Advanced Computer Architecture

Mahsa Rezaei Firuzkuhi

Problem Description:

The goal of the project is to measure the cache performance parameters using PAPI library present in C. We need to measure the total number of accesses in L1 cache, L2 cache and L3 cache and total number of misprediction and misprediction rate for each level. We also need to compare these parameters with the same parameters extracted using the Valgrind tool for each sorting algorithms. The execution time for the sorting algorithms with different input array sizes of 1024, 8192 and 65536 should be reported.

Solution Strategy:

Instrumented the provided code to use PAPI hardware performance counters to determine the behavior of each sort algorithm separately, I have been using the PAPI - Performance API that is provided on SLURM cluster with certain number of hardware counters offered by the cluster used to compute operations like Cache Misses and various other Performance related measures. This information regarding the available counters and PAPI events that could be used to accomplish the task could be viewed by "PAPI_avail" command. This list also discusses the derivable and available events that current cluster offers. I need to understand the behavior of cache accesses when running different sort algorithms. When the program is executing, we are only interested in the data fetched by the processor as sorting algorithm needs data elements from the array. Therefore, to measure the behavior of cache and better understanding the performance I should only consider data cache misses and total data cache access. This will help in analyzing the behavior of sorting algorithms for input array with different sizes. The PAPI counters used to measure the attributes are listed in table one and they are all available and derived counters.

Metric	Description
PAPI_LD_INS + PAPI_SR_INS	Level 1 total data cache accesses
PAPI_L1_TCM	Level 1 data cache misses
PAPI_L2_TCA	Level 2 total data cache accesses
PAPI_L2_TCM	Level 2 data cache misses
PAPI_L3_TCA	Level 3 total data cache accesses
PAPI_L3_TCM	Level 3 data cache misses

Table 1. Measured metrics used in the implementation.

I am running each sorting function three times in the program. First, I measure the L1 total cache accesses and execution time. Second, I reset the resultant input array and count the total L2 cache accesses and total number of miss predictions. Also, I count the total number of L1 cache misses and total accesses in L3 in the second run. Finally, in the third run, I counted the total number of L3 cache misses. I repeated these three steps for each sort algorithm. As a result, I had defined three separate event sets at the beginning and added the counters to them using the

"PAPI_add_event" function. Then I have defined an overflow handler for each of the PAPI metrics in my program using the function "PAPI_overflow" and used the try and error technique to find the best threshold for the event overflow function. To find an appropriate size for the threshold, I have run the program several times and extract the number of overflows had encountered on each metrics. Table two shows a few of the threshold size variation results for different input sizes. I have selected the threshold value of 1000000000 as a trade of between input size and number of counter overflows.

THRESHOLD	1000000			1E+09			1E+10			1E+11		
Input Size	1024	8192	65536	1024	8192	65536	1024	8192	65536	1024	8192	65536
PAPI_LD_INS	0	0	0	0	0	0	0	0	0	0	0	0
PAPI_SR_INS	7	506	32286	0	0	31	0	0	22	0	26	0
PAPI_L1_TCM	0	0	0	0	0	0	0	0	0	0	0	0
PAPI_L2_TCM	0	0	0	0	0	0	0	0	0	0	0	0
PAPI_L2_TCA	0	0	0	0	0	0	0	0	0	0	0	0
PAPI_L3_TCM	0	0	0	0	0	0	0	0	0	0	0	0
PAPI_L3_TCA	0	0	0	0	0	0	0	0	0	0	0	0

Table 2. Number of overflows on each metric for different threshold value.

We can easily deduce the miss rate from above information. Miss rates can be calculated by dividing the number of cache misses by the number of cache accesses for each cache level.

Results

Description of Resources:

We are using the slurm cluster for conducting this experiment. The hardware information that was used when the experiment was conducted are as follows:

PAPI version: 5.7.0.0 Operating system: Linux 4.4.138-59-default Vendor string and code: Genuine Intel (1, 0x1) Model string and code: Intel(R) Xeon(R) CPU E5-2665 0 @ 2.40GHz (45, 0x2d) CPU revision: 7.000000 CPUID: Family/Model/Stepping 6/45/7, 0x06/0x2d/0x07 CPU Max MHz: 2400 CPU Min MHz: 1200 Total cores: 32 SMT threads per core: 2 Cores per socket: 8	Sockets: 2 Cores per NUMA region: 16 NUMA regions: 2 Running in a VM: no Number Hardware Counters: 10 Max Multiplex Counters: 384 Virtualization: VT-x L1d cache: 32K L1i cache: 32K L2 cache: 256K L3 cache: 20480K NUMA node0 CPU(s): 0-7,16-23 NUMA node1 CPU(s): 8-15,24-31

Table 3. Hardware Configuration

Measurements

For all the statistic measurements in this project, I have run the program three times and have reported the average of those information in all the tables and plots.

