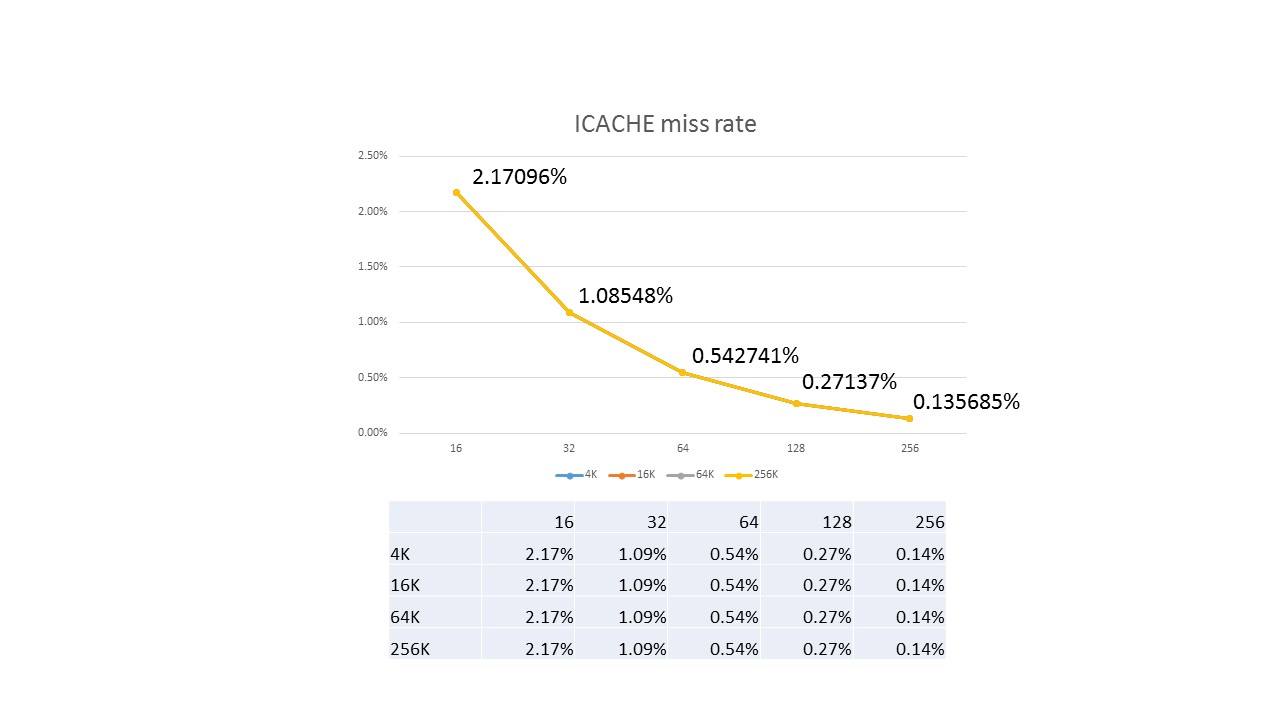
**Computer Organization**

**0416021曾香耘、0416246王彥茹**

**Direct Mapped:**

ICACHE miss rate:

