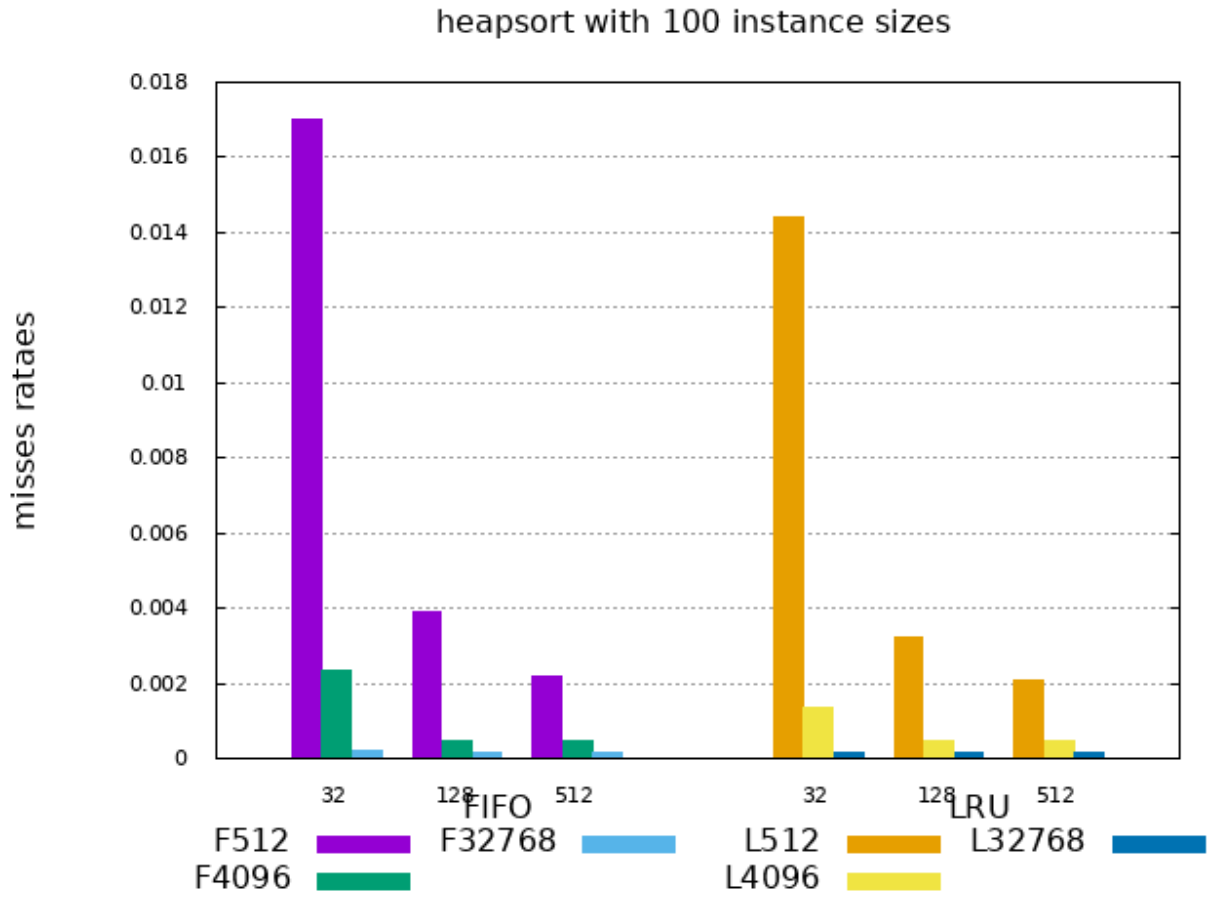
Report of TLB Performance

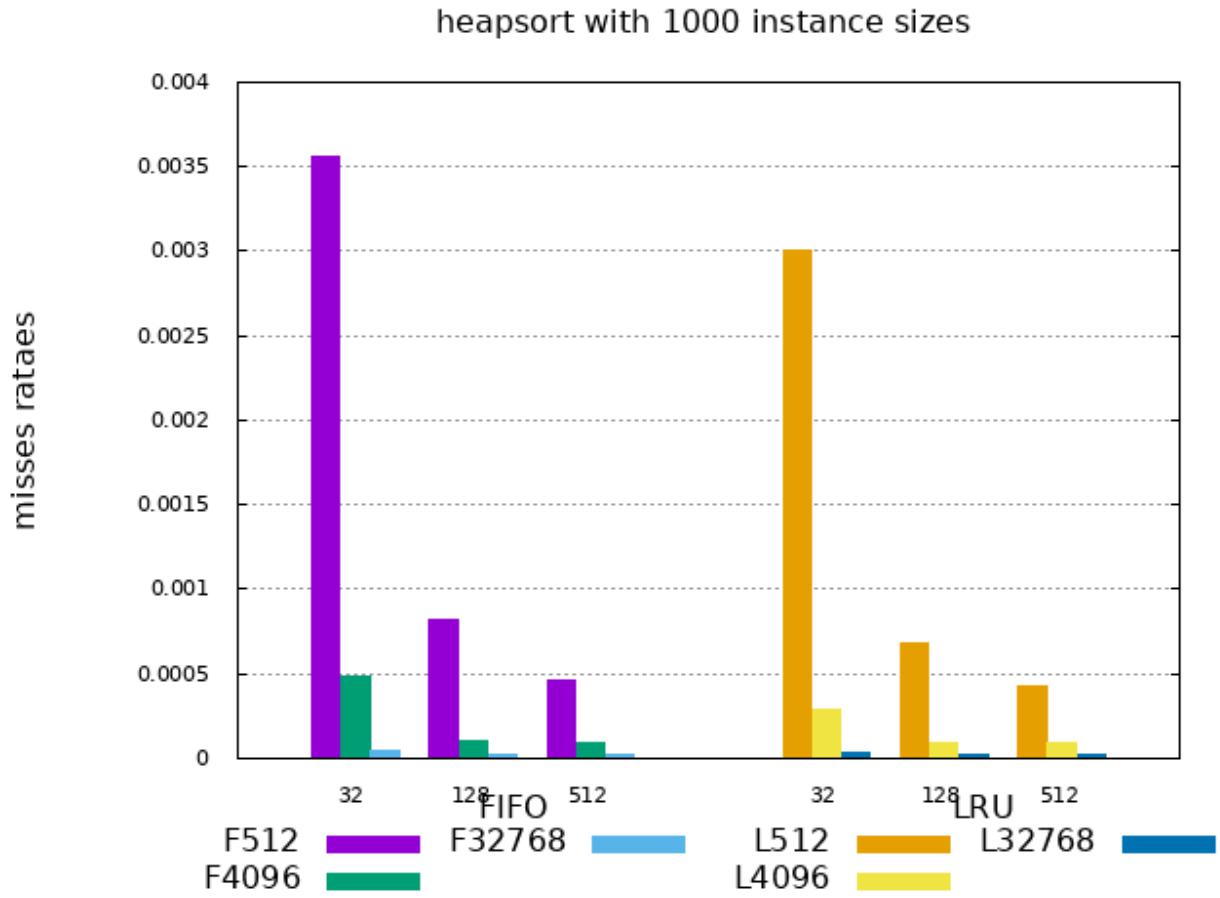
DESIGN

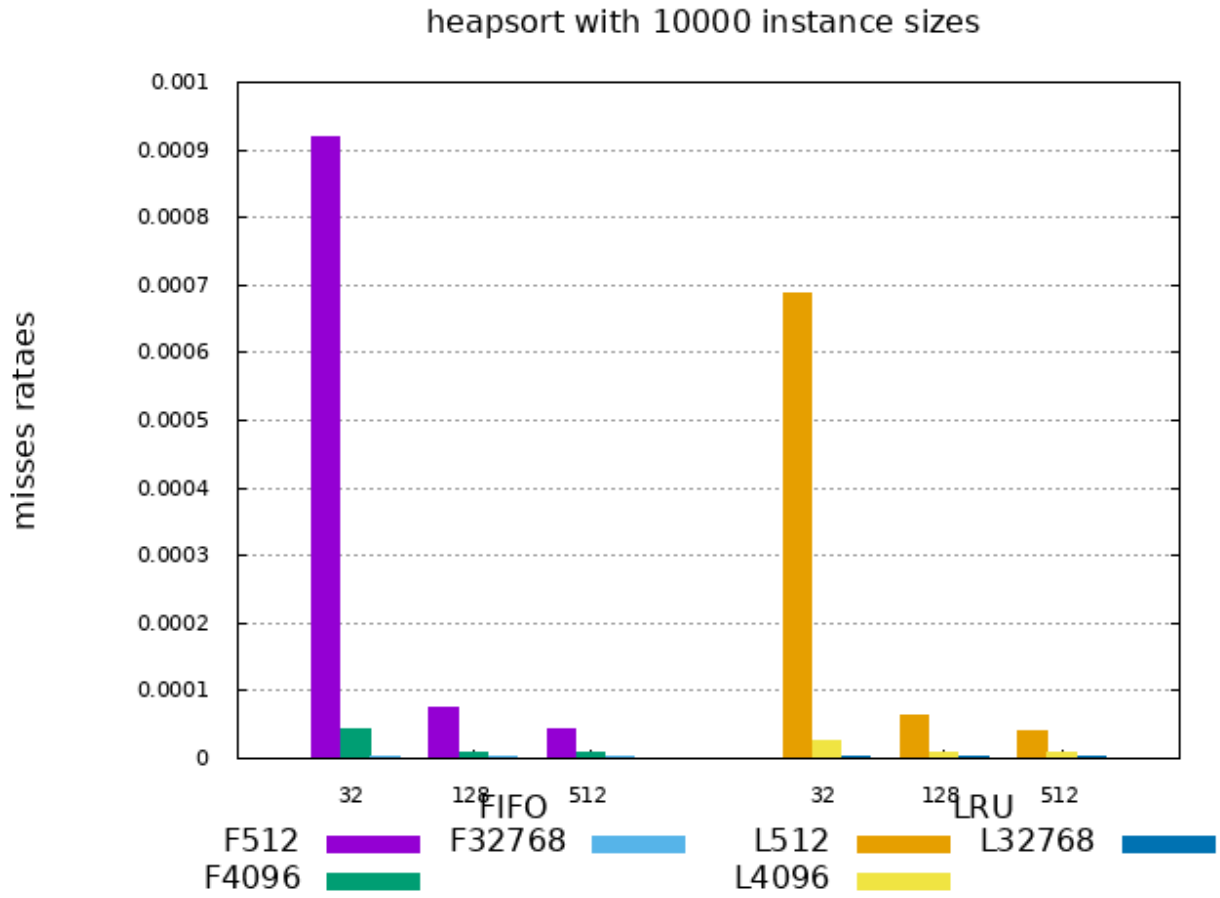
For this assignment. To analyze the TLB performance. I have generated 10 graphs by using gnuplot with following combinations: 3 programs x 3 instance sizes (100, 1000, 10000) x 2 policies x 3 page sizes (512, 4096, 32768) x 3 tlb sizes (32, 128, 512) [no flush period, no -i option] and a heapsort with -i option, 10000 instance and other parameters. This is total of 180 runs. The tested programs are the heapsort, the mergesort, and the quicksort.

