TLB Performance

Table of Contents

Results

The valtlb379 program is the implementation of the TLB simulator which is able to simulate the behavior of a TLB that stores and handles, indiscriminately, page table entries needed for both instruction and data references. The output results in the terminal include the number of hits and miss, as well as the total number of references. We have collected these data and visualized them by execl, with the aim of examining the effect of different scenarios on the performance of TLB activity. To be specific, there are following combination that we have tested: three different sample programs, three different instance size, FIFO and LRU eviction policy, three pagesizes, as well as three TLB sizes. We also executed several cases of with the instruction address, i, and with the flush period, f for example program. Noticeably, all the plots with clear label generated based on the output result have been attached in the Appendix section. These graphs cover all the combinations stated above, and we indeed compared and analyzed the patterns in the Discussion section.

Discussion

First of all, we have used three sorting codes for the simulation of TLB. As we can see, three different sample programs show the similar trend while the value of miss rates are quite close. For example, under the same policy, tlbsize, pagesize, and instance, the miss rates of heapsort are slightly smaller than the mergesort and quicksort as shown in Fig.1, 13, 25. We think that different sort algorithms have their specific sorting method and the method for memory reference allocation, and the improvement of the cache locality of mergesort and quicksort can reduce the number of TLB misses efficiently.

In addition, we tried out different instance sizes which are 100, 10000, 100000 for small, medium, large respectively of the same program and parameters. According to the quicksort refer to Fig.26, 27, 28, the increase of instance of the program leads to the decrease of the miss rate. Since the total number of references dramatically raised as increasing instance size, the possibility of hit becomes larger. Although the number of miss pages goes up a little, but the total page number increase more, so the miss rate goes down.

In terms of the policy effect on the TLB performance, the LRU obviously perform better than the FIFO regarding the miss rate, because LRU keeps the pages that were most recently used, whereas the FIFO keep the pages that were most recently added. FIFO has a painful disadvantage which is it does not account for "permanently needed pages". According to temporal locality of reference, memory that has been accessed recently is more likely to be accessed again soon, since old need of the page does not mean useless. In this case, the miss rate for the LRU is smaller than the FIFO by comparing the Fig.1-6 with Fig.7-12 correspondingly.

Progressing to exam the effect of page size on the TLB performance by choosing the size of 512, 4096, 32768. It is clear that the address and the page size are used to calculate the page number. As the page size grows, the total memory reference number drops a little, with the decreasing rate of page number becoming smaller. As a result, the number of miss decrease drastically according to the Fig.1, 13, 25 for all sorting algorithms. Then, the miss rate tends to decrease.

Furthermore, the TLB size also plays a significant role in the performance of TLB. By looking at the Figures of all the cases, we can discover that the miss rates generally demonstrate a decline trend as the TLB size goes from 32 to 128 then to 512. The lager TLB size will provide more space to store distinct page numbers, so when new page number coming to the TLB, it is more likely to hit the existing page number stored in the TLB previously.

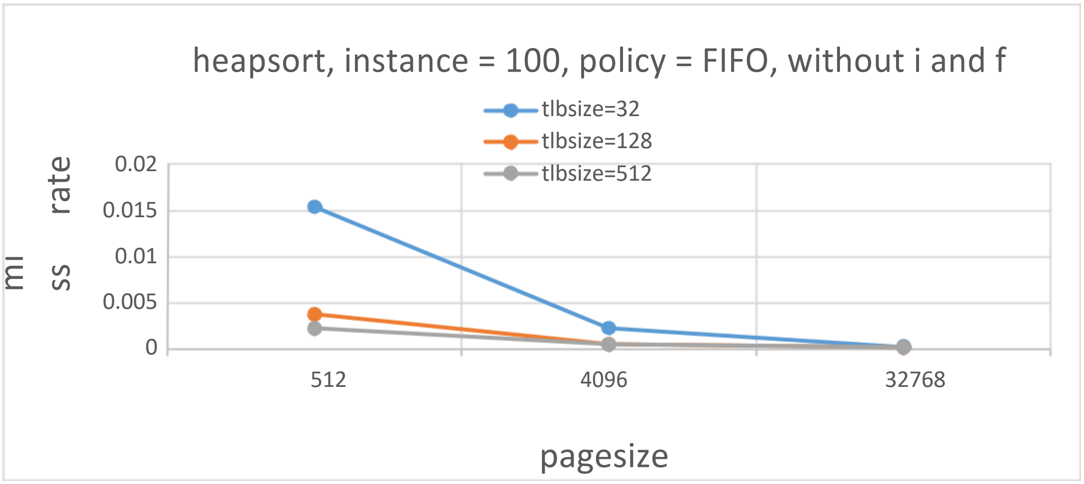
Finally, we have compared the results with instruction address -i to the results without -i. The Fig.5, 11, 17, 23, 29, 35, clearly indicate the execution with -i has higher miss rate than that without -i overall. By adding the optional parameter, -i, the program completely ignore form processing the memory references that are instruction fetches. It is known that the instruction memory references will be reused frequently as the program processing. Thus, when we ignore the instruction fetches, the hit cases will decrease significantly while the total number of references will also decrease. However, the decline rate in the miss is much smaller than that of in the hit. Therefore, the final result of the miss rate in the program execution with -i parameter is higher than the miss rate in that without -i.

Speaking of the influence of flush period, there is extremely large gap between the miss rate of the TLB with f=500 and that of TLB without flush period. If we increase the flush period, the miss rate will consequently go down. Since bigger flush period means that the existing page number entering previously will be kept longer in the TLB, and when a new page number coming, it more likely to hit. Therefore, we can conclude that the flush period will clean up the TLB, then, the original page will not in the TLB anymore, at this time, the number of miss will actually increase.

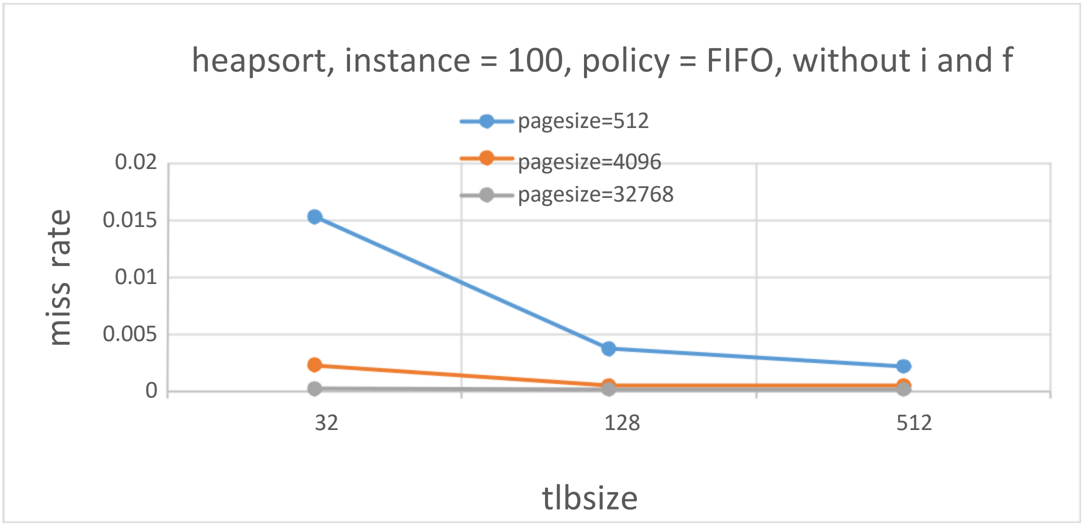
Conclusion

Overall, the objective of this lab is to create a simulation to study the performance of the TLB in terms of several factors. In details, the code has implemented two page replacement algorithms which are FIFO, LRU. In order to cover all the combinations for given parameters and sample sorting program, we ran the basic 162 cases to generate graph representations of the resulting miss rate as y scale for different combinations. In addition to that, we tested the -i and -f options for each example program and each policy. As discussed above, the following factors: programs, instance sizes, policies, page sizes, tlb sizes and -i and -f parameters, are all able to impact the performance in a certain way. As mentioned, we chose three different instance sizes which are small: 100, medium: 10000, large: 100000 for the input of the sort program. The speed of obtaining the output result is fast; for example, the times taken to run the small and medium size cases are almost instant. For the large size, the program required to spend less than 4 minutes. Further, we tried out the size of 1000000 for over 20 minutes. But for the maximum instance size, the program took couple hours to receive the outputs. Indeed, we have investigated the performance of the TLB through our code with good efficiency. Also, we have explored the reasons and basic principles of the observed behaviors for TLB under unique conditions.

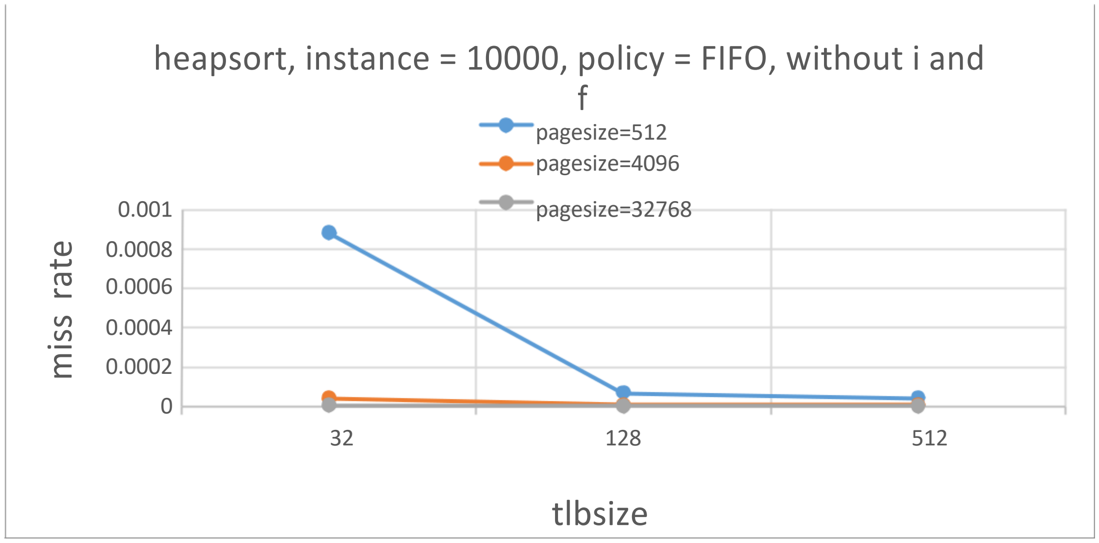
Appendix



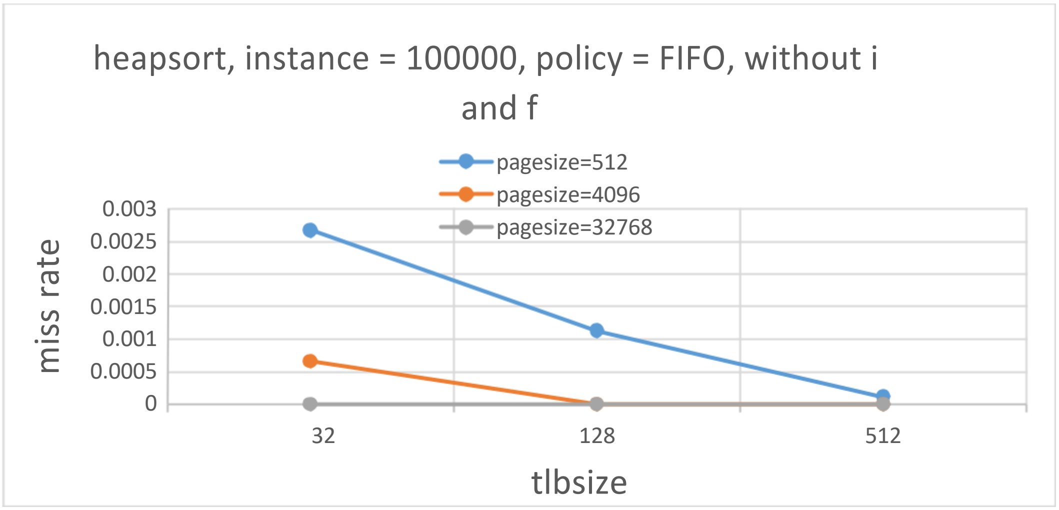
(Figure.1: heapsort, instance = 100, policy = FIFO, without i and f).