Sort	Exec Time	L1_Misses	L1_Accesses	L2_Misses	L2_Accesses	L3_Misses	L3_Accesses	L1_Miss Rate	L2_Miss Rate	L3_Miss Rate
QSORT	105	640	149448	249	696	0	249	0	36	0
HEAPSORT	115	298	163846	25	328	0	25	0	8	1
SHELLSORT	127	293	300532	19	321	0	19	0	6	0
INSORT	3337	371	7876670	59	405	0	59	0	14	0
QUICKSORT	69	304	163172	26	333	0	26	0	8	0

Table 4. Average of PAPI events report (on three different run) for different sort algorithms with input array size 65536

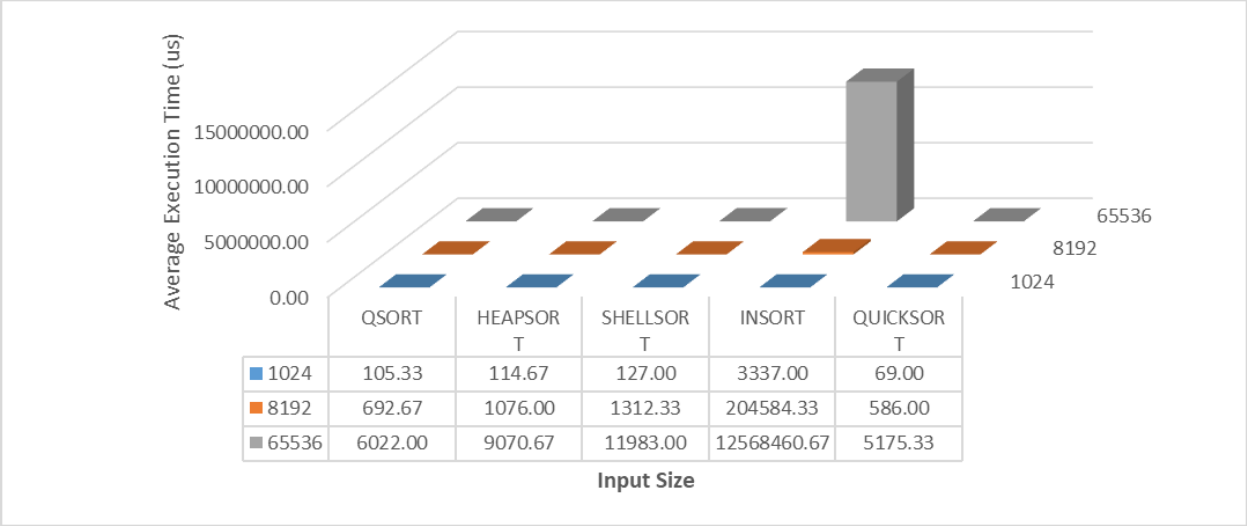
Sort	Exec Time	L1_Misses	L1_Accesses	L2_Misses	L2_Accesses	L3_Misses	L3_Accesses	L1_MissRate	L2_MissRate	L3_MissRate
QSORT	692	19237	1448292	1978	19293	0.0	1978.7	1.3	10.3	0.0
HEAP	1076	9201	1655364	140	9230	0.0	140.3	0.6	1.5	0.0
SHEL	1312	22038	3175034	171	22068	0.0	171.7	0.7	0.8	0.0
INSORT	204584	5549993	503418570	743	5550068	21.7	743.3	1.1	0.0	2.1
QUICK	586	5809	1612748	150	5836	1.0	150.7	0.4	2.6	0.6

Table 5. Average of PAPI events report (on three different run) for different sort algorithms with input array size 65536

Sort	Exec Time	L1_Misses	L1_Accesses	L2_Misses	L2_Accesses	L3_Misses	L3_Accesses	L1_MissRate	L2_MissRate	L3_MissRate
QSORT	6022	286393	13405112	98096	286493	0.3	98096	2.1	34.2	0.0
HEAP	9071	102789	15989949	13808	102833	0.0	13808	0.6	13.4	0.0
SHELL	11983	265383	31589434	8225	265425	0.0	8225	0.8	3.1	0.0
INSOR	12568461	401988992	32213480084	14931480	401991084	5.3	14931480	1.2	3.7	0.0
QUICK	5175	101331	15273419	4015	101374	0.0	4015	0.7	4.0	0.0