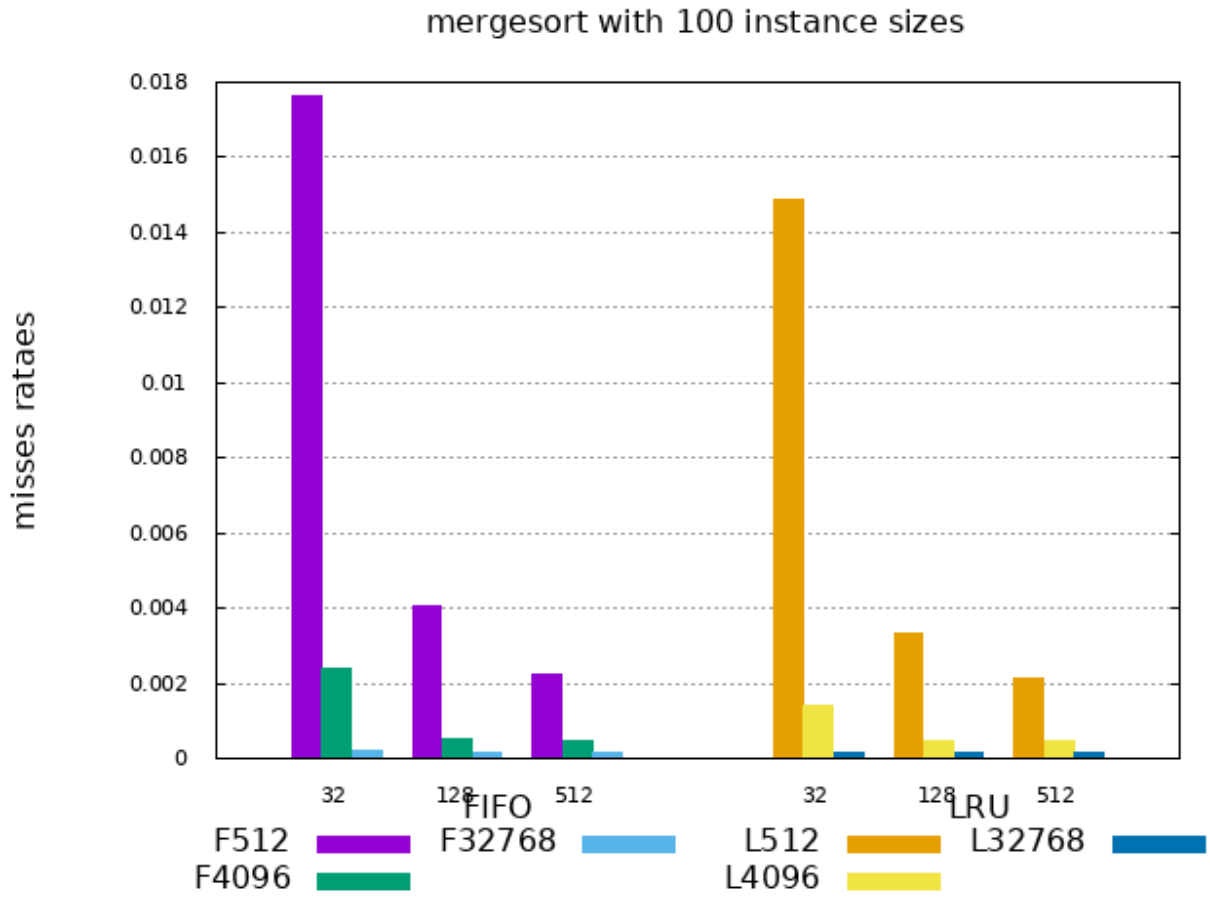
ANALYZATION

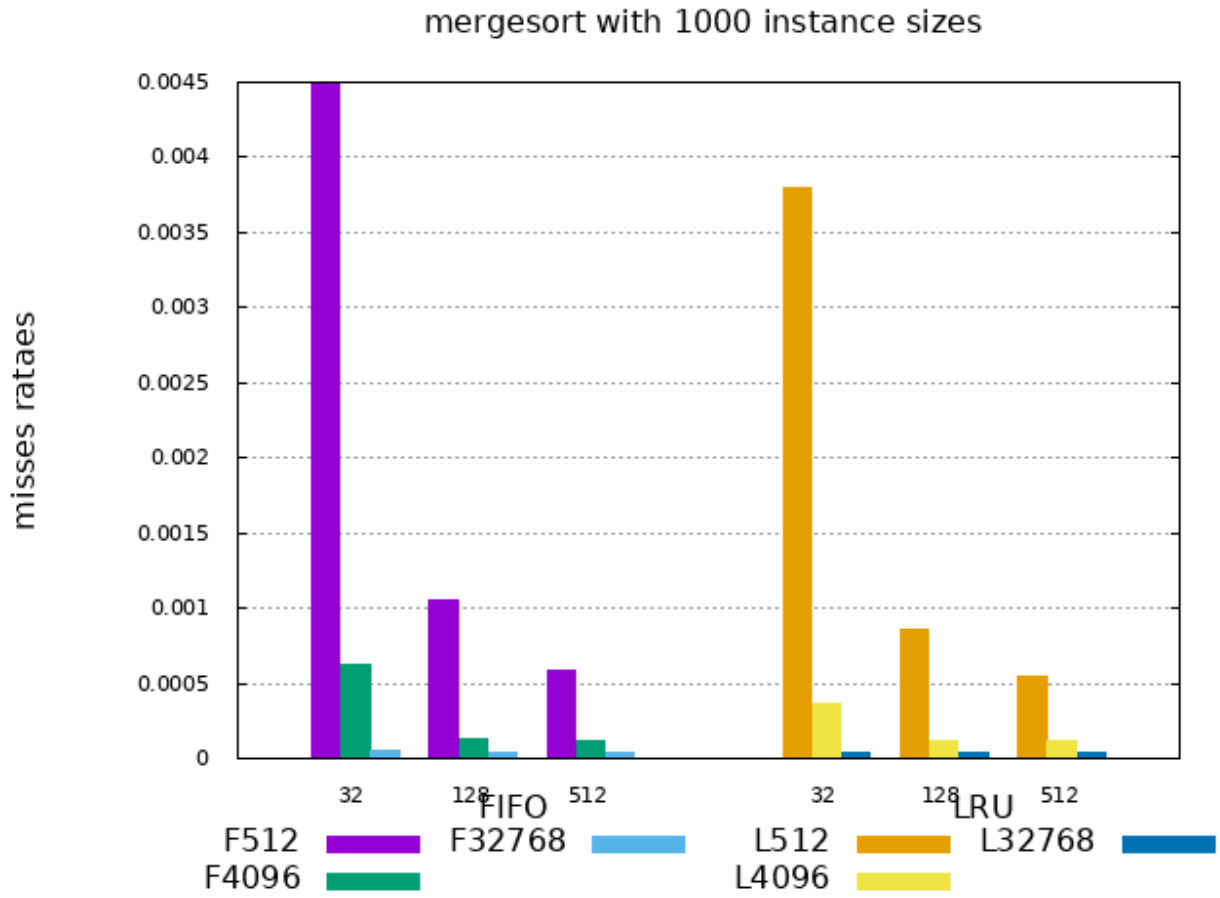
By study the differences between these graphs. We can notice that, with identical settings, the results of TLB miss rate by LRU policy is relatively lower than the miss rate of using FIFO policy. The reason for these outcomes might be the LRU_search searchs the least recently used node. Therefore, when programs are executing, LRU stores the address used frequently, thus improve the efficiency and reduce the miss rate. When comparing the result of heapsort 10000 instance with -i argument and the result of heapsort 10000 instance without -i argument, we can notice that, the outcome with -i argument will result a higher miss rate, since the -i argument makes the program to completely ignore from processing the memory references that are instruction fetches, thus the denominator decreases. However, the misses stay about the same, therefore, the miss rate is higher. As for different instance size, we can find that the miss rate becomes lower as the instance size goes larger. This is because the growth rate of misses(numerator) is much lower than the growth rate of instances(denominator).For page size and tlb size, by observing the different bars on different x-axis, we can notice that as the page size or tlb size goes up, the miss rate goes down. General speaking, larger page size/tlb size help the tlb performance a lot. Last but not Least, by comparing these 3 sorting algorithm, we can notice that the heap sort has a higher miss rate, the merge sort and the quick sort are about the same in terms of tlb miss rate. This might due to the complexity of sorting algorithm themselves.

