**Project 3 Report**

**Yi Han**

**Introduction**

This project is about different designs of Cache and the ways they work. Simulators for Direct Mapped, Set Associative and Fully Associative caches will be implemented and they will be used to simulate a series of write and read operations. Their hit rates, number of bytes memory transferred to cache and the inverse will be reported.

**Implementation of Simulator**

This simulator is implemented in C++. A class called Cache represent a single cache with pre-specified properties include block size, cache size, number of ways, placement type and write policy. This class contains some interfacing member functions. The following is a brief explanation of these functions

Cache(size\_t s, size\_t bs, PlacementType pt, WritePolicy wp);

Constructor for the Cache. S for cache size, bs for block size, etc.

bool readAddress(uint32\_t ad);

read a 32bit address. Return true if hit (return value not used. It was there for the sake of simplicity of implementing the function)

bool writeAddress(uint32\_t ad);

write to a 32 bit address.

const float reportHitRate();

report current hit rate. -1 if haven’t write/read.

string toString();

return a string representation in the required output format.

void clear();

This function is called at the end of the program to simulate the process of writing all dirty blocks back to the memory.

Other functions are just getters for getting the corresponding property the cache.

The following is a brief explanation of the private helper functions.

void cacheMiss(uint32\_t ad, bool WR);

This function is called upon cache misses. ad is the 32-bit address. WR stands for operation. True for write, false for read. This function will compute set index and tag. It will also decide with way to evict if necessary. It will then call evict to simulate fetching from memory.

void evict(size\_t way, size\_t setIndex, uint32\_t tag, bool WR);

This function is called by cacheMiss at the end. way stands for the way that will be evicted.

The class is constructed by the specified input. The number of sets is computed as cache size / block size / number of ways. In case of fully associative cache, the number of sets is one. In case of direct mapped cache, the number of ways is one. A 2D vector serves the purpose of storing all the blocks. The first index of this array is the set index and the second index is way. Block is a structure which has the following properties:

size\_t size;

uint32\_t tag;

bool valid;

size\_t useTime;

bool dirty;

size is the block size. useTime is the nth time of reading/writing memory when that particular block is used. It’s used to implement LRU. Dirty marks whether this block is overwritten but haven’t updated to the ram in case of write-through cache.

Here are more details about how I approached this simulator.

After the construction of the 2D cache array, everything inside the block will be initialized to its initial value. Dirty and valid will be initialized to false, tag and useTime will be initialized to zero. After the construction of all 128 possible caches, the program will start to parse the file and simulate the process of reading/writing to a memory location. During a memory read, the set index and tag are computed from the memory address. Then all ways in that particular set are checked, if any of them have the same tag value as the computed one and valid is set to true, a cache hit is reported. If none of the ways match these conditions, a cache miss will be reported. For a write operation, the procedure is similar, the difference is that in case of write-through policy, the memory will be updated by the cache. When a cache miss is reported, data from the memory that corresponds to the entire block will be fetched by the cache, within a set, the ways are sorted by their useTime, the block with the least useTime will be evicted. During the eviction, valid is set to true, tag is set to the computed tag. In case of write back, if the block is dirty, cache will write the data in that block back to memory before fetching the data. If this is a write miss, dirty will be set, if this is a read miss, dirty will be set to false. Upon finishing all the read/write operations, clear will be called. Details about clear can be found above where the functions are introduced.

**Testing**

This simulator can be considered as working because it passes all the test cases given. In order to show a change in hit rate with respect to increase in associativity, and increase in block size, two separate test cases are generated.

The first test case is created in favor of associativity. Because associativity follows the principle of temporal locality, for a sequence of random memory addresses that’s used for two or more times, the more associativity the cache has, the more hit rate it will get. The test case is generated based on that point.