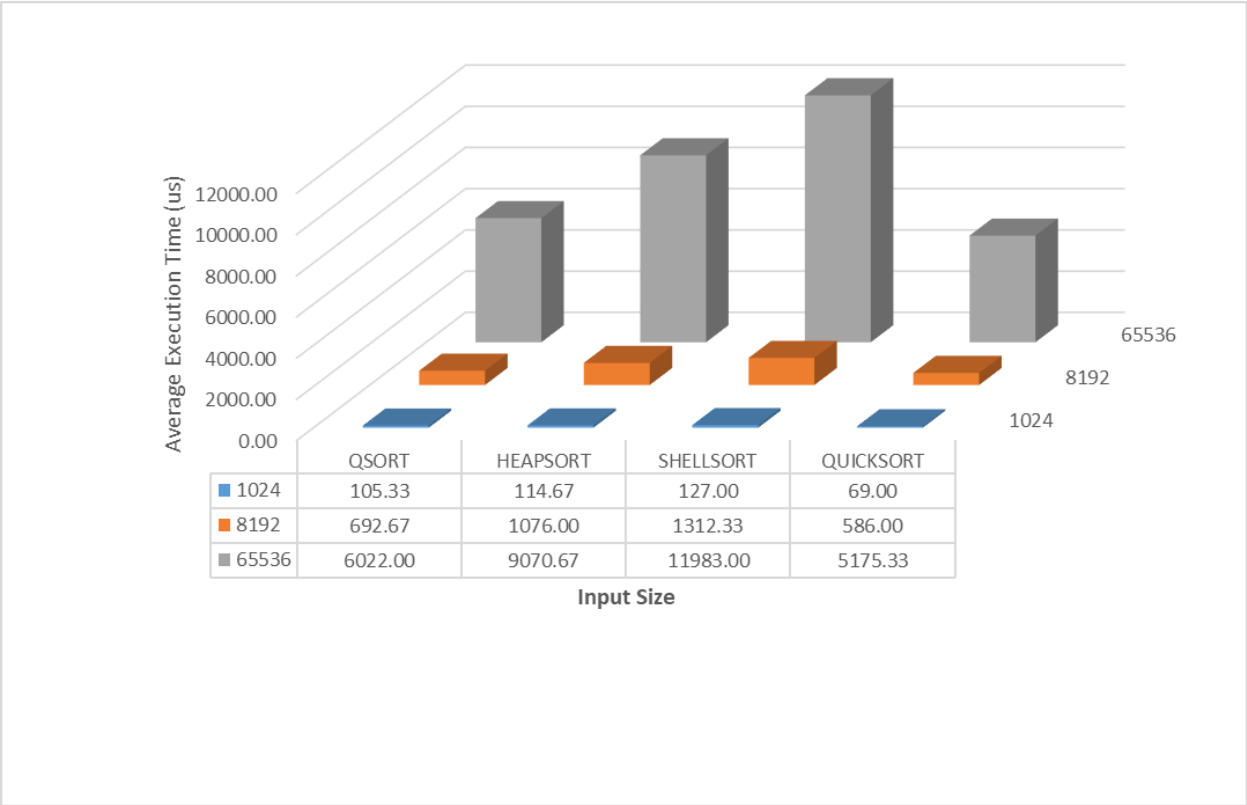
Table 6. Average of PAPI events report (on three different run) for different sort algorithms with input array size 65536

Execution Time:



	QSORT	HEAPSORT	SHELLSORT	INSORT	QUICKSORT
1024	105.33	114.67	127.00	3337.00	69.00
8192	692.67	1076.00	1312.33	204584.33	586.00
65536	6022.00	9070.67	11983.00	12568460.67	5175.33

Fig. 1. Averaged processing time comparison for the five sorting algorithms with different input array sizes.



