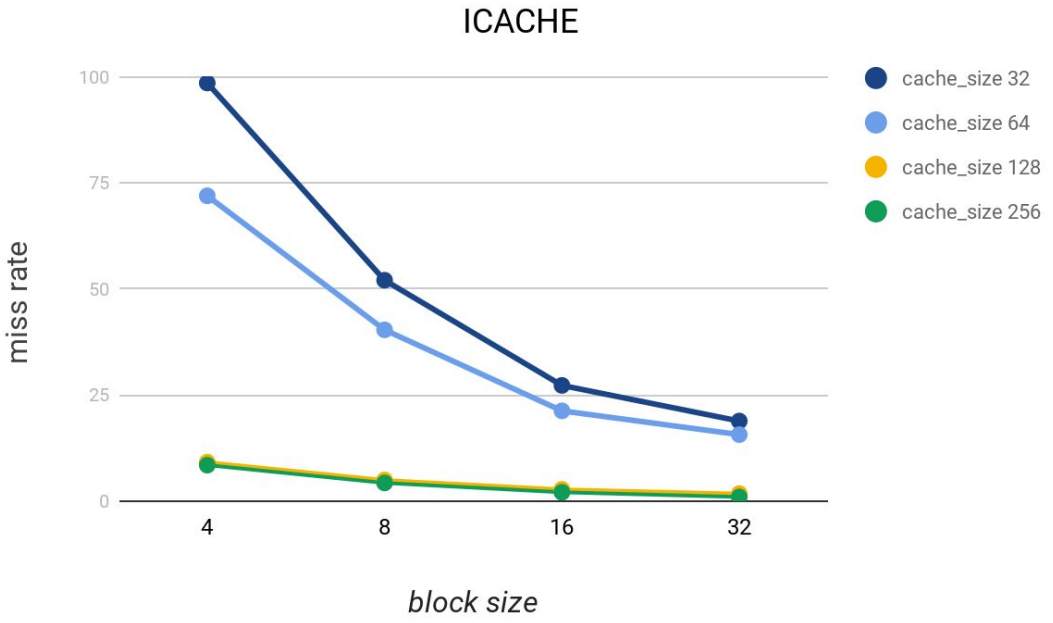
Computer Organization

ID: 0616214 0616221

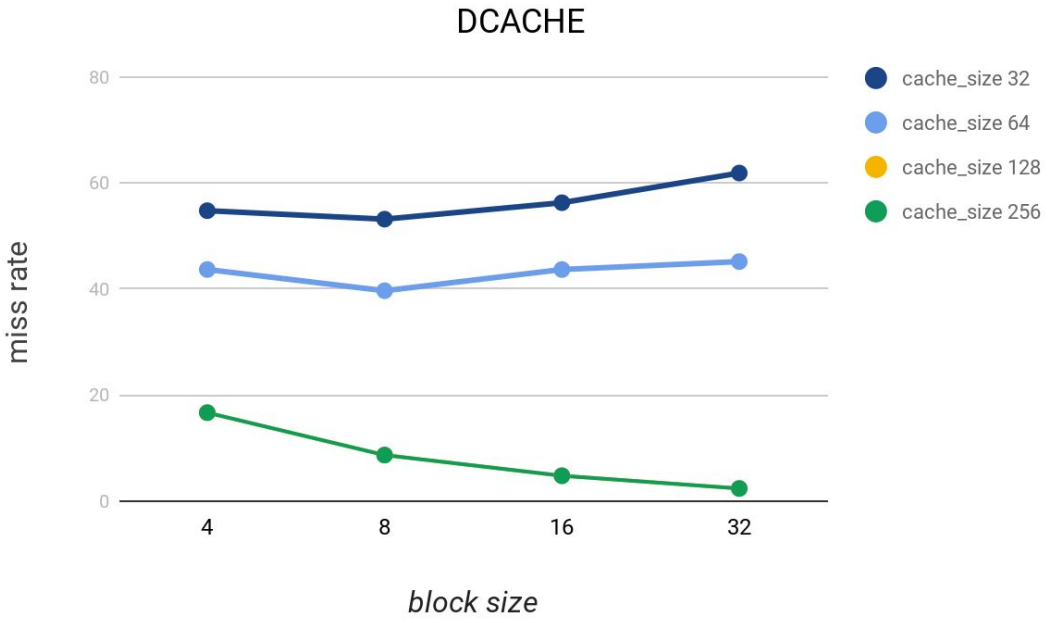
I. Basic problem: (p391)

1. ICACHE:



a. **Larger cache size:** It means that more blocks to use, and that leads to less collision would be occurred. Since the first bits of addresses in the ICACHE.txt file are all zeros, the affection of increasing cache size to 128 and 256 would not be such visible.
b. **Larger block size**: It means that more spatiality to utilize. Because the distribution of addresses in ICACHE.txt file is so linear, the affection of more spatiality would overstep the less-block disadvantage which would be occurred in random addresses distribution (showed in the picture of CO_Lab3.pdf).

