Christopher Gong

Computer Architecture Cache Simulator Report

Part 1

I believe both replacement policies, LRU and FIFO, as well as both write policies, write through and write back, work in my program.

FIFO write-through:

- For read hits, my program will increment the total number of hits.
- For read misses, my program will increment the total number of misses and reads. Then, it will proceed to look for the first empty/null block in the set. If there is a null block, then the null block is overwritten and added to the end of the queue. If the set is full, then the program will overwrite the first block in the queue, remove the evicted block from the queue, and add this new block to the end of the queue.
- For write hits, my program will increment the total number of hits and writes.
- For write misses, my program will increment the number of misses and reads. Then, it will proceed to look for the first empty/null block in the set. If there is a null block, then the null block is overwritten and added to the end of the queue. The number of writes is also incremented by one. If the set is full, then the program will overwrite the first block in the queue, increment the number of writes, remove the evicted block from the queue, and add this new block to the end of the queue.

FIFO write-back:

- For read hits, my program will increment the total number of hits and find the looked-up block in the queue to evict it and re-add it to the end of the queue.
- For read misses, my program will increment the total number of misses and reads. Then, it will proceed to look for the first empty/null block in the set. If there is a null block, then the null block is overwritten and added to the end of the queue. Since it’s the first time that data was written in this block, the block’s dirty bit is set to zero. If the set is full, then the program will first check if the first block in the queue has a dirty bit of one. If the dirty bit is one, then the program will increment the total number of writes, reset the block’s dirty bit to zero, overwrite the first block in the queue, remove the evicted block from the queue, and add this new block to the end of the queue.
- For write hits, my program will increment the total number of hits, and set the dirty bit of the looked up block to one.
- For write misses, my program will increment the total number of misses and reads. Then, it will proceed to look for the first empty/null block in the set. If there is a null block, then the null block is overwritten and added to the end of the queue. The block’s dirty bit is set to one. If the set is full, then the program will first check if the first block in the queue has a dirty bit of one. If the dirty bit is one, then the program will increment the total number of writes, reset the block’s dirty bit to one, overwrite the first block in the queue, remove the evicted block from the queue, and add this new block to the end of the queue.

LRU write-through:

- For read hits, my program will increment the total number of hits and find the looked up block in the queue. The block is then evicted and re-added to the end of the queue.

- For read misses, my program will increment the total number of misses and reads. Then, it will proceed to look for the first empty/null block in the set. If there is a null block, then the null block is overwritten and added to the end of the queue. If the set is full, then the program will overwrite the first block in the queue, remove the evicted block from the queue, and add this new block to the end of the queue.

- For write hits, my program will increment the total number of hits and writes. Then it will proceed to look for the looked-up block in the queue in order to evict it and re-add it to the end of the queue.

- For write misses, my program will increment the total number of misses and reads. Then, it will proceed to look for the first empty/null block in the set. If there is a null block, then the null block is overwritten, and added to the end of the queue. The number of writes is also incremented by one. If the set is full, then the program will overwrite the first block in the queue, increment the number of writes, remove the evicted block from the queue, and add this new block to the end of the queue.

LRU write through:

- For read hits, my program will increment the total number of hits and find the looked-up block in the queue to evict it and re-add it to the end of the queue. Then it will search for the looked-up block in the queue in order to evict it and read it to the end of the queue.
- For read misses, my program will increment the total number of misses and reads. Then, it will proceed to look for the first empty/null block in the set. If there is a null block, then the null block is overwritten and added to the end of the queue. Since it's the first time that data was written in this block, the block's dirty bit is set to zero. If the set is full, then the program will first check if the first block in the queue has a dirty bit of one. If the dirty bit is one, then the program will increment the total number of writes, reset the block's dirty bit to zero, overwrite the first block in the queue, remove the evicted block from the queue, and add this new block to the end of the queue.
- For write hits, my program will increment the total number of hits, and set the dirty bit of the looked up block to one. Then it will proceed to search for the looked-up block in the queue in order to evict it and re-add it to the queue.
- For write misses, my program will increment the total number of misses and reads. Then, it will proceed to look for the first empty/null block in the set. If there is a null block, then the null block is overwritten and added to the end of the queue. The block's dirty bit is set to one. If the set is full, then the program will first check if the first block in the queue has a dirty bit of one. If the dirty bit is one, then the program will increment the total number of writes, reset the block's dirty bit to one, overwrite the first block in the queue, remove the evicted block from the queue, and add this new block to the end of the queue.

<u>Part 2</u>

Results for 2-way associative cache of size 32, block size 4, **trace1.txt** with tag bits in the most significant position,

Trace1.txt Tag bits most significant	FIFO write through	FIFO write back	LRU write through	LRU write back
Memory reads	336	336	336	336
Memory writes	334	330	334	330
Cache hits	664	664	664	664
Cache misses	336	336	336	336
Hit rate	0.664	0.664	0.664	0.664
Miss rate	0.336	0.336	0.336	0.336

Results for 2-way associative cache of size 32, block size 4, **trace2.txt** with tag bits in the most significant position,

Trace2.txt Tag bits most significant	FIFO write through	FIFO write back	LRU write through	LRU write back
Memory reads	3499	3499	3292	3292
Memory writes	2861	2855	2861	2853
Cache hits	6501	6501	6708	6708
Cache misses	3499	3499	3292	3292
Hit rate	0.6501	0.6501	0.6708	0.6708
Miss rate	0.3499	0.3499	0.3292	0.3292

Results for 2-way associative cache of size 32, block size 4, **trace1.txt** with index bits in the most significant position,

Trace1.txt Index bits most significant	FIFO write through	FIFO write back	LRU write through	LRU write back
Memory reads	336	336	336	336
Memory writes	334	332	334	332
Cache hits	664	664	664	664
Cache misses	336	336	336	336
Hit rate	0.664	0.664	0.644	0.644
Miss rate	0.336	0.336	0.336	0.366

Results for 2-way associative cache of size 32, block size 4, **trace2.txt** with index bits in the most significant position,

Trace2.txt Index bits most significant	FIFO write through	FIFO write back	LRU write through	LRU write back
Memory reads	3942	3942	3943	3943
Memory writes	2861	2861	2861	2861
Cache hits	6058	6058	6057	6057

Cache misses	3942	3942	3943	3943
Hit rate	0.6058	0.6058	0.6057	0.6057
Miss rate	0.3942	0.3942	0.3943	0.3943

When the middle bits are used as the set index, the cache performance will improve in terms of hit rate, especially for larger files. In this case, for mytrace2.txt we notice that the hit rate decreased when we used the most significant bits for the set index. We didn't notice this for mytrace1.txt because it was a relatively smaller file. The bigger the file, the bigger the decrease in hit rate and the worse the performance. The reason for the poor performance is the poor spatial locality. Using high order bit indexing will result in adjacent memory lines mapping to the same cache entry. With middle order bit indexing, consecutive memory lines are more likely to map to different cache lines. Therefore, the hit rate goes up in this case.