	QSORT	HEAPSORT	SHELLSORT	QUICKSORT
1024	105.33	114.67	127.00	69.00
8192	692.67	1076.00	1312.33	586.00
65536	6022.00	9070.67	11983.00	5175.33

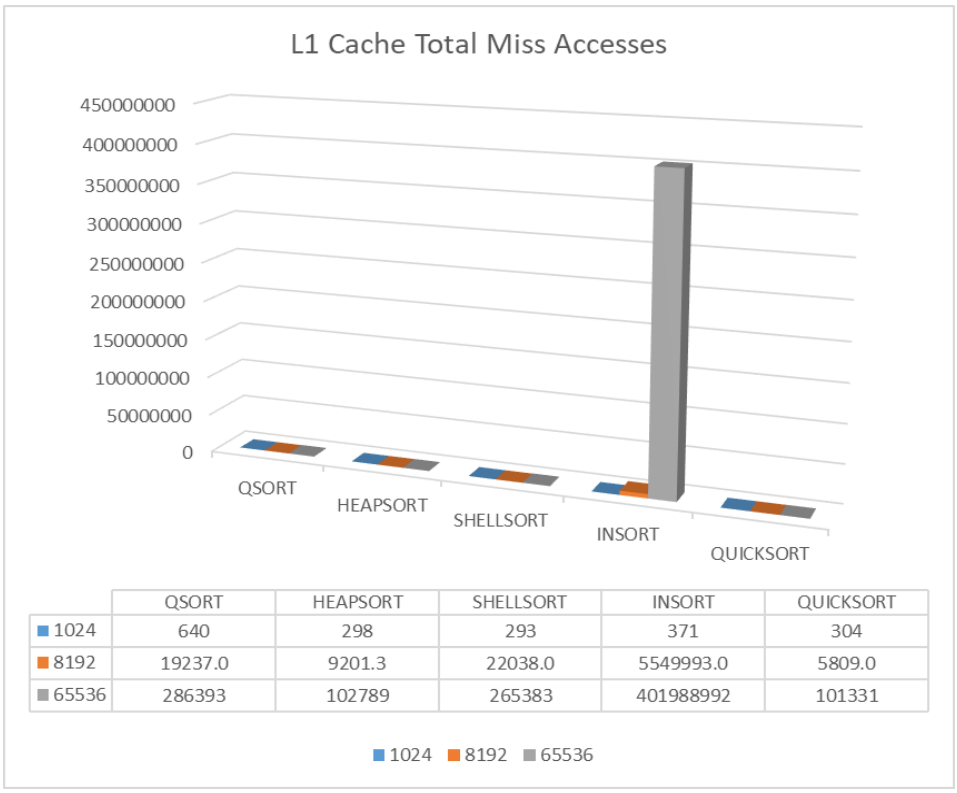
Fig. 2. Averaged processing time comparison for the four sorting algorithms (removed INSORT algorithm information) with different input array sizes.

I have compared execution time in term of processing time for different input array sizes with different sorting algorithms. Figure 1 shows that using the Insort algorithm increases processing time in average 100 times compared to the other sorting implementation, while the input array has the maximum value which is 65536. On the other hand, for input sizes 1024 and 8192, I observe that the increase in the execution time of the Insort is in the order of 10 times of the rest of the sorting algorithms. As a result, I conclude that the execution time of the Insort algorithm will increase gradually with the input array size.