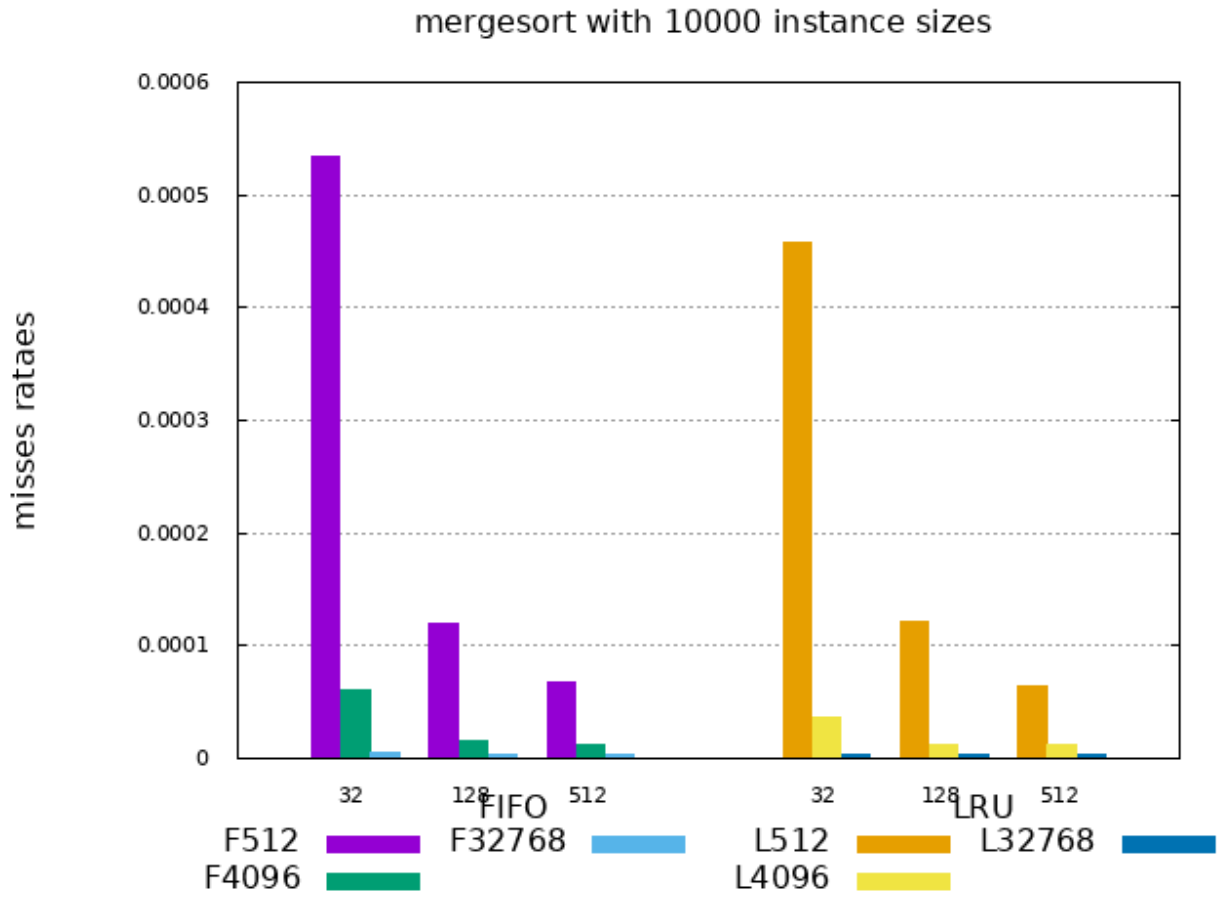

heapsort with 100 instance sizes
misses rataes
0.018
0.016
0.014
0.012
0.01
0.008
0.006
0.004
0.002
0
32
128
512
32
128
512
FIFO
LRU
F512
F4096
F32768
L512
L4096
L32768