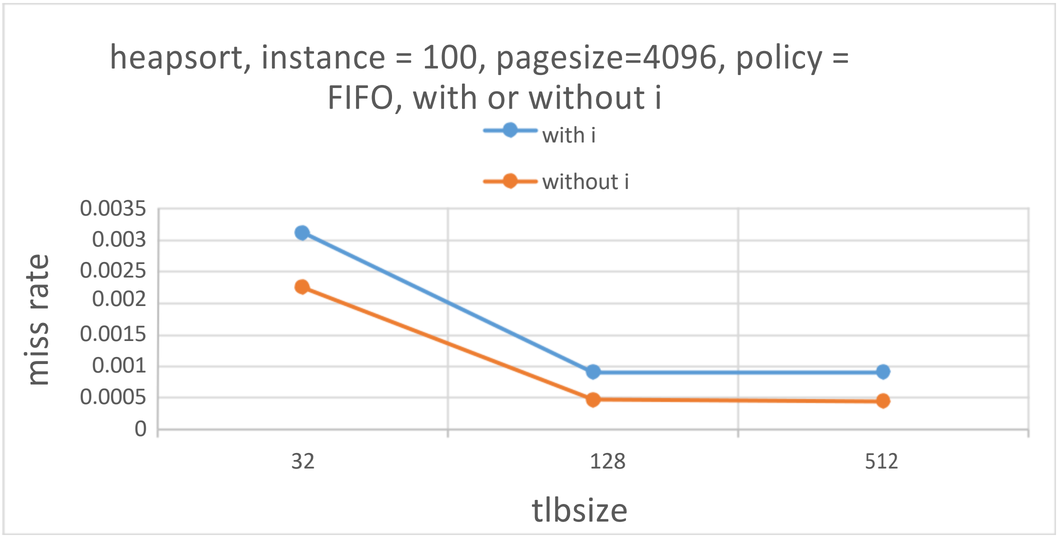
(Figure.2: heapsort, instance = 100, policy = FIFO, without i and f).



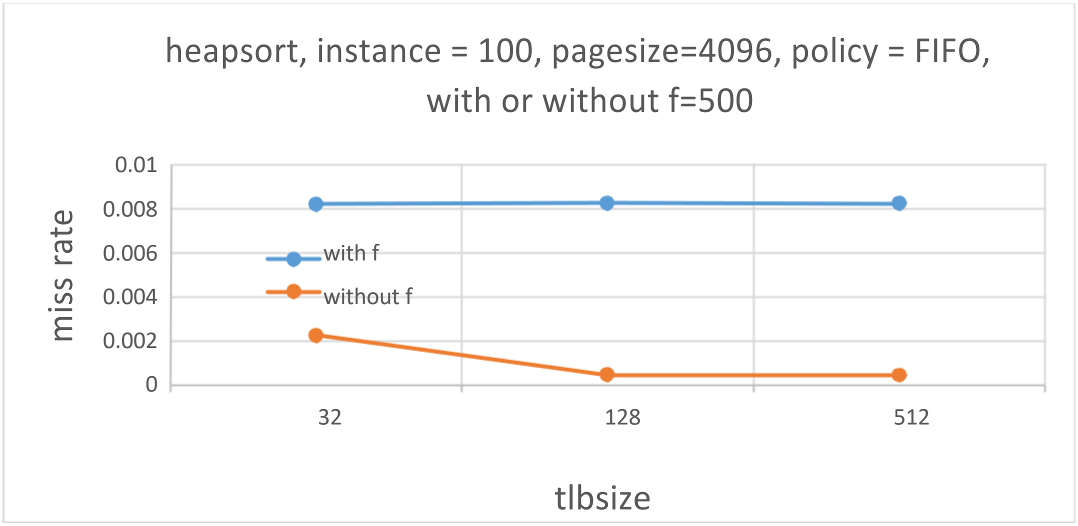
(Figure.3: heapsort, instance = 1000, policy = FIFO, without i and f).



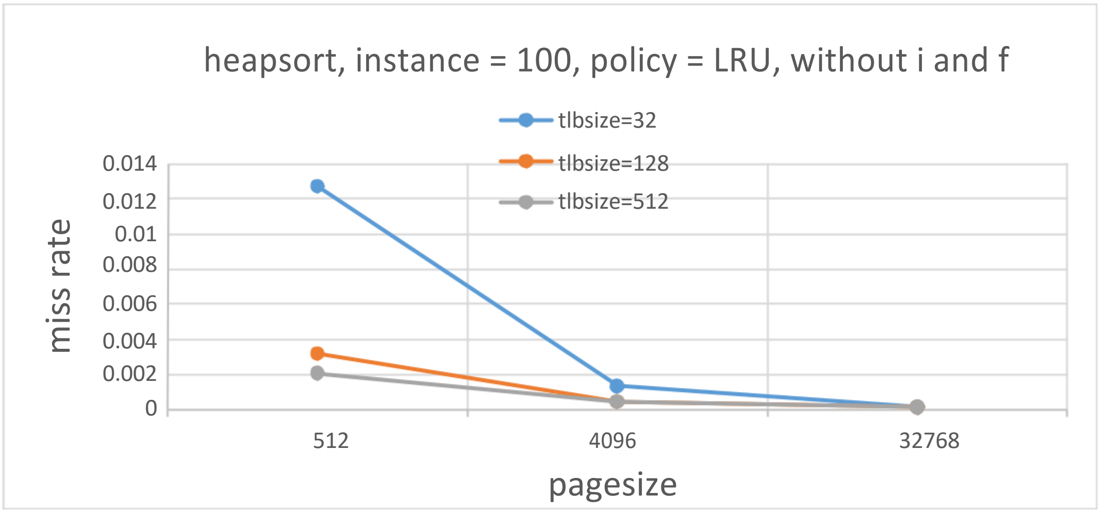
(Figure.4: heapsort, instance = 10000, policy = FIFO, without i and f).



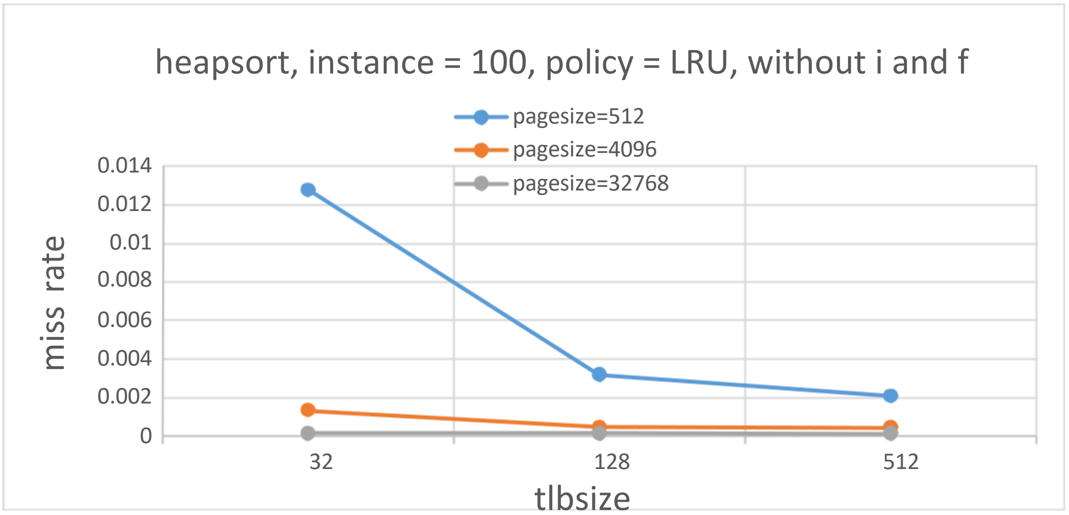
(Figure.5: heapsort, instance = 100, policy = FIFO, pagesize = 4096 with or without i).



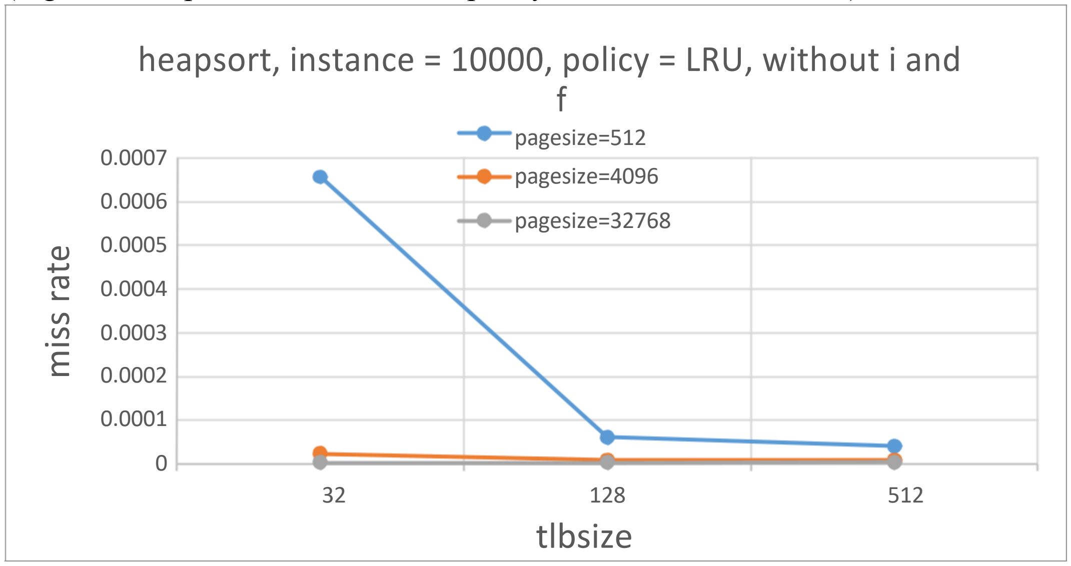
(Figure.6: heapsort, instance = 100, policy = FIFO, pagesize = 4096 with or without f).