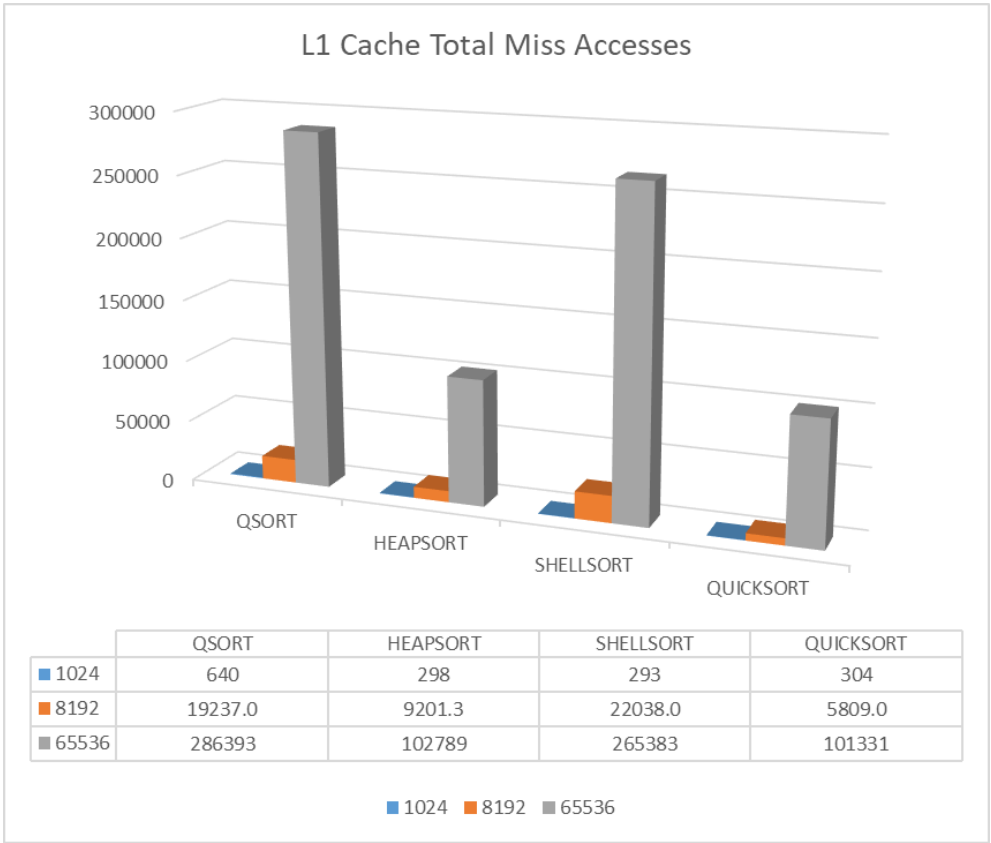
Moreover, to be able to compare the rest of the four algorithms execution times in more details, I have removed the Insort information from the bar chart. From figure 2, I can conclude that the execution time of the Qsort and Quicksort are very close to each other and in both cases, the increase in the input size results the increase in execution time. But still both algorithms have the minimum execution time. I can also conclude that after the Insort, the Shell sort and Heapsort in order, have the largest processing time and both increase the execution time gradually with the increase in input size. As a result, increasing the input size has the most effect on the Insort and Shell sort. The Quicksort, as I expected, takes the least execution time and Insort takes the highest execution time for any input size.

Cache Misses:

I have compared number of cache misses for different input sizes with different sort algorithms. Figure 3 show the results for the total cache misses for L1 cache. I see similar pattern where the number of misses increases significantly by increasing input size for the Insort algorithm. The first plot in Figure 3 shows that using the Insort algorithm increases number of cache misses in average 1000 times compared to the other sorting implementation, while the input array has the maximum value which is 65536. Like the execution time analysis, to be able to compare the rest of the four algorithms in more details, I have removed the Insort information from the bar chart. From the second plot in figure 3, I can conclude that the Heapsort and Quicksort both have similar number of the cache miss accesses while they have the least number of misses with any input size. On the other hand, the two algorithms Shell sort and Qsort have similar number of cache misses while both have the largest number of cache misses between four algorithms. Quicksort still has the least number of cache misses for all input sizes. The interesting observation is in the heapsort cache misses when comparing figure 2 and 3. Heapsort has least number of misses in the maximum input size (in the level of Quicksort) while it had the highest execution time in maximum input size.



	QSORT	HEAPSORT	SHELLSORT	INSORT	QUICKSORT
1024	640	298	293	371	304
8192	19237.0	9201.3	22038.0	5549993.0	5809.0
65536	286393	102789	265383	401988992	101331



	QSORT	HEAPSORT	SHELLSORT	QUICKSORT
1024	640	298	293	304
8192	19237.0	9201.3	22038.0	5809.0
65536	286393	102789	265383	101331

Fig. 3. L1 total cache miss comparison for different sorting algorithms with different input sizes.

Total Cache Access:

We compare the total number of cache accesses for different input sizes with different sort algorithms. Figure 4 show the results for the total cache access of L1 cache. I see similar pattern where the number of L1 cache accesses increases significantly by increasing input size for the Insort algorithm. The upper plot in Figure 3 shows that using the Insort algorithm increases number of cache accesses in average 1000 times compared to the other sorting implementation, while the input array has the maximum value which is 65536. Like the execution time analysis, to be able to compare the rest of the four algorithms in more details, I have removed the Insort information from the bar chart. From the second plot in figure 4, I can conclude that the Heapsort and Quicksort both have similar number of cache accesses while they have the second least number of accesses after Qsort (which has the least number of total accesses) with any input size. On the other hand, the algorithms Qsort have the least number of cache accesses while it has the largest number of cache misses between four algorithms. So, we expect that the Qsort to have the highest miss rate in comparison to other algorithms. We can see this fact in the figure 9. where I compared the miss rates. Regarding the total cache accesses in L2 and L3, based on the figures 6, 7 and 8, I see similar pattern where the number of L2 and L3 cache accesses and misses increases significantly by increasing input size for the Insort algorithm. So, I have removed that information from bar charts.