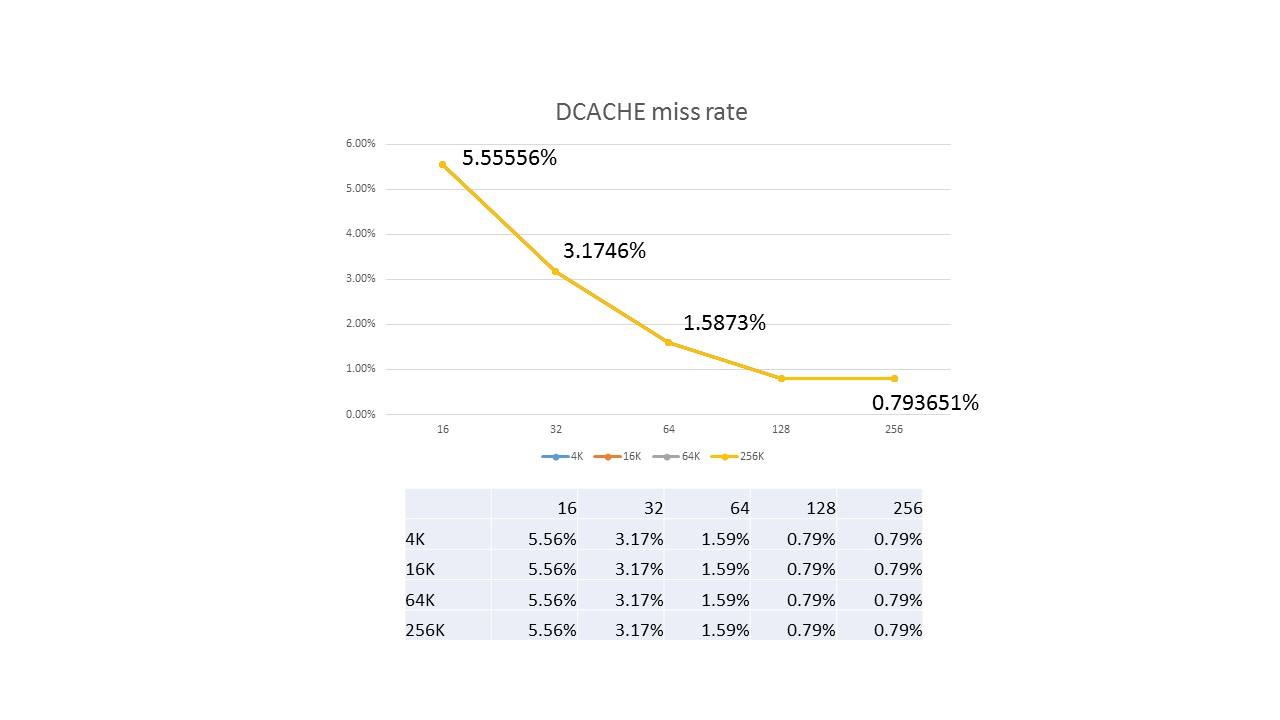


The bigger the block size is, the less we have to fetch from the main memory (assuming there is one beneath our cache in the hierarchy), so the miss rate is lower since we decrease compulsory misses which are caused by first-time accesses.

The miss rate has not changed regarding the cache size because the actual address we access is way too little such that a 4K cache is big enough and there doesn’t exist any capacity misses.

Overall, the miss rate is low because ICACHE is having good spatial locality.

DCACHE miss rate:



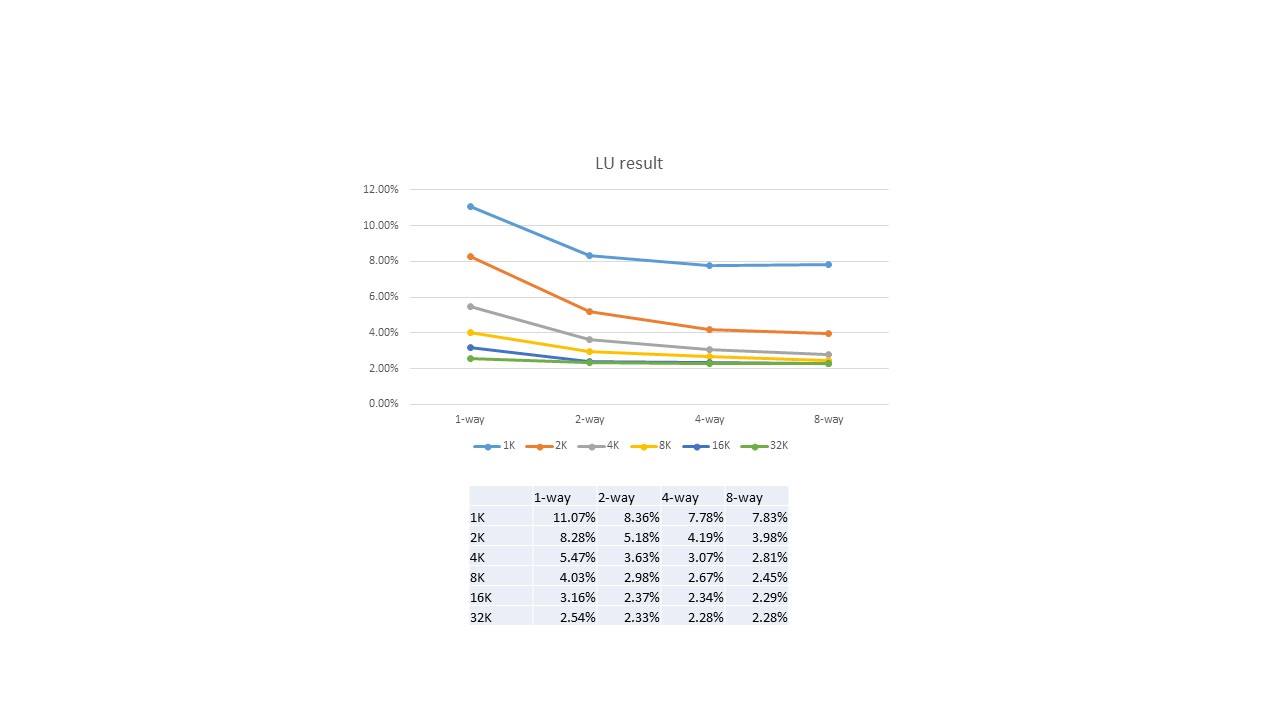
The bigger the block size is, the less we have to fetch from the main memory (assuming there is one beneath our cache in the hierarchy), so the miss rate is lower since we decrease compulsory misses which are caused by first-time accesses.