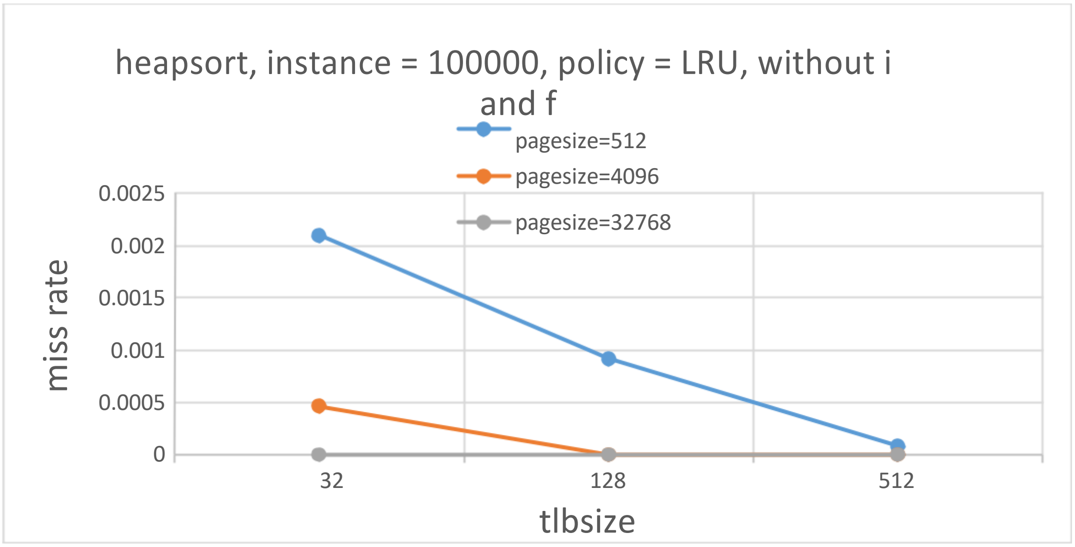
(Figure.7: heapsort, instance = 100, policy = LRU, without i and f).



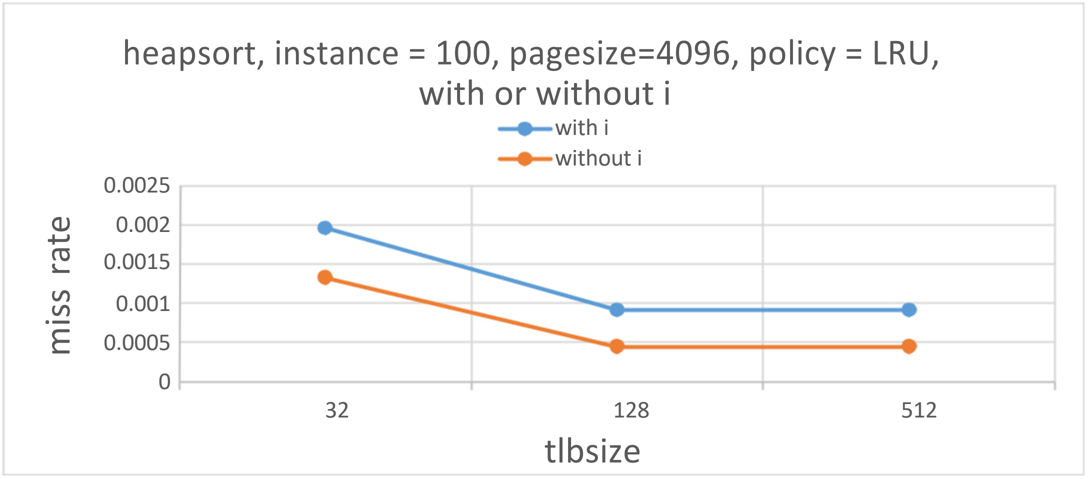
(Figure.8: heapsort, instance = 100, policy = LRU, without i and f).



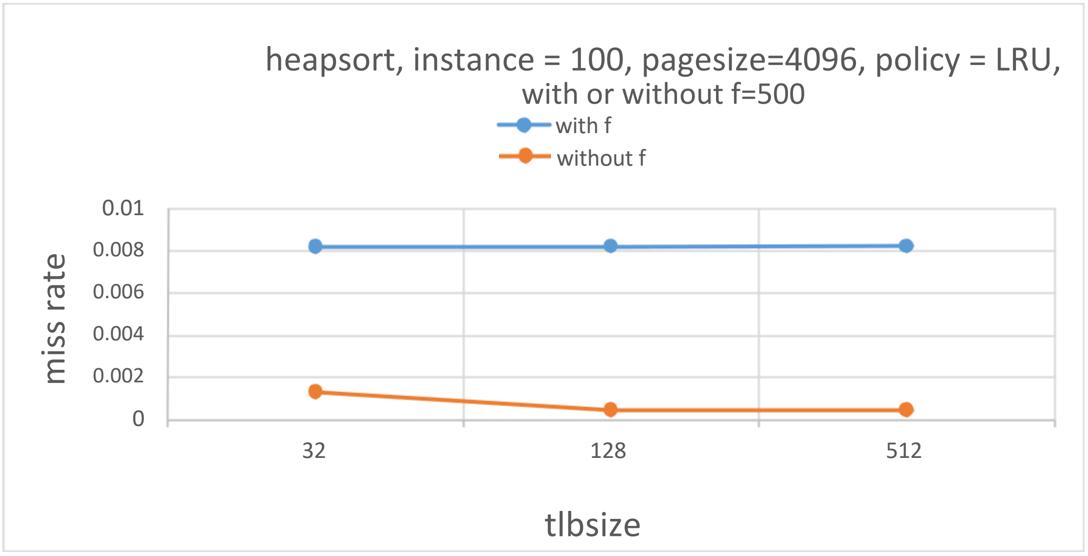
(Figure.9: heapsort, instance = 10000, policy = LRU, without i and f).



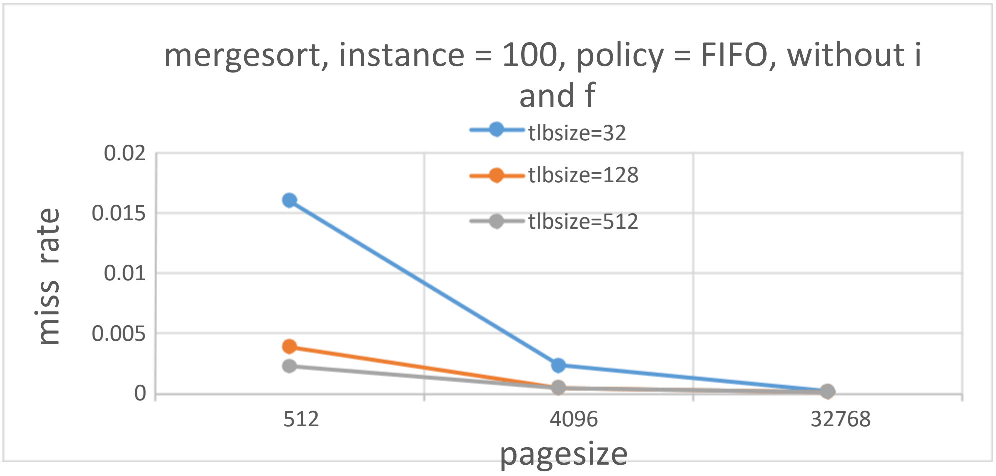
(Figure.10: heapsort, instance = 100000, policy = LRU, without i and f).



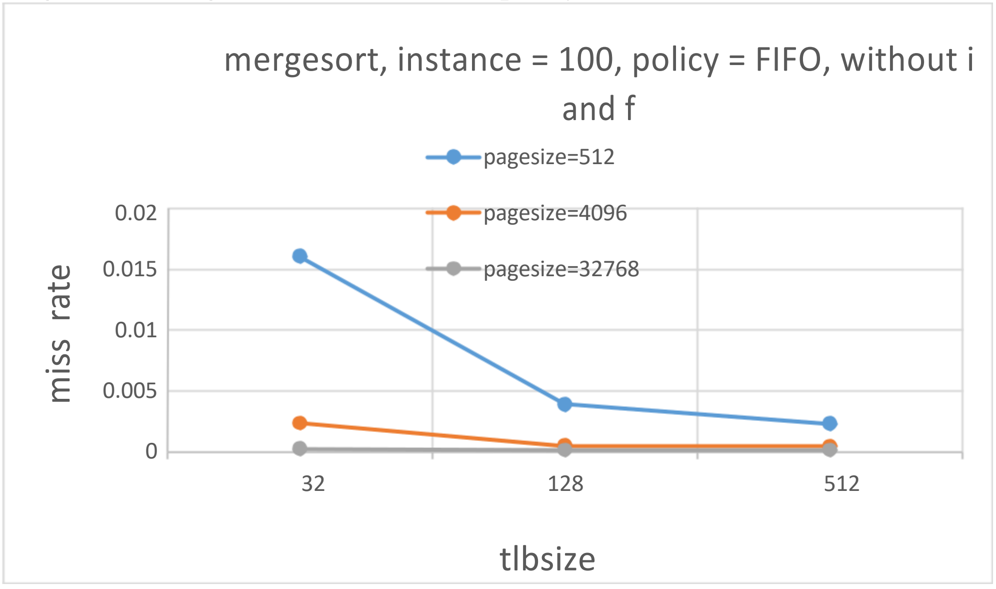
(Figure.11: heapsort, instance = 100, policy = LRU, pagesize = 4096 with or without i).



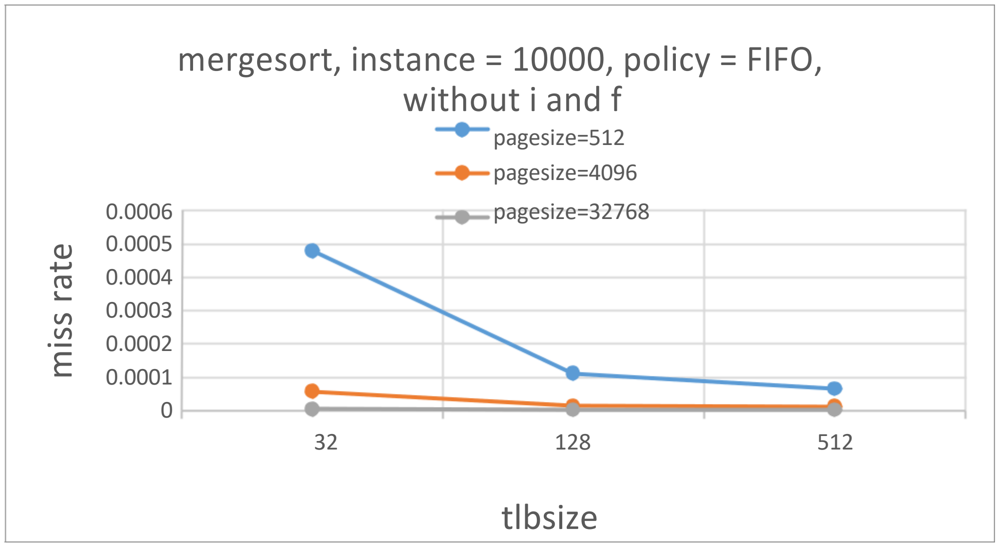
(Figure.12: heapsort, instance = 100, policy = LRU, pagesize = 4096 with or without f).