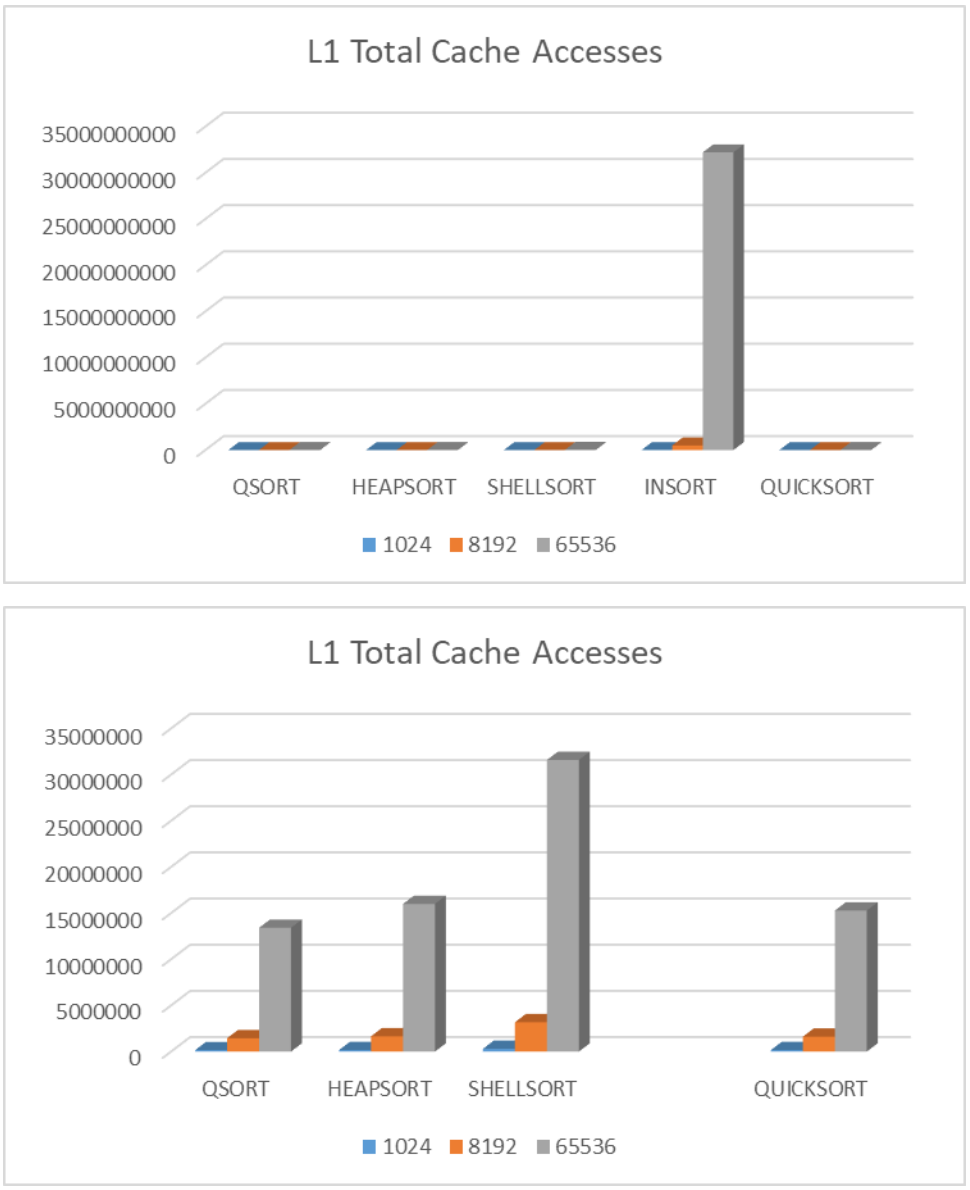


Fig. 4. L1 total cache access comparison for different sorting algorithms with different input sizes.