The miss rate has not changed regarding the cache size because the actual address we access is way too little such that a 4K cache is big enough and there doesn’t exist any capacity misses.

DCACHE does not possess good spatial locality in contrast to ICACHE, thus the miss rate is higher overall.

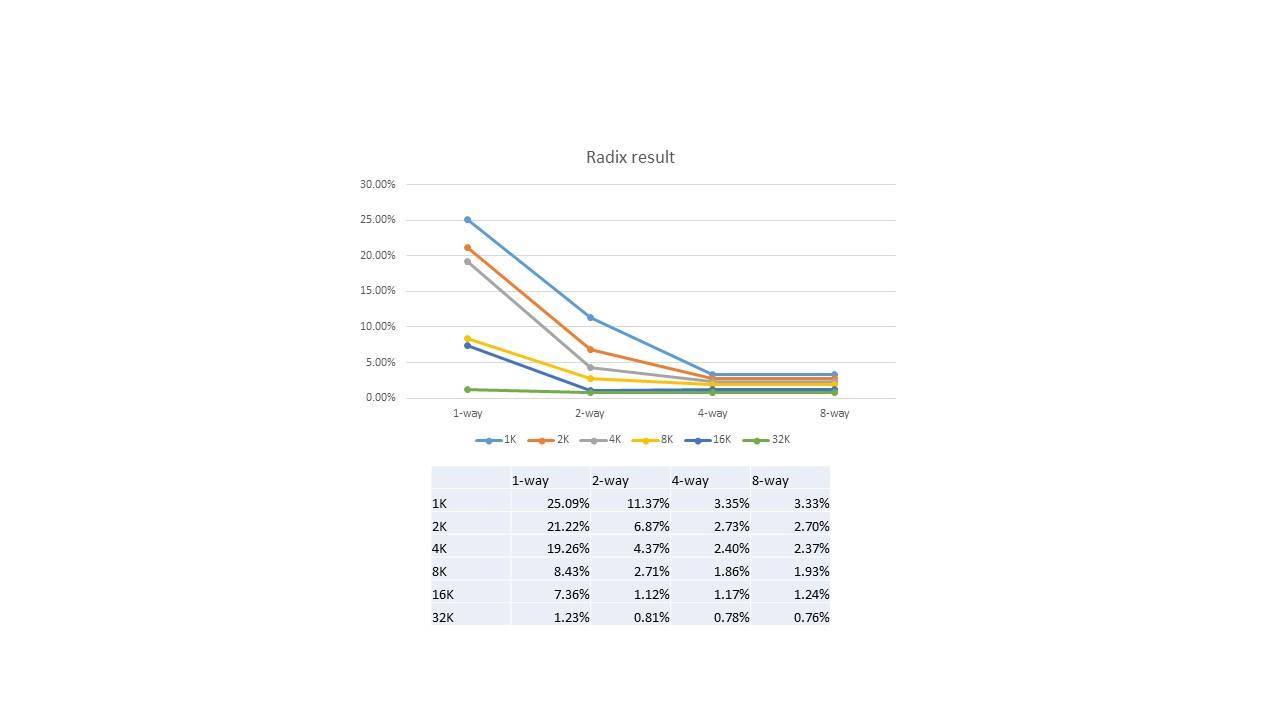
**N-way Set-associative With LRU Replacement:**

LU miss rate:



When we increase the cache size, we decrease capacity misses, thus the miss rate is lower; when we increase associativity, we decrease conflict misses, so the competition in a single set will be less intense, and thus the miss rate is lower.