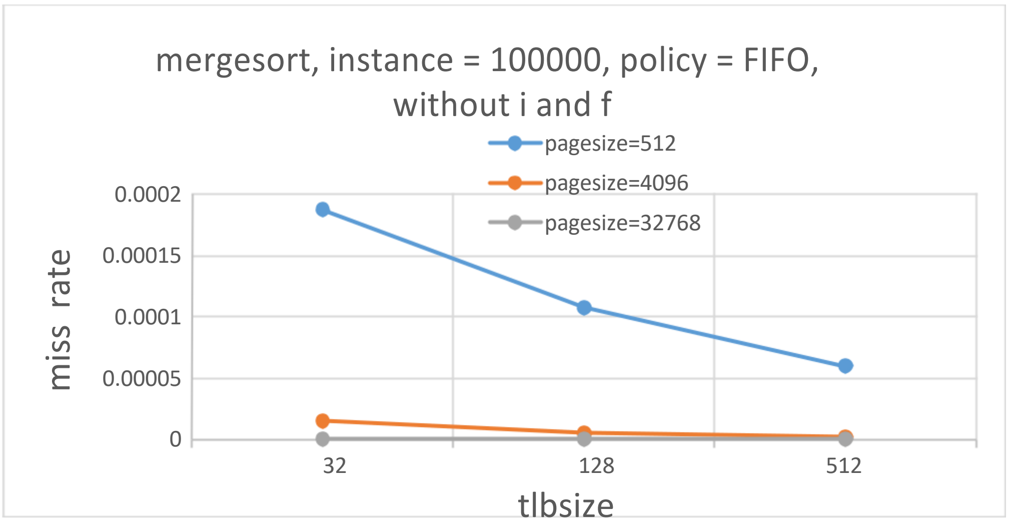
(Figure.13: mergesort, instance = 100, policy = FIFO, without i and f).



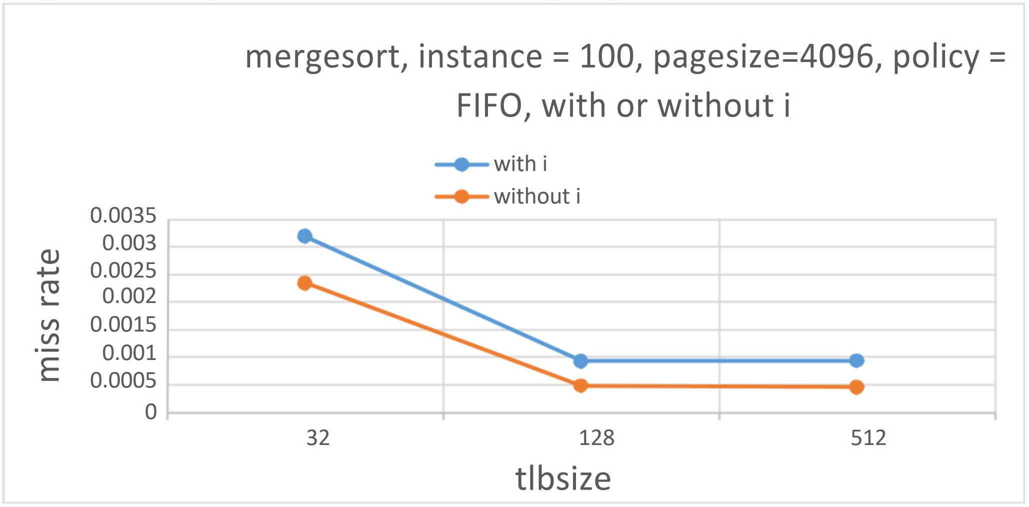
(Figure.14: mergesort, instance = 100, policy = FIFO, without i and f).



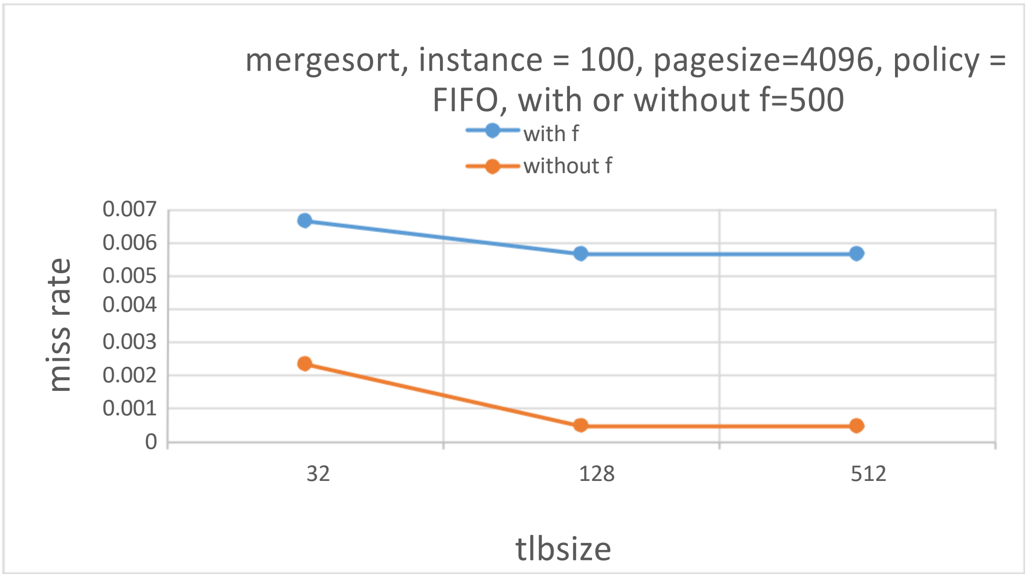
(Figure.15: mergesort, instance = 10000, policy = FIFO, without i and f).



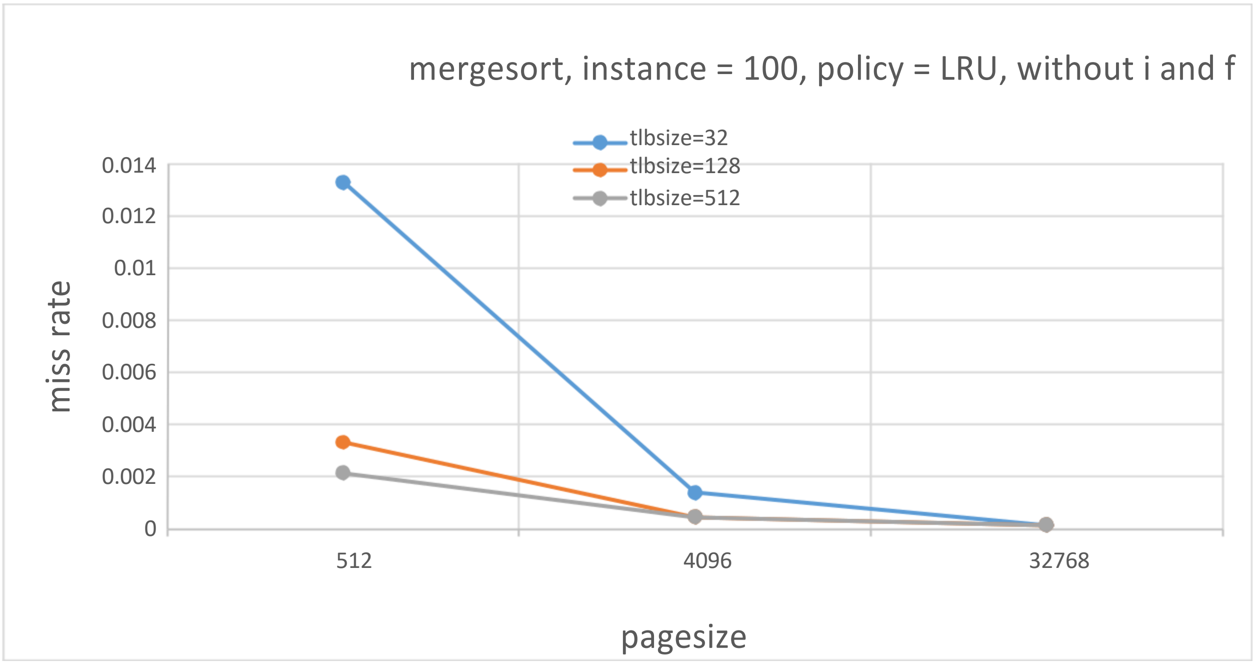
(Figure.16: mergesort, instance = 100000, policy = FIFO, without i and f).



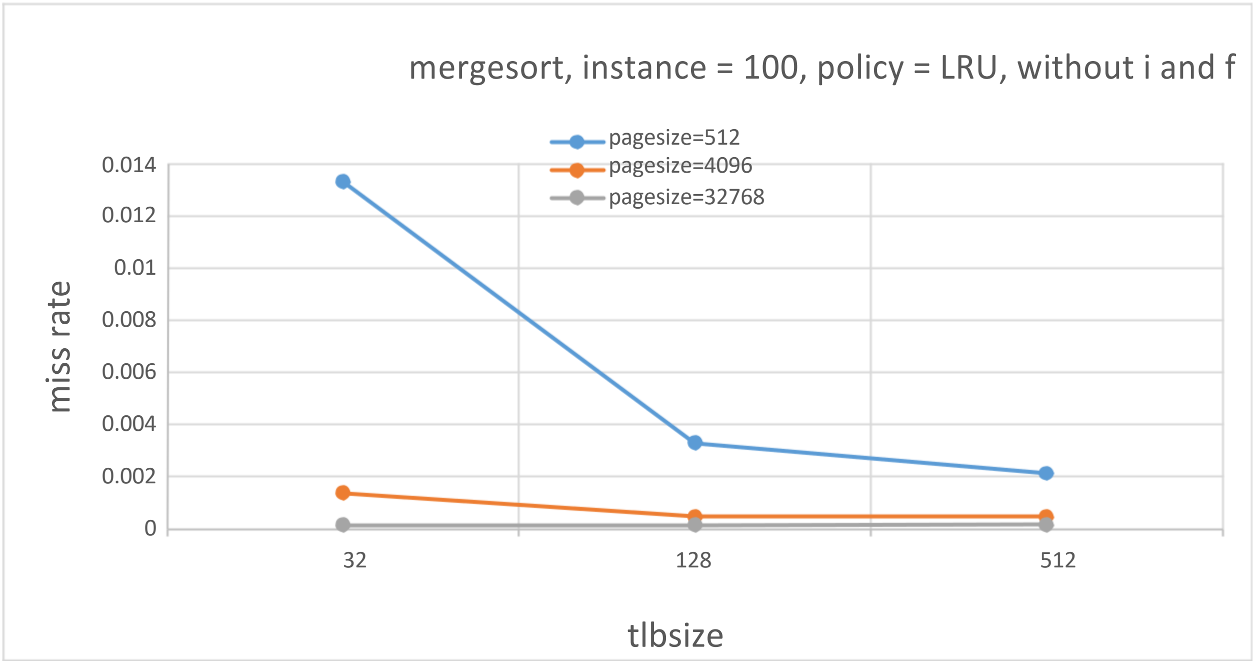
(Figure.17: mergesort, instance = 100, policy = FIFO, pagesize = 4096 with or without i).