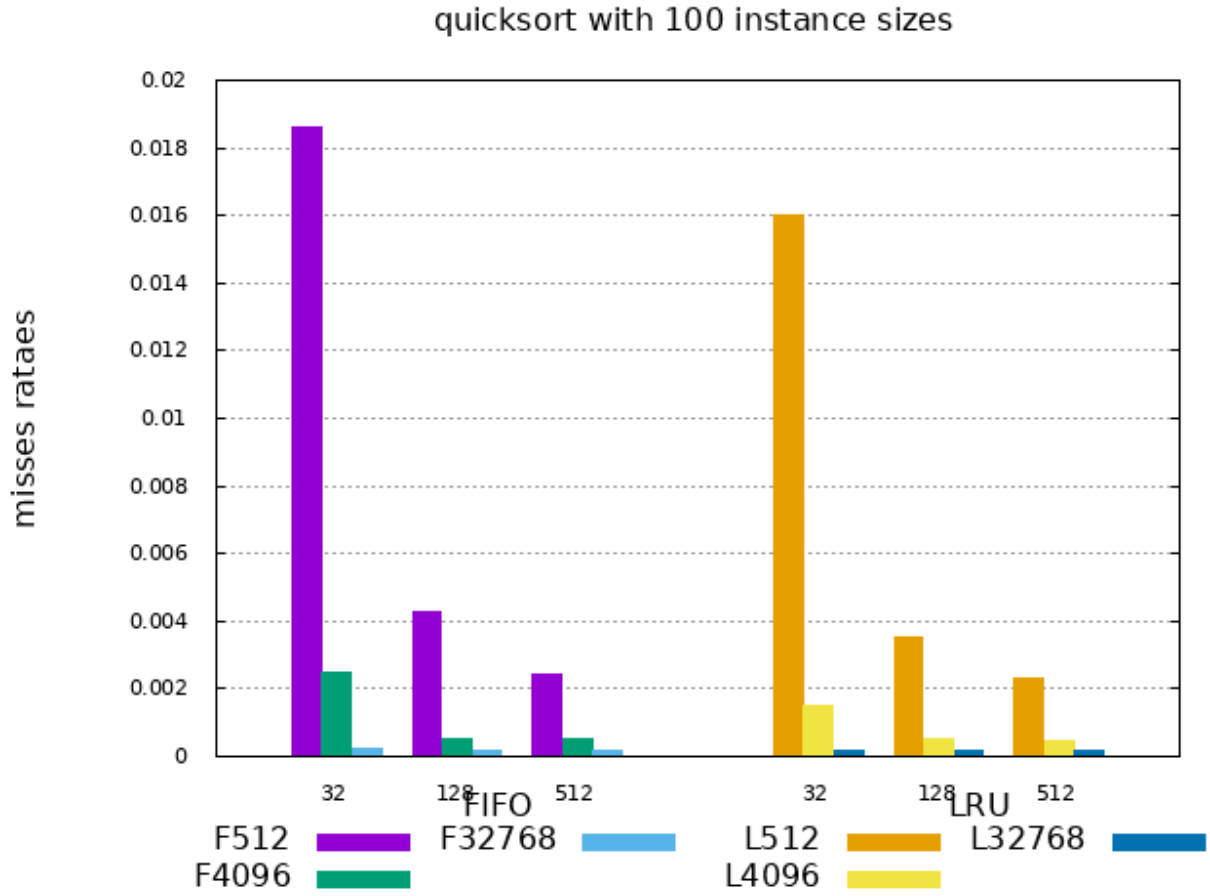

heapsort with 1000 instance sizes
misses rataes
0.004
0.0035
0.003
0.0025
0.002
0.0015
0.001
0.0005
0
32
128
512
32
128
512
FIFO
LRU
F512
F4096
F32768
L512
L4096
L32768