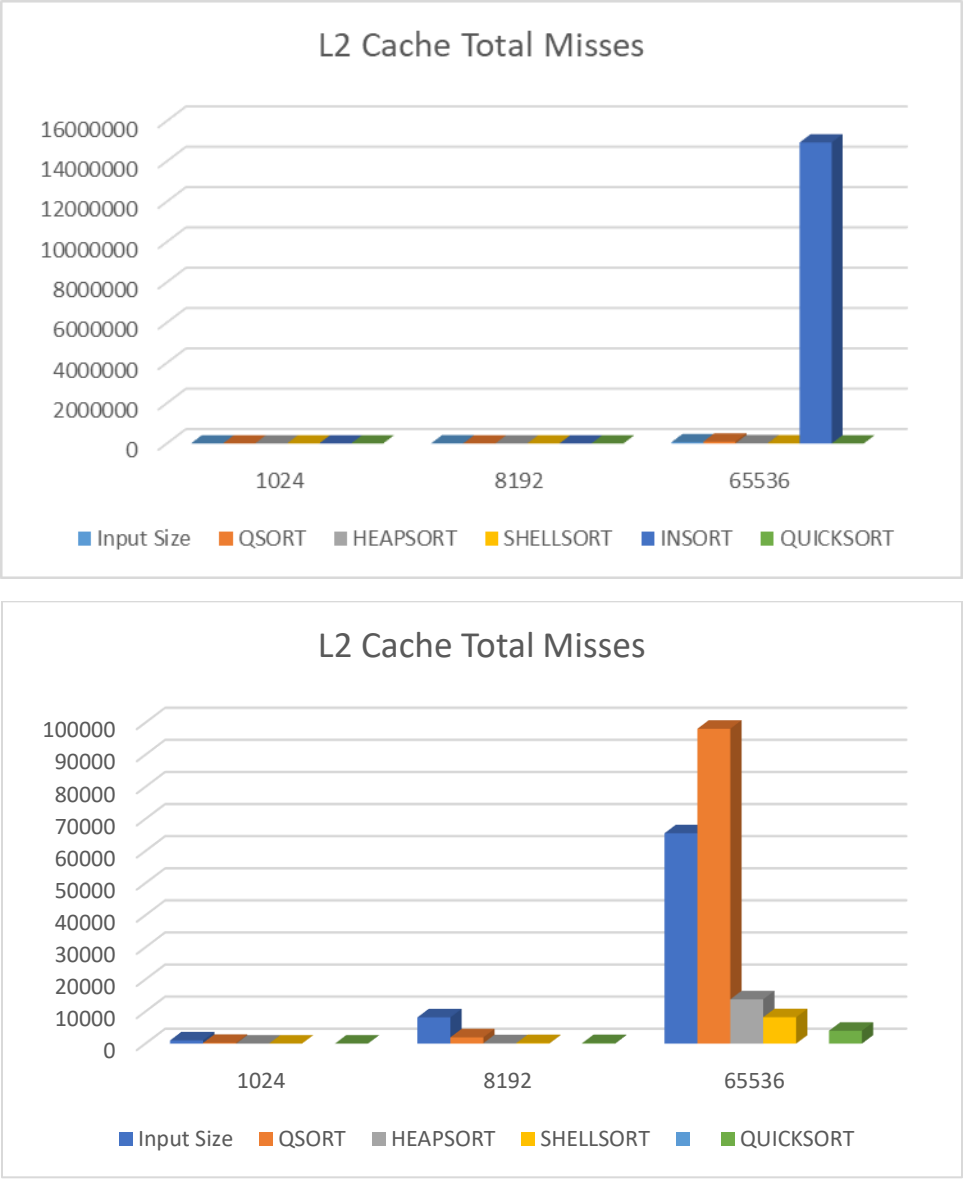


Fig. 5. L2 total cache miss comparison for different sorting algorithms with different input sizes.