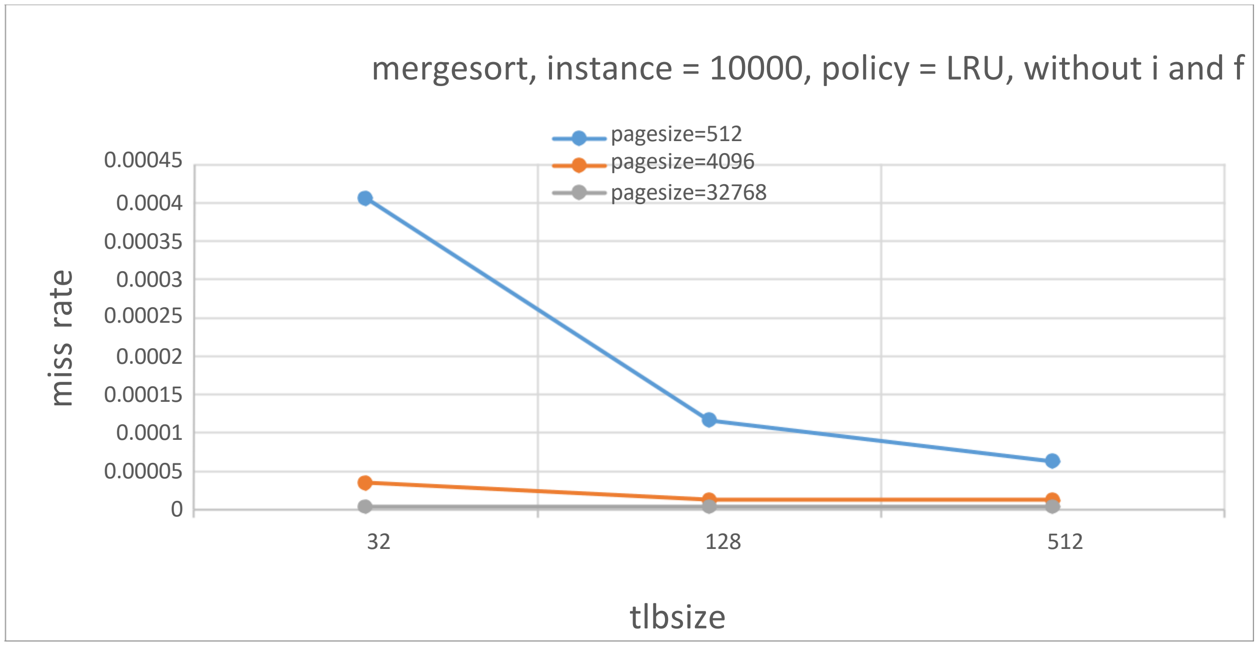
(Figure.18: mergesort, instance = 100, policy = FIFO, pagesize = 4096 with or without f).



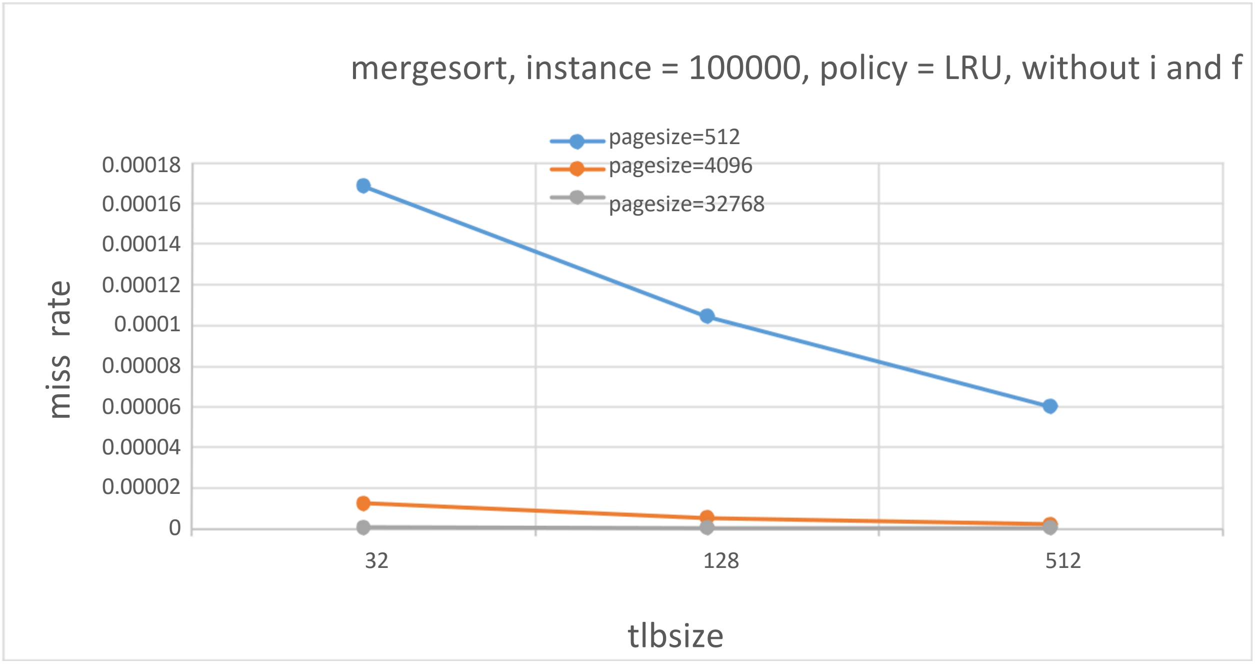
(Figure.19: mergesort, instance = 100, policy = LRU, without i and f).



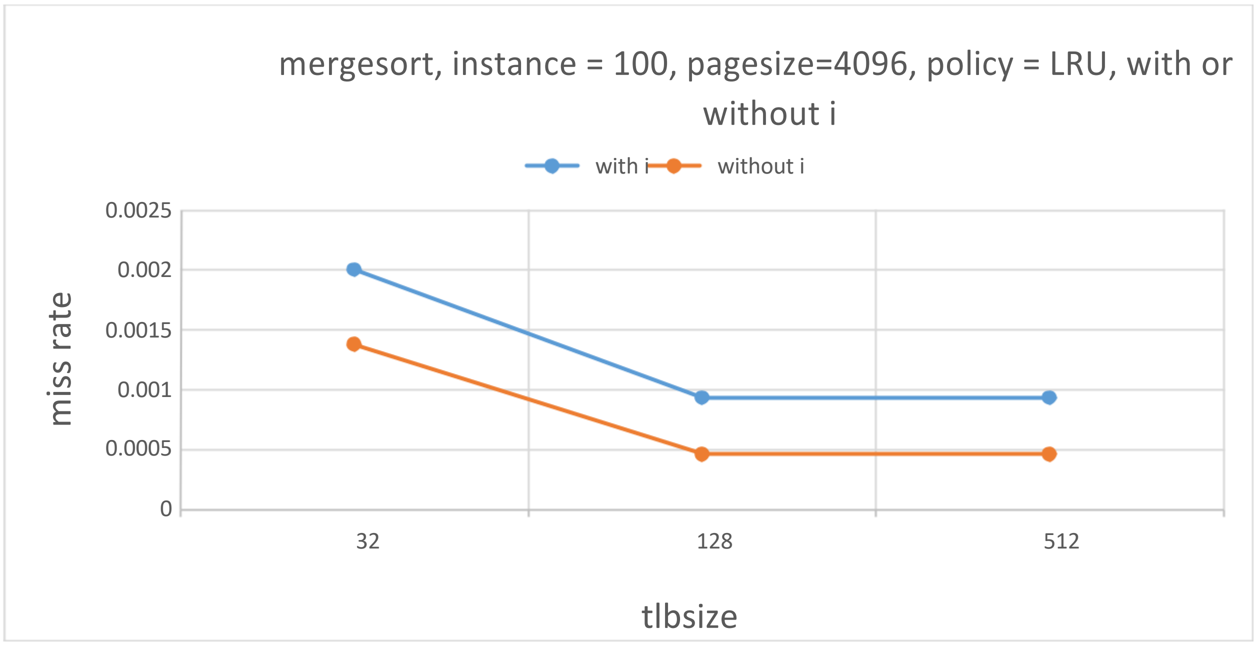
(Figure.20: mergesort, instance = 100, policy = LRU, without i and f).



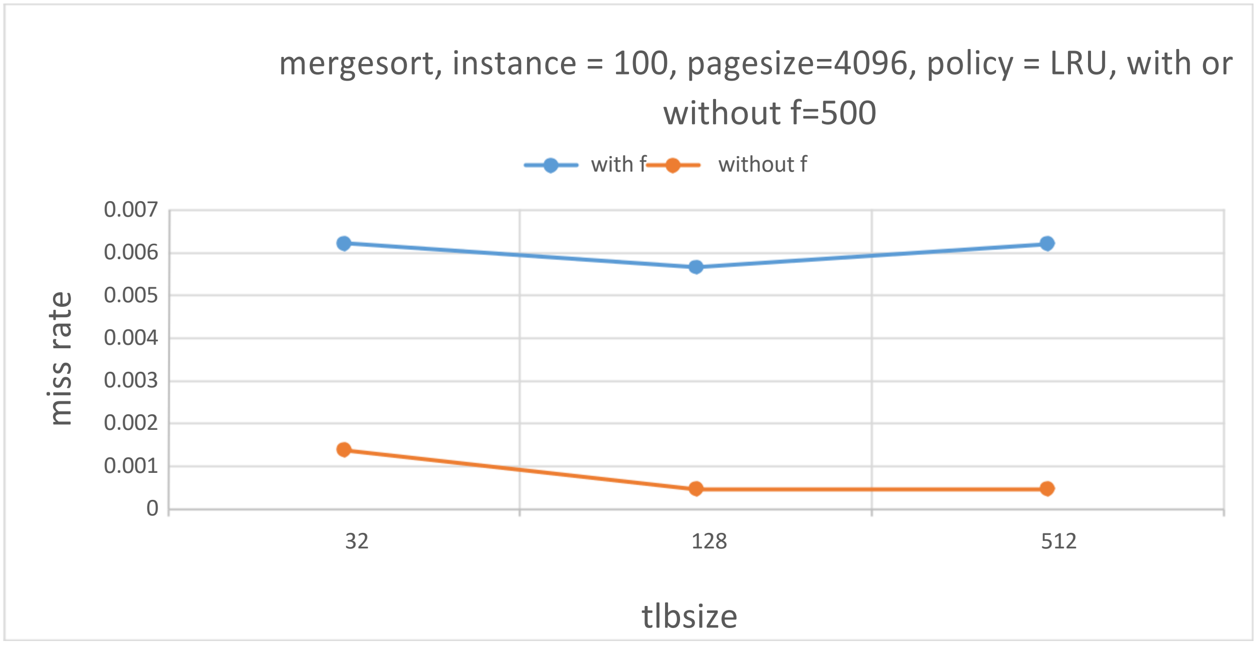
(Figure.21: mergesort, instance = 10000, policy = LRU, without i and f).