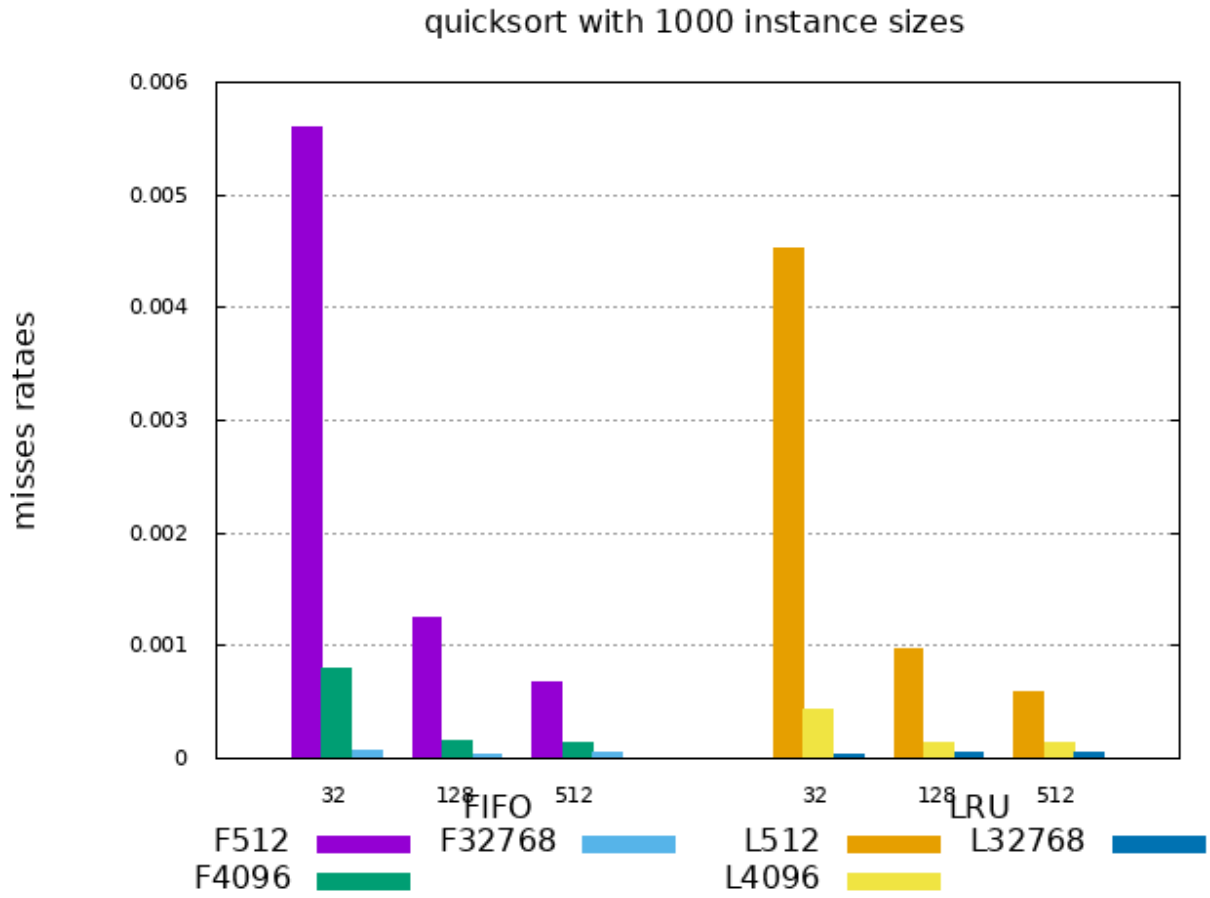

heapsort with 10000 instance sizes
misses rataes
0.001
0.0009
0.0008
0.0007
0.0006
0.0005
0.0004
0.0003
0.0002
0.0001
0
32
128
512
FIFO
32
128
512
LRU
F512
F4096
F32768
L512
L4096
L32768

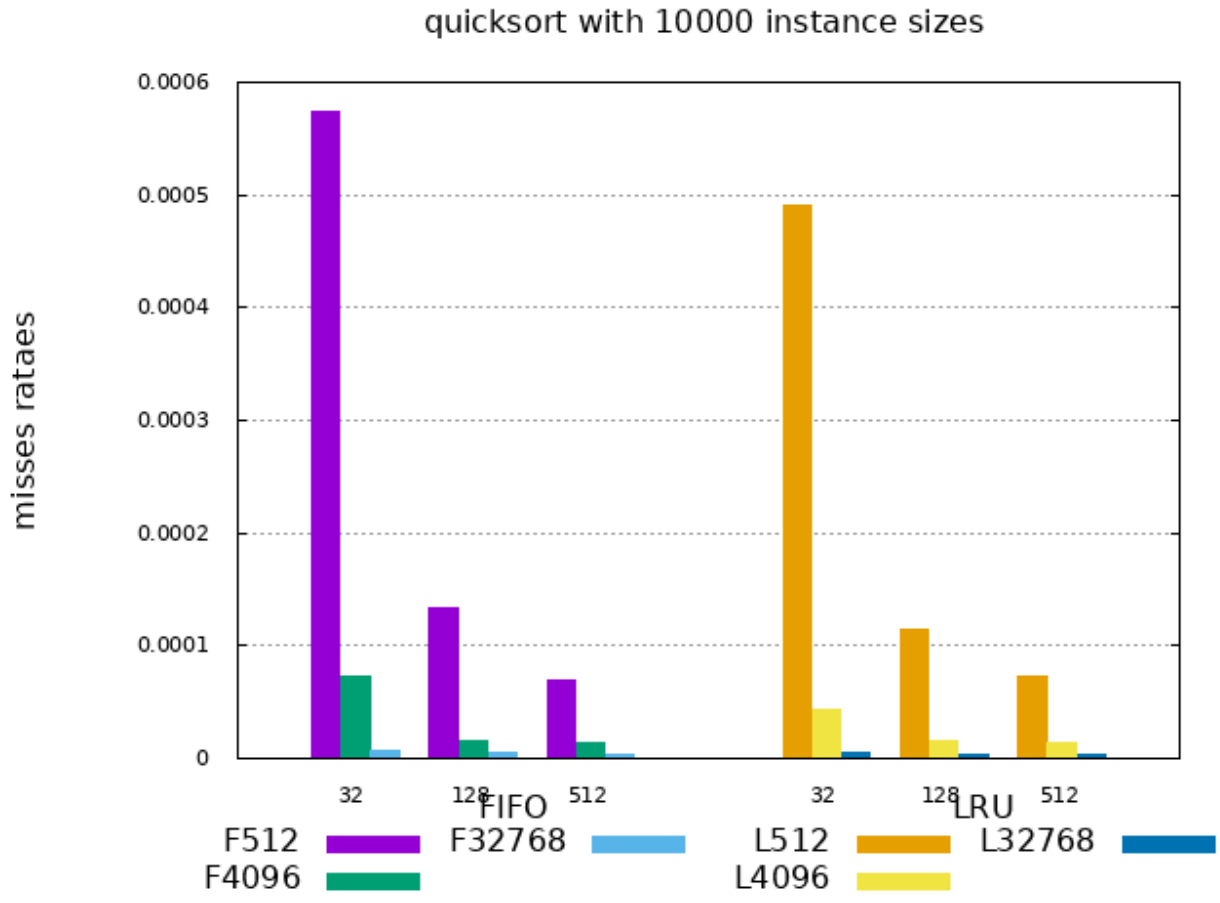

mergesort with 100 instance sizes
misses rataes
0.018
0.016
0.014
0.012
0.01
0.008
0.006
0.004
0.002
0
32
128
512
32
128
512
FIFO
LRU
F512
F4096
F32768
L512
L4096
L32768