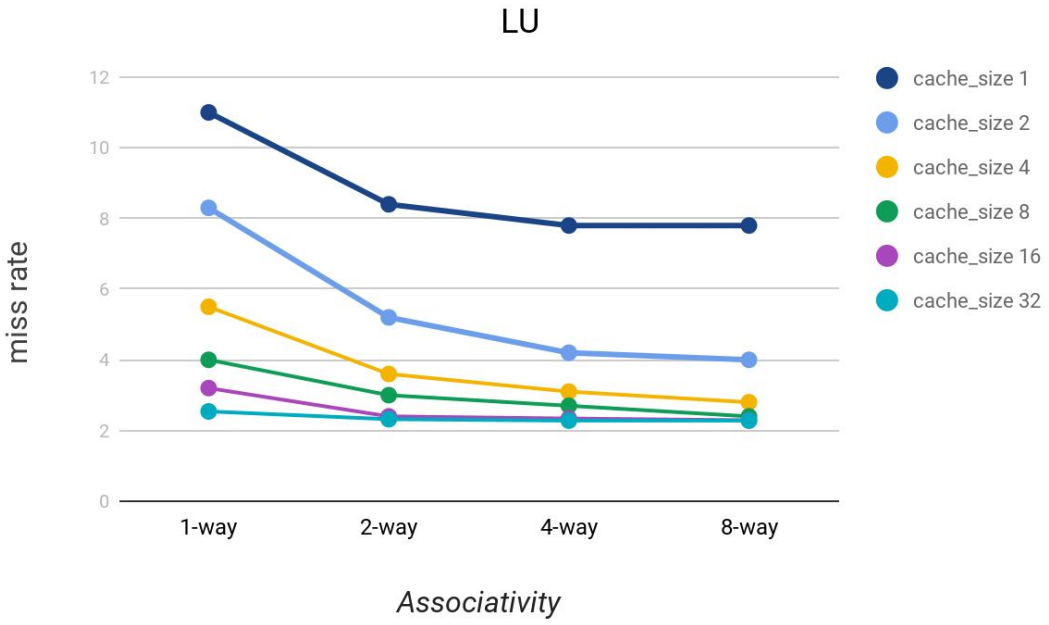
2. DCACHE:



a. **Larger cache size:** Like the situation of ICACHE.txt. (showed previously) By the way, cache size 128 and 256 are overlapped.
b. **Larger block size:** Different from ICACHE.txt, the distribution of addresses in DCACHE.txt is more random. As to result, the less-block disadvantage would be so effective when the cache size is small (such like 32 and 64 in picture above).

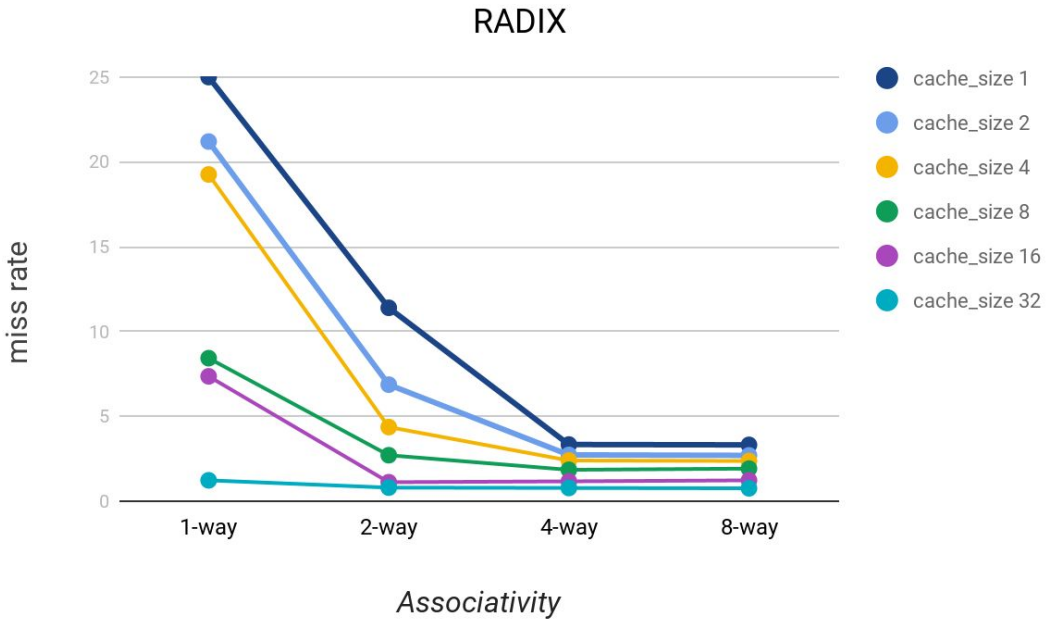
II. Advanced problem: (p455)

1. LU (LU decomposition)



a. **Larger cache size:** It means more blocks and less collisions.
b. **More associativity:** It means less collision which corresponds to the index. When cache size increase to large enough (16 and 32), associativity would be almost no effective.

2. RADIX (Radix sort)



a. **More associativity and larger cache size:** Same as the situation of LU.
b. **By this sorting algorithm**, only increase the associativity up to 4-way, we could limit the miss rate down to 3% (the same cache size and associativity in LU would cost more!)

3. Total bits table (including tags and one valid bit)

	1-way	2-way	4-way	8-way
1K	880	896	912	928
2K	1728	1760	1792	1824
4K	3392	3456	3520	3584
8K	6656	6784	6912	7040
16K	13056	13312	13568	13824
32K	25600	26112	26624	27136

The formula:

- index bit = log2(cache size / 64 / set number)
- tag bits = 32 - index bit - 6
- bits per block = tag bits + 32(data) + 1(valid)
- total bits = bits per block * (cache_size / 64)