Anthony Waddell

CSS 430

HW4 Report

**Algorithm Description**

This lab uses a cache and corresponding cache blocks to store frequently accessed data in memory. A cache block is an object that contains three components, the block frame number, a reference bit, and a dirty bit. Cache blocks get allocated by the cache on memory and contain data equal to a predetermined block size. The cache class contains function to perform block read and block writes that both sequentially scan the cache looking for a cache block with a particular blockId. If found, that cache block’s data is read; if not, it finds another block in the cache by looking for the next block whose blockId is -1 indicating that it is a free block. Once a free block is found, data is read from the disk to this block. If no free blocks exist within the cache, the read function uses an enhanced second chance algorithm to look for blocks that have not been accessed recently. If the next victim block that’s chosen has the dirty bit flipped, the data it contains is written to disk first before new data can be written to it.

The write function behaves similarly to the read function, scanning the cache for the block with the given blockId. If a cache block with that blockId exists in the cache, it writes to that block and sets its dirty bit to true. If a block with that blockId does not exist within the cache, it will look for an empty block to write to using by looking for the next block with a -1 blockId. If there isn’t an empty block, like the read algorithm it will use the enhanced second chance algorithm to locate the next victim block to write to, looking for one whose reference bit is set to false. Like the read function, if the victim block has it’s dirty bit set to true, before writing data to this block it must copy the current data back to the disk.

Aside from some utility functions, one of the other functions implemented in this program was the second chance algorithm. This algorithm continuously iterates over the cache table looking for the next “victim” block by finding the next block whose reference bit is false. While it iterates over the blocks, if it comes across blocks whose reference bit is set to true, indicating they have been recently used, it will set it to false so that next time it iterates over, if it has not been changed, it can be used again. If it finds a cache block whose reference bit is false, it returns the index of this cache block as the next “victim”.

**Performance Evaluation – with all commands ran consecutively**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test Case** | **With Cache enabled** | | **With Cache disabled** | |
| **Average read (ms)** | **Average write (ms)** | **Average read (ms)** | **Average write (ms)** |
| **Random** | **674** | **74** | **112** | **92** |
| **Local** | **1** | **1163** | **439** | **497** |
| **Mixed** | **9** | **528** | **57** | **65** |
| **Advers.** | **96** | **128** | **69** | **57** |

Looking at these performance results, I am not quite sure I executed my implementation properly. These numbers are a little higher than other runs but seem a little too high for some of the different test cases. I am most interested in the localized accesses with caching enabled. This performance metric had the widest variance by far and the average write was more than twice as long as any of the other average write test cases. I am not as surprised about the local accesses average read time since it kind of makes sense since that it’s just reading directly from the cache but I am not so sure why the write had an average of 1163 milliseconds.

With these numbers, it’s observed that caching reduces the read time for local and mixed but it is slightly higher with random and adversarial. Overall it appears that the average write times are longer with caching. Another interesting performance indicator I found was the average read time for random accesses with caching enabled. It seems to be significantly higher with caching enabled.

**Performance Evaluation – with each test case ran alone**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test Case** | **With Cache enabled** | | **With Cache disabled** | |
| **Average read (ms)** | **Average write (ms)** | **Average read (ms)** | **Average write (ms)** |
| **Random** | **84** | **71** | **69** | **43** |
| **Local** | **2** | **670** | **270** | **387** |
| **Mixed** | **20** | **48** | **23** | **34** |
| **Advers.** | **2** | **39** | **20** | **20** |

The results listed above are only running Test4 with one test case at a time, for both enabled and disabled caching. These results look a lot better than the previous where all test cases were ran concurrently. The number appear proportionally smaller than the previous test results. I was expecting adversarial to be much worse than it was but for some reason it was still just about faster than all other test cases whether or not caching was enabled.