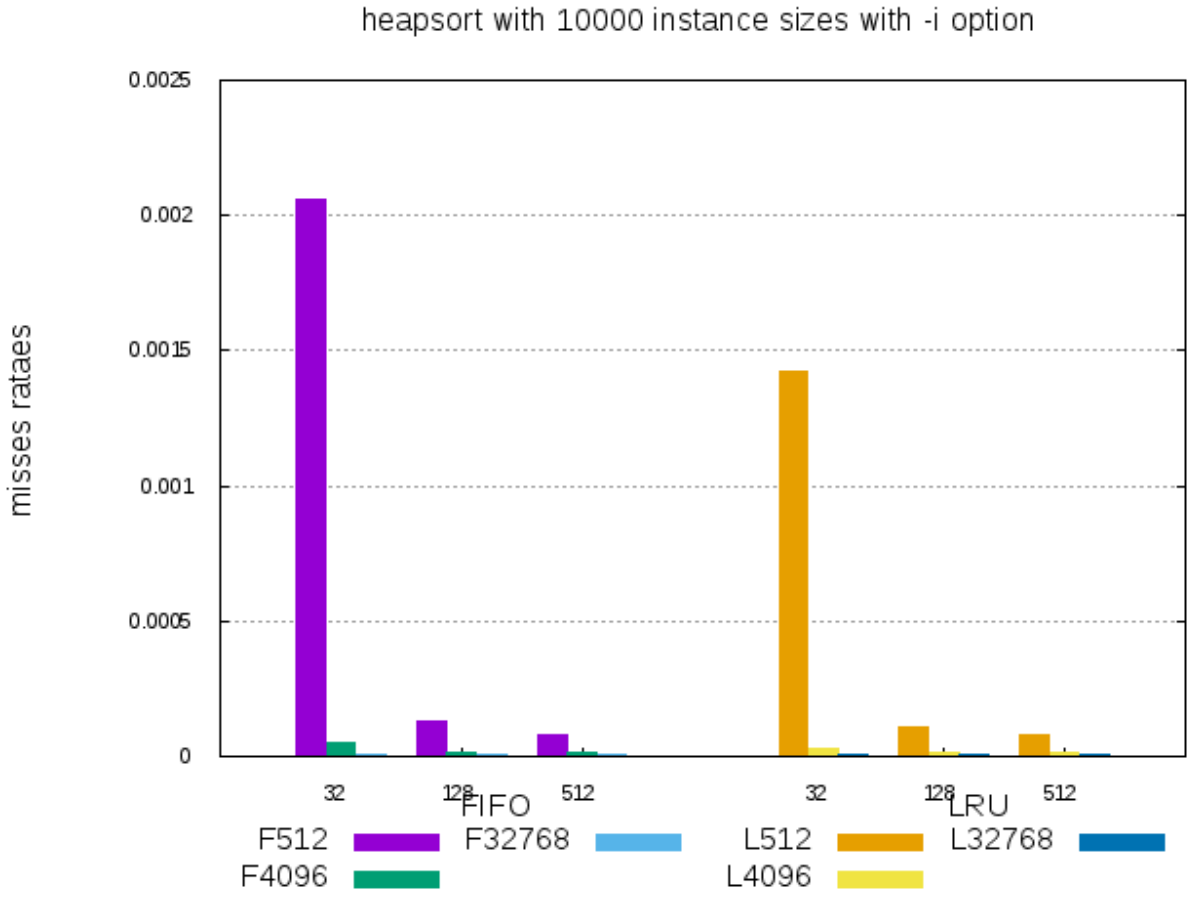

mergesort with 1000 instance sizes
misses rataes
0.0045
0.004
0.0035
0.003
0.0025
0.002
0.0015
0.001
0.0005
0
32
128
512
32
128
512
FIFO
LRU
F512
F4096
F32768
L512
L4096
L32768


mergesort with 10000 instance sizes
misses rataes
0.0006
0.0005
0.0004
0.0003
0.0002
0.0001
0
32
128
512
FIFO
32
128
512
LRU
F512
F4096
F32768
L512
L4096
L32768


quicksort with 100 instance sizes
misses rataes
0.02
0.018
0.016
0.014
0.012
0.01
0.008
0.006
0.004
0.002
0
32
128
512
FIFO
F512
F4096
F32768
32
128
512
LRU
L512
L4096
L32768


quicksort with 1000 instance sizes
misses rataes
0.006
0.005
0.004
0.003
0.002
0.001
0
32
128
512
32
128
512
FIFO
LRU
F512
F4096
F32768
L512
L4096
L32768


quicksort with 10000 instance sizes
misses rataes
0.0006
0.0005
0.0004
0.0003
0.0002
0.0001
0
32
128
512
32
128
512
FIFO
LRU
F512
F4096
F32768
L512
L4096
L32768


heapsort with 10000 instance sizes with -i option
misses rataes
0.0025
0.002
0.0015
0.001
0.0005
0
32
128
512
FIFO
32
128
512
LRU
F512
F32768
F4096
L512
L32768
L4096