**Program 4J Report**

**Specification/design explanation on Cache.java**

In figure 1, you can see the design for Cache.java. A small class that is in the Cache class is the Entry class. The Entry class is just an entry of data that has both reference and modified bits (dirty). The Cache itself have 3 global variables. blockSize holds the block size of each page, pages are the pages in the cache, and the victim is the chosen victim using enhanced second chance algorithm.

The enhanced second chance algorithm is literally the second chance algorithm but not only are we taking the reference bit into consideration, we are taking the modified bit into consideration. The next victim chosen will depend on the factors below, where 1 is most likely to be the victim and 4 is the least:

1. Reference = false, dirty = false
2. Reference = false, dirty = true
3. Reference = true, dirty = false
4. Reference = true, dirty = true

When we are picking the victim using the enhanced second chance algorithm, if the victim has a reference bit set to 1 or true, they get a second chance. So, we will go to the next victim if this is the case. So, we go through all the entries and find the first victim that best fits, either case 1 or 2 mentioned above since case 3 and 4 both get second chances. The worst-case scenario is after all the second chance, we still have case all the entries in case 2, which means we have to get the first victim that has case 2. This implementation can be seen in the nextVictim() function in our Cache.

All the implementation details of the functions and how they work are in the Cache.java file so I won’t be going into that too in depth. But the main functions of the Cache class are the read and write function. The read functions find the entry in memory and return true if found and false otherwise. The write function writes the content of the buffer given and puts it in the cache block given.

A screenshot of a computer program

Description automatically generated

Figure 1 – Cache UML Diagram

In figure 2, we have the output we get from enabling and disabling cache. Along with that we have the output for random accesses, localized accesses, mixed accesses, and adversary accesses.

A computer screen shot of a black and white screen

Description automatically generated

Figure 2 – Test4 random, localized, mixed, and adversary output.

**Performance analysis**

* Performance consideration on random accesses
  + When cache is enabled, the performance of the random accesses was the 2nd worst. This is probably because when it is randomly accessing blocks in memory, it is unlikely to get many cache hits as everything is chosen at random. This is why the performance of this is 2nd worse when cache is enabled.
  + When cache is disabled, the performance of the randomly accesses was the second best but far from being close to the best, which is mixed accesses. Randomly accessing memory can be a hit or miss because you are just randomly accessing the memory with no pattern.
* Performance consideration on localized accesses
  + When cache is enabled, the performance of the localized accesses was the best. This is because it accesses small selections of blocks in memory many times, which means those blocks are more likely to be stored in cache since the cache is made for this. This will cause more cache hits and therefore a very good performance overall (cache hit is when we find the data we want to access in the cache).
  + When cache is disabled, the performance of the localized accesses is worse than random accesses but not too far off from it. This is probably because without the cache, there is no buffer to save the data accesses the most therefore we still have to access the memory just in the same place.
* Performance consideration on mixed accesses
  + When cache is enabled, the performance of mixed access is the 2nd best. This is expected since mixed access utilizes 90% localized access and only 10% random access. This makes sense that mixed accesses’ performance is in between random accesses and localized accesses. Since localized access is very good with cache enabled and 90% of the mixed access is localized accesses, the performance should be very good also.
  + When cache is disabled, the performance of mixed accesses was surprisingly the best. This might be because random access makes us access different parts of the memory, but localized access allows us to access the memory near the same place. If we randomly choose a memory that is easy to access, and we continue to do so then we are more likely to get a better performance.
* Performance consideration on adversary accesses
  + When cache is enabled, the performance of adversary accesses is the worst. That is because adversary access purposefully access memory in a way that it doesn’t make good use of the cache at all. It does this by making sure we are accessing something we have not accessed in the last 10 accesses.
  + When cache is disabled, the performance of adversary accesses is the worse also. This is probably because it is not only random but also the accesses are chosen to be different from each other.