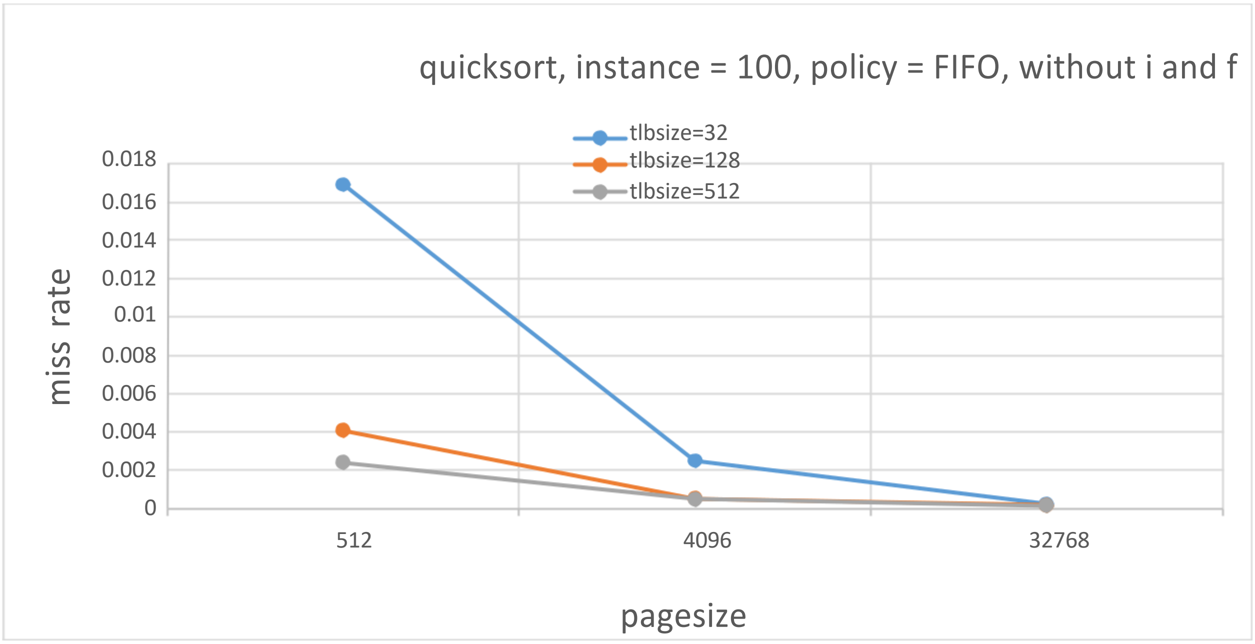
(Figure.22: mergesort, instance = 100000, policy = LRU, without i and f).



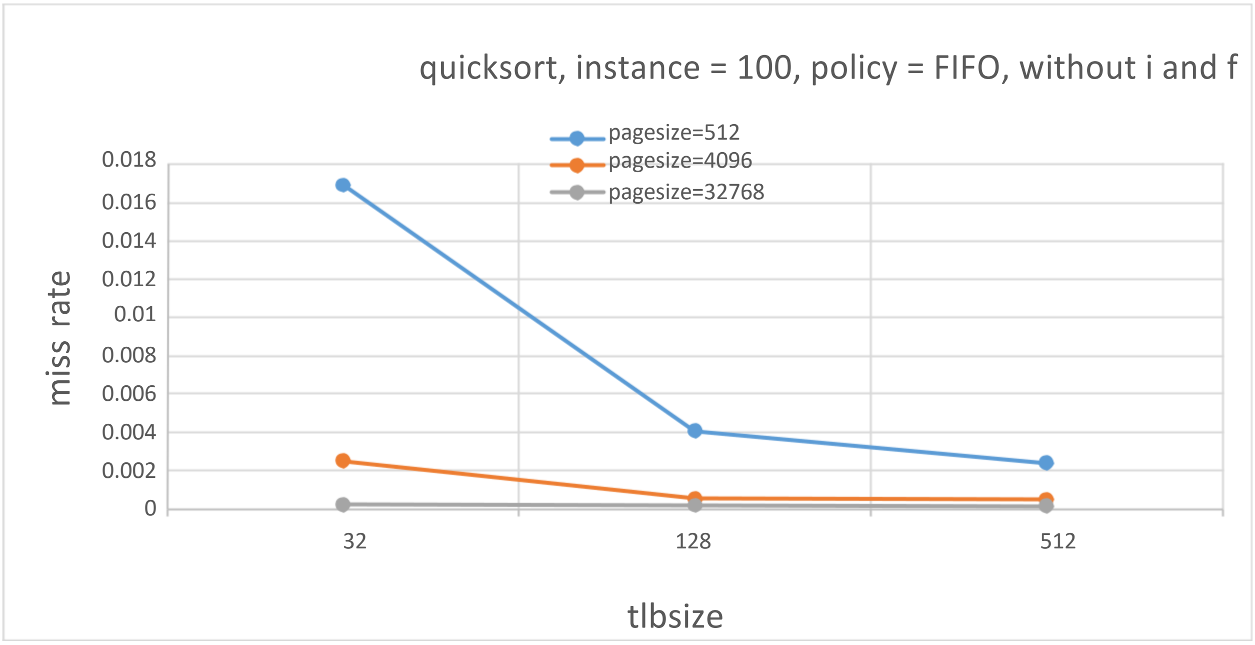
(Figure.23: mergesort, instance = 100, policy = LRU, pagesize = 4096 with or without i).



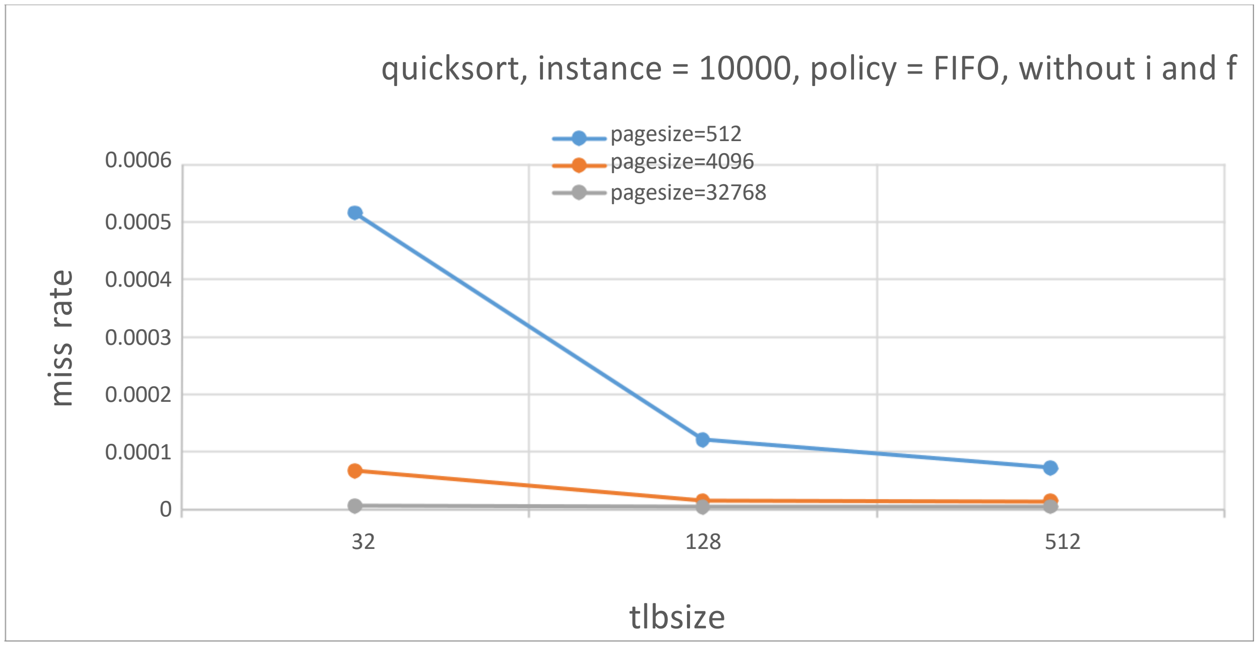
(Figure.24: mergesort, instance = 100, policy = LRU, pagesize = 4096 with or without f).



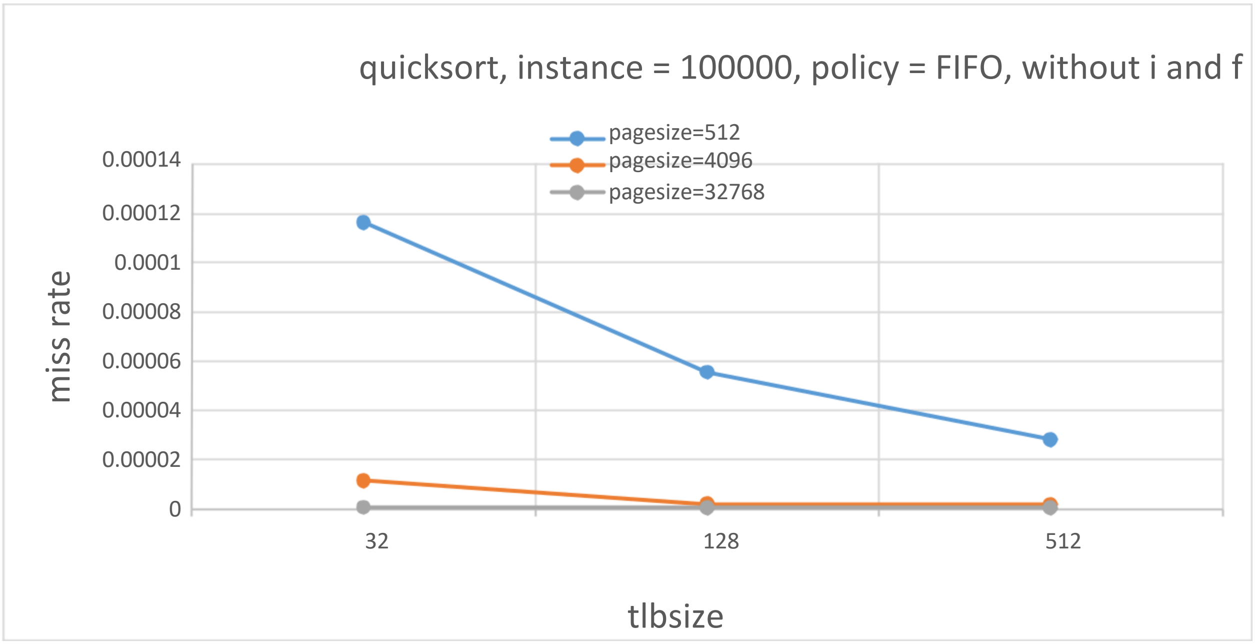
(Figure.25: quicksort, instance = 100, policy = FIFO, without i and f).