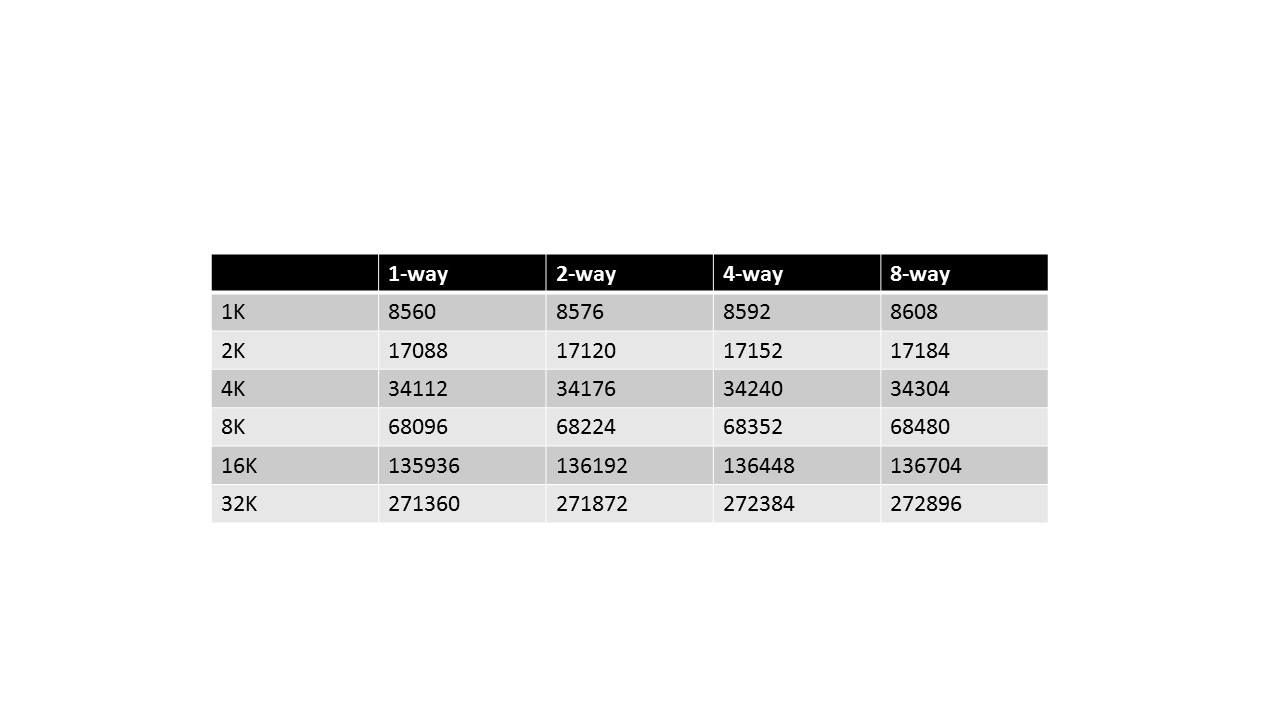
RADIX miss rate:



When we increase the cache size, we decrease capacity misses, thus the miss rate is lower; when we increase associativity, we decrease conflict misses, so the competition in a single set will be less intense, and thus the miss rate is lower.

For some reason, RADIX is more sensitive to increasing associativity than LU, but unfortunately we figure out nothing from the data itself.

Cache size (in bits):



We did not consider the time bits used to implement LRU. (If needed, each is incremented with number of time bits \* number of blocks.)

Consider 32 bit byte-address, cache size 2^n blocks, block data size 2^m bytes, 2^k-way associative.

Tag size: 32 – ( n – k + m)

Total number of bits: 2^n \* (2^m \* 8 + 33 - n + k - m)

**Comments On CPP:**

We basically implemented the cpp file as stated. For the LRU replacement, we first go through the cache to look for hits, if a miss is confirmed, we look for the least recently used index with a value named time, which indicates the last time it was accessed.

We found some problem in the given cpp file.

“int offset\_bit = (int) log2(block\_size);”

should be written as

“int offset\_bit = (int) round(log2(block\_size));”

Otherwise when block\_size = 8, offset\_bit = 2.