the implementations of each type of cache access are notably different from each other and comparatively irrelevant.

For both the Random accesses and the Adversary accesses tests, there is little difference observed in the performance between having the cache enabled and having it disabled. This observation makes sense considering that the cache is not deliberately being made use of in the Random accesses test and that the cache is deliberately not being made use of in the Adversary accesses test.

However, for Localized accesses and Mixed accesses we can see an extremely clear difference in performance. Particularly with Localized accesses, having cache enabled resulted in extremely fast -- *stupid fast* -- read and write access. This is no surprise because the cache is being intentionally optimally utilized. Similarly, though not to the same extreme as Localized

accesses, Mixed accesses yielded notably faster results with cache enabled, and again, this makes sense because it intentionally utilizes the cache some of the time, but not ubiquitously.

Test4 Performance Results

Cache enabled	Average write	Average read
Random accesses	11752	12508
Localized accesses	0	1
Mixed accesses	3771	3710
Adversary accesses	11296	12861

Cache disabled	Average write	Average read
Random accesses	11897	11979
Localized accesses	8137	8196
Mixed accesses	8382	8281
Adversary accesses	11728	11830