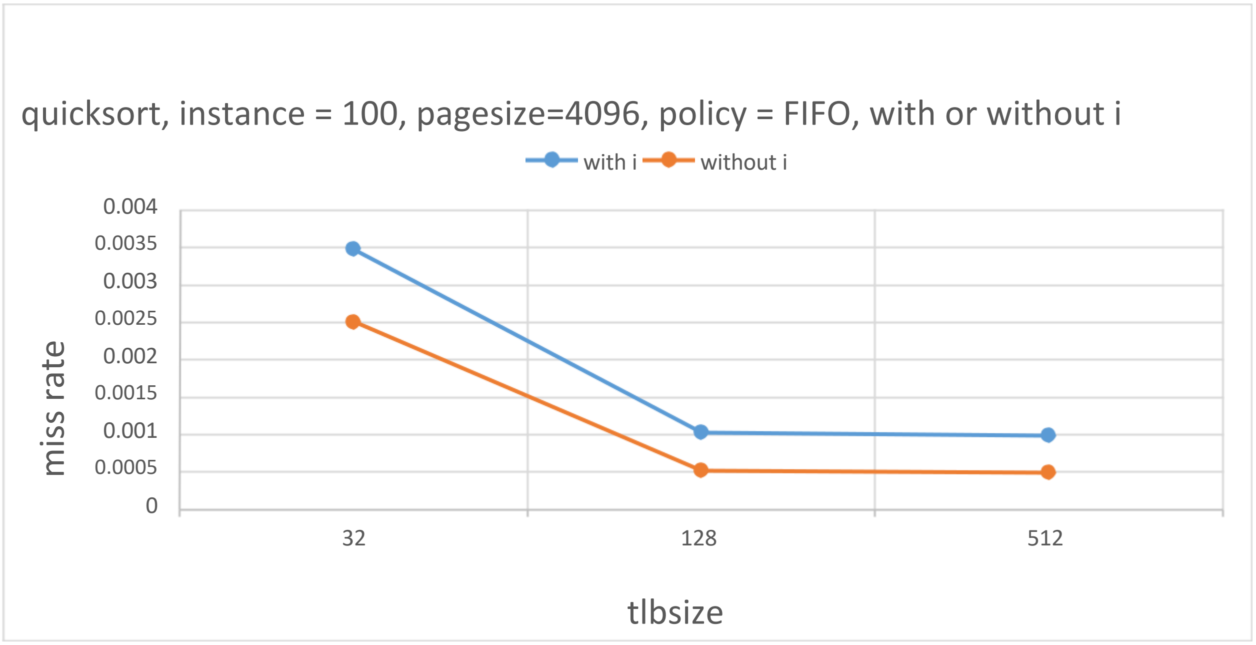
(Figure.26: quicksort, instance = 100, policy = FIFO, without i and f).



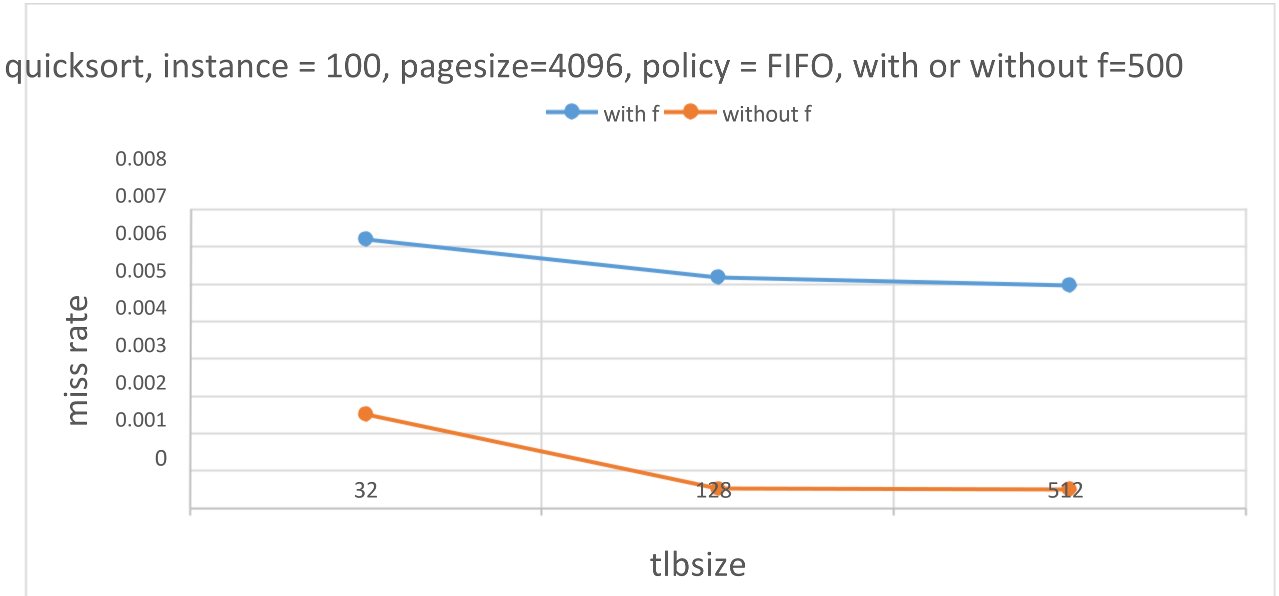
(Figure.27: quicksort, instance = 10000, policy = FIFO, without i and f).



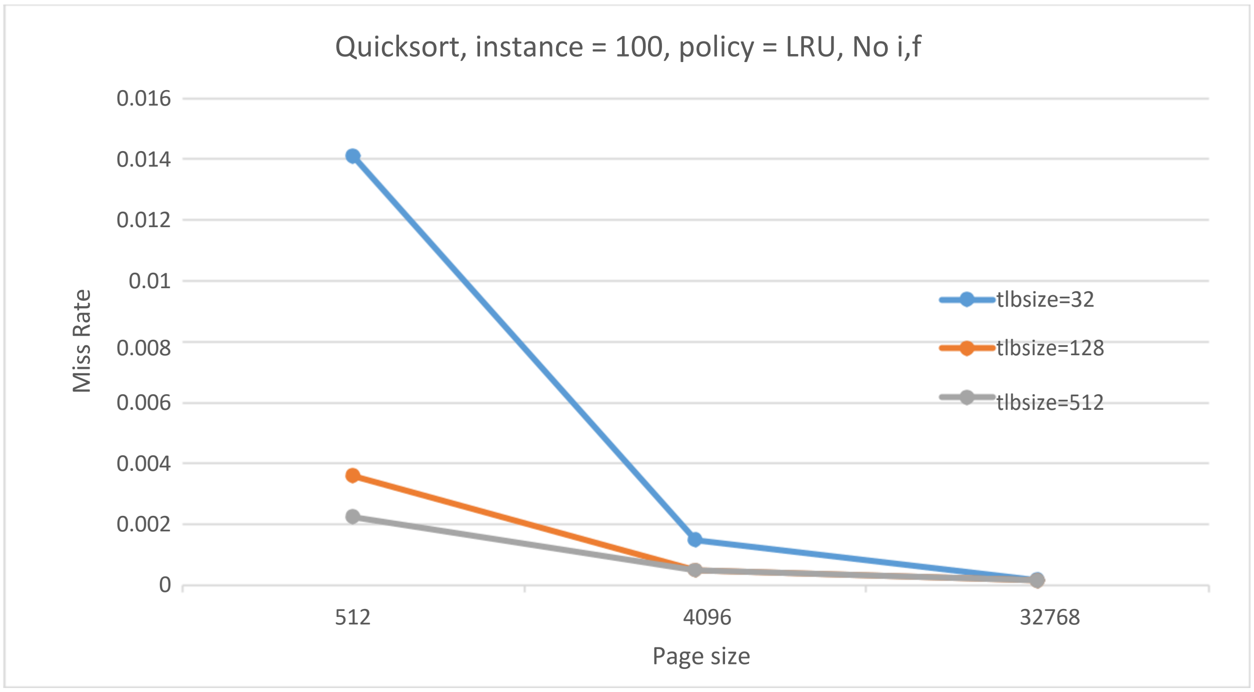
(Figure.28: quicksort, instance = 100000, policy = FIFO, without i and f).



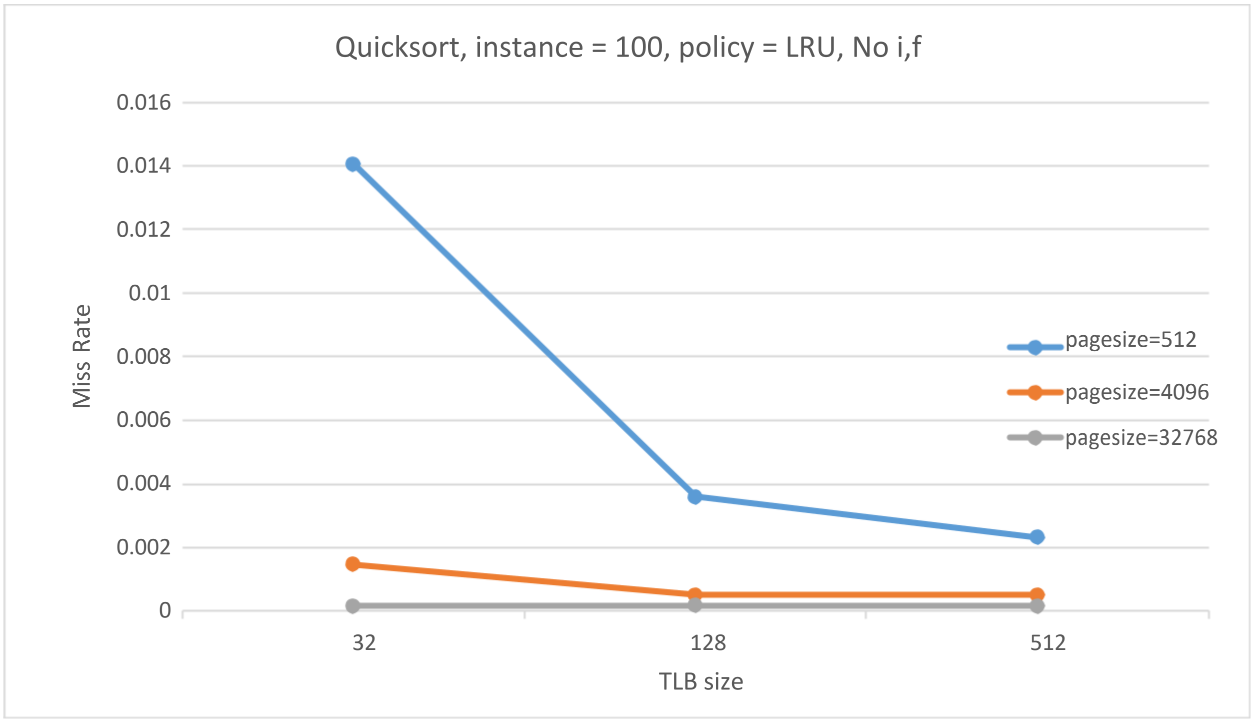
(Figure.29: quicksort, instance = 100, policy = FIFO, pagesize = 4096 with or without i).