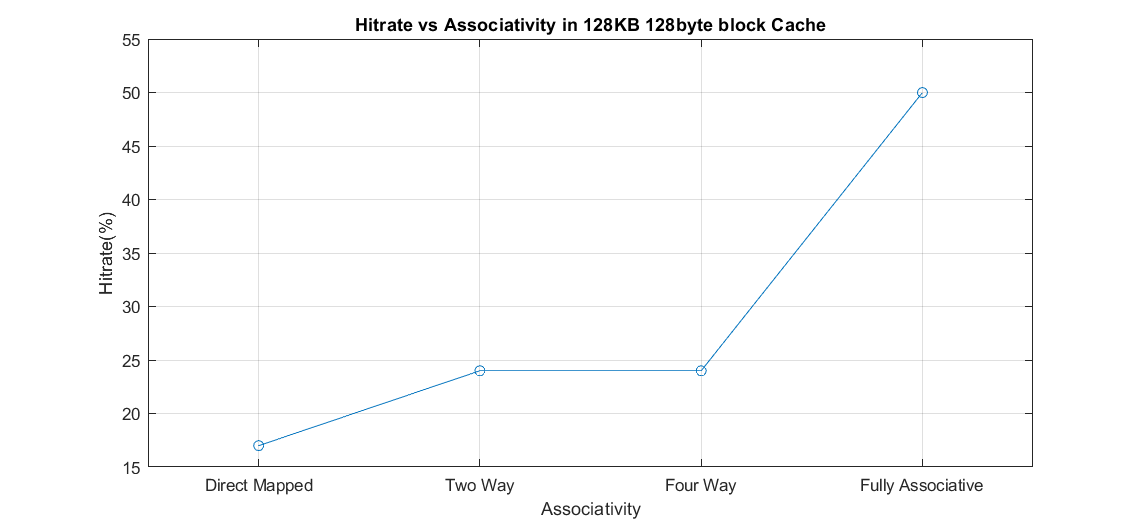


Figure Hitrate vs Associativity

This graph shows that for relatively random memory addresses, the more associative the cache is, the higher the hit rate.

The second test case is generated in favor of having a block size of 16 bytes for a 1KB Direct Mapped cache. Having a big block size follows the principle of spatial locality. However, having a very large block size relative to the cache size in a direct mapped cache can lower hit rate when the following pattern of memory operations are performed. Assume there are two direct-mapped caches, Cache A and Cache B, both are 1KB cache, referred as Cache A has a block size of 16 bytes referred as Cache B has a block size of 32byte. When a memory read at set index i and tag 0 for both cache are performed, Cache A will fetch 16 bytes of data from the memory, while Cache B will fetch 32 bytes from the memory. Note that one block in Cache B can fit two blocks in Cache A, so two block index in Cache A will be one block index in Cache B. If next time a Cache read at some address with a tag 1 but at block index i+1 for Cache A, the block index will still be i for Cache B. As a result, contents in block index i for Cache B will be evicted, however, Cache A stores both the data. When we read the content in the first memory address again, Cache A still has the information, it will result in a Cache Hit, for Cache B however, the content is no longer there, so it will result in a Cache Miss.

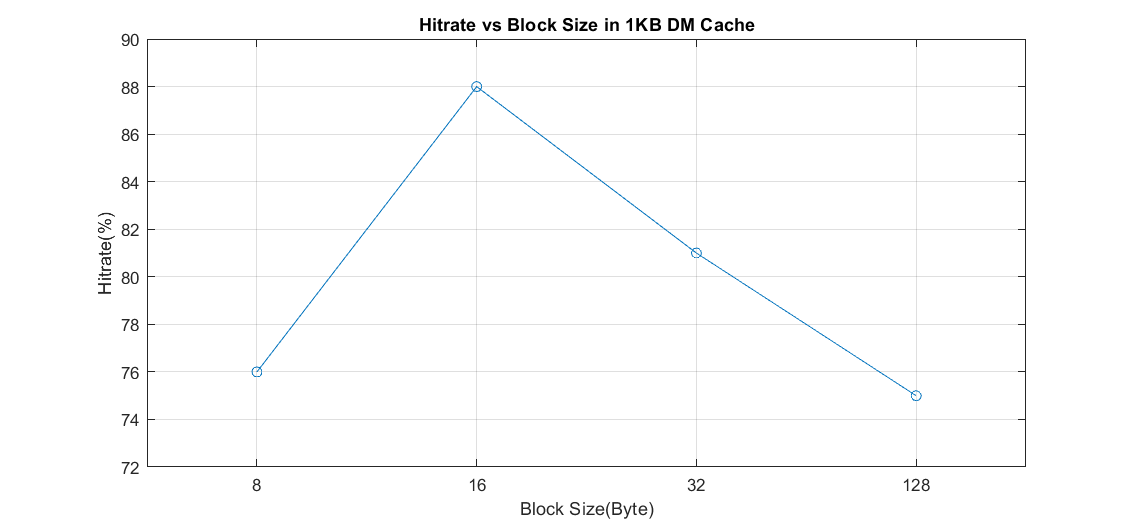


Figure Hitrate vs Block Size

This graph show that having an appropriate block size can lead to high hit rate, plus we get diminishing returns or even worse performance when we increase the block size.

**Conclusion**

After implementing the simulator and generated two test cases by myself, I have a deeper understanding of how exactly Caches work and the trade-off between spatial and temporal locality when designing caches for processors. I learned about the key differences between different kind of placement types and write policies. By comparing the number of bytes of data that caches write to memory, I can really see that write-back caches are much superior to write-through caches in case of performance. I can also see that we need to balance associativity and block size.