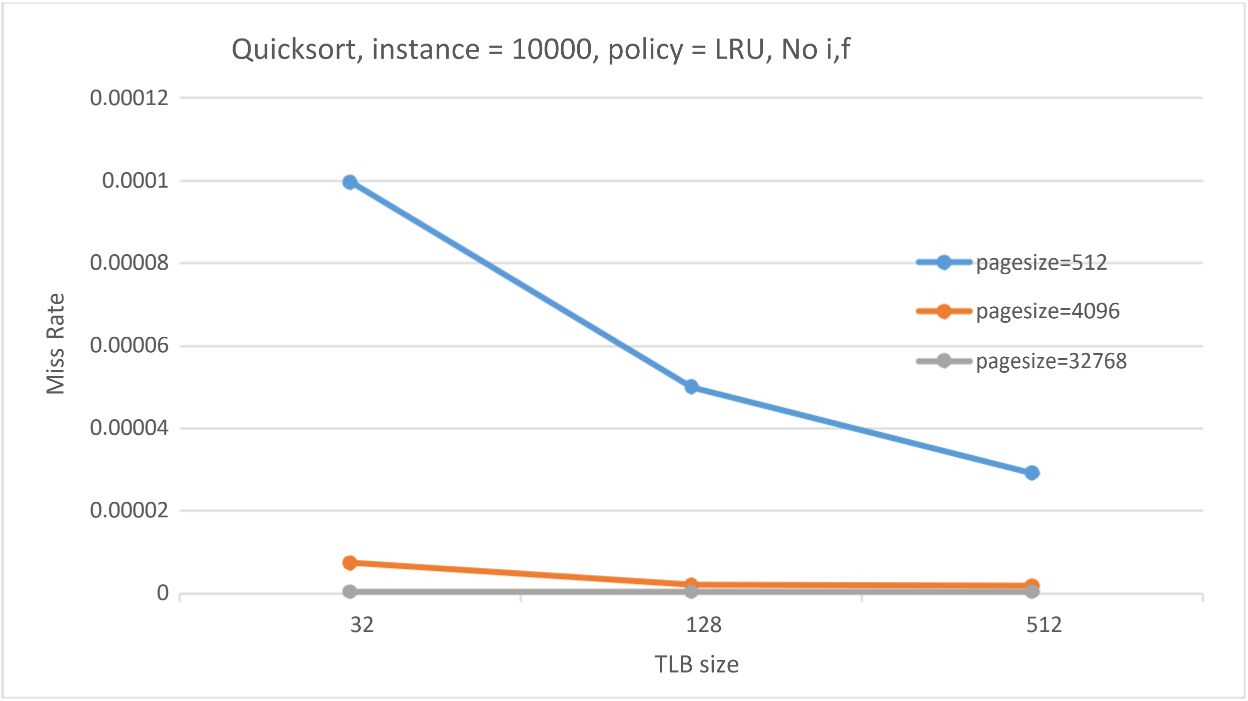
(Figure.30: quicksort, instance = 100, policy = FIFO, pagesize = 4096 with or without f).



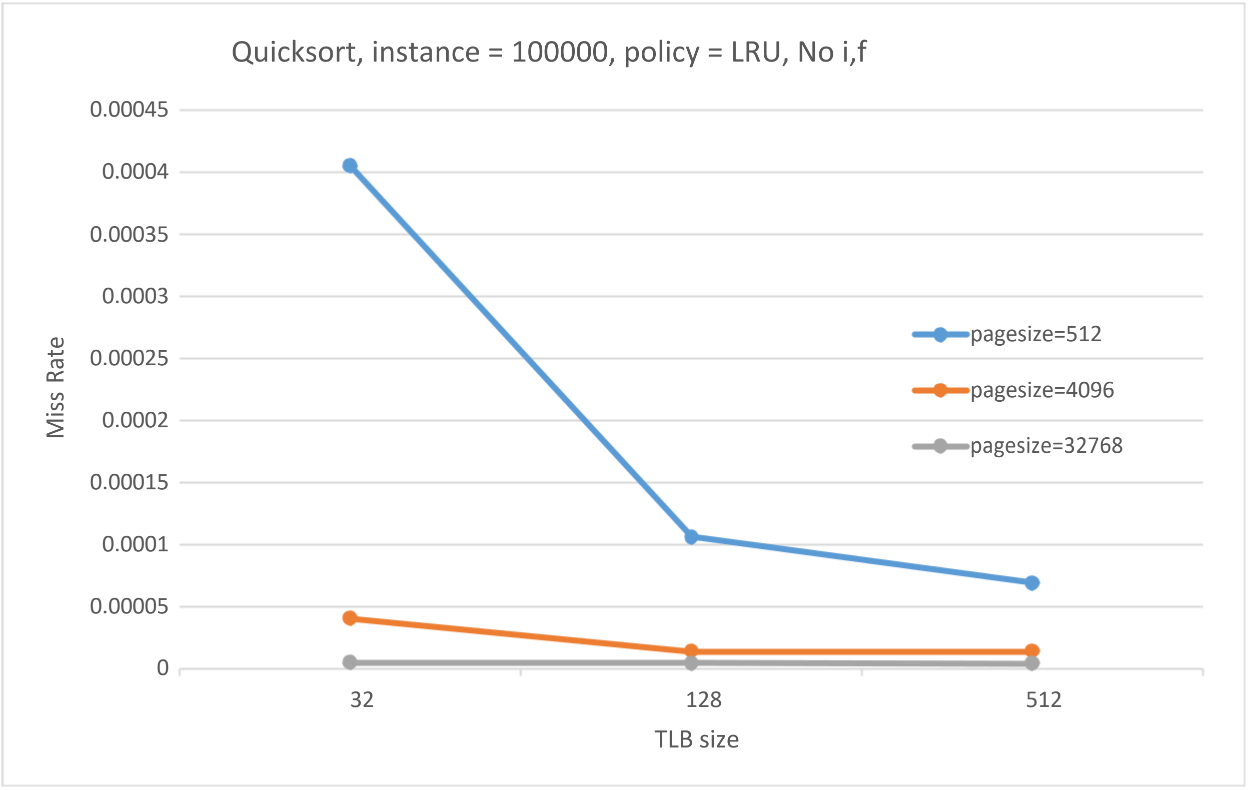
(Figure.31: quicksort, instance = 100, policy = LRU, without i and f).



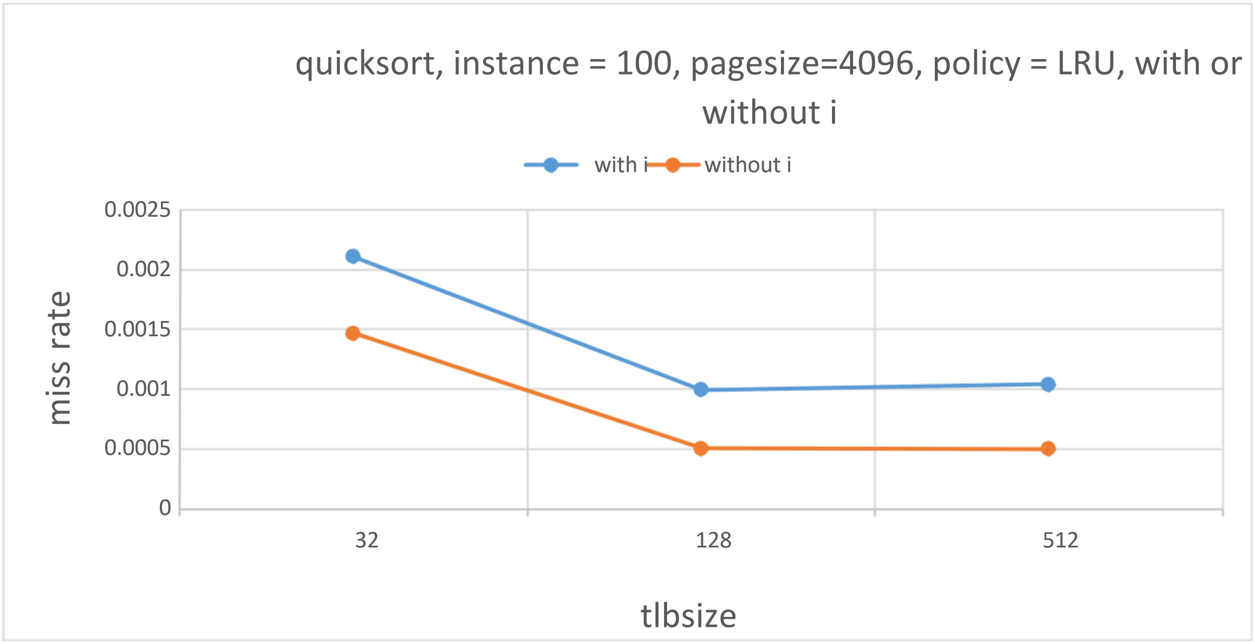
(Figure.32: quicksort, instance = 100, policy = LRU, without i and f).



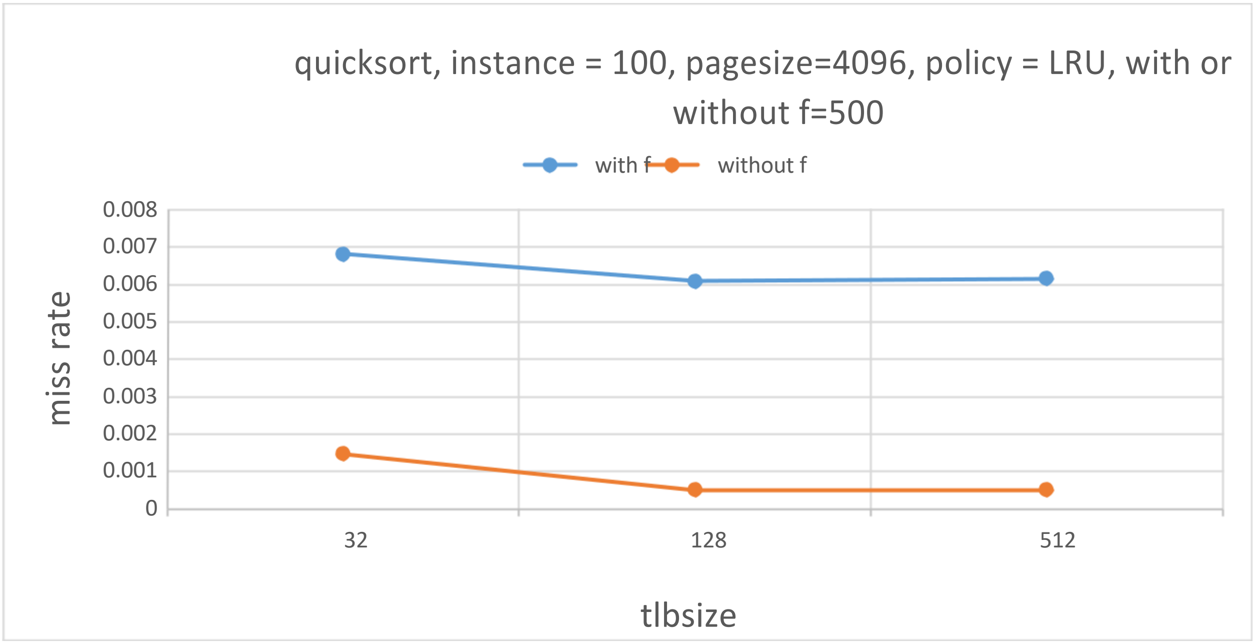
(Figure.33: quicksort, instance = 10000, policy = LRU, without i and f).



(Figure.34: quicksort, instance = 100000, policy = LRU, without i and f).



(Figure.35: quicksort, instance = 100, policy = LRU, pagesize = 4096 with or without i).



(Figure.35: quicksort, instance = 100, policy = LRU, pagesize = 4096 with or without f).