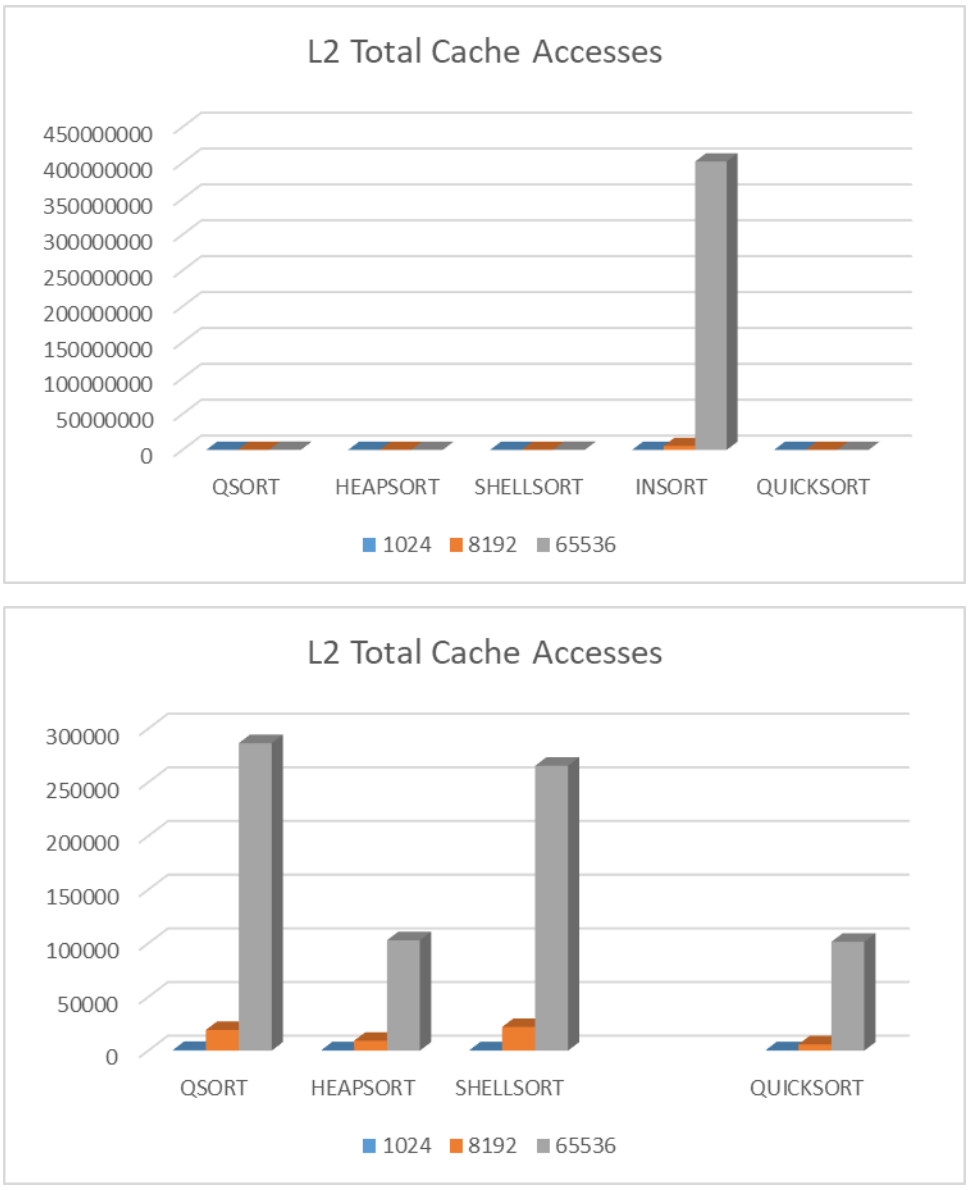


Fig. 6. L2 total cache access comparison for different sorting algorithms with different input sizes.

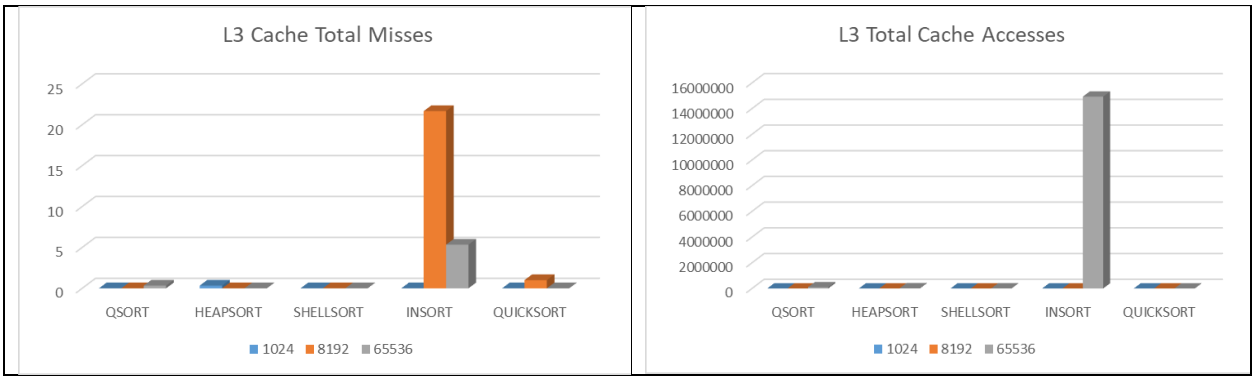


Fig. 7. L3 total cache access and total miss comparison for different sorting algorithms with different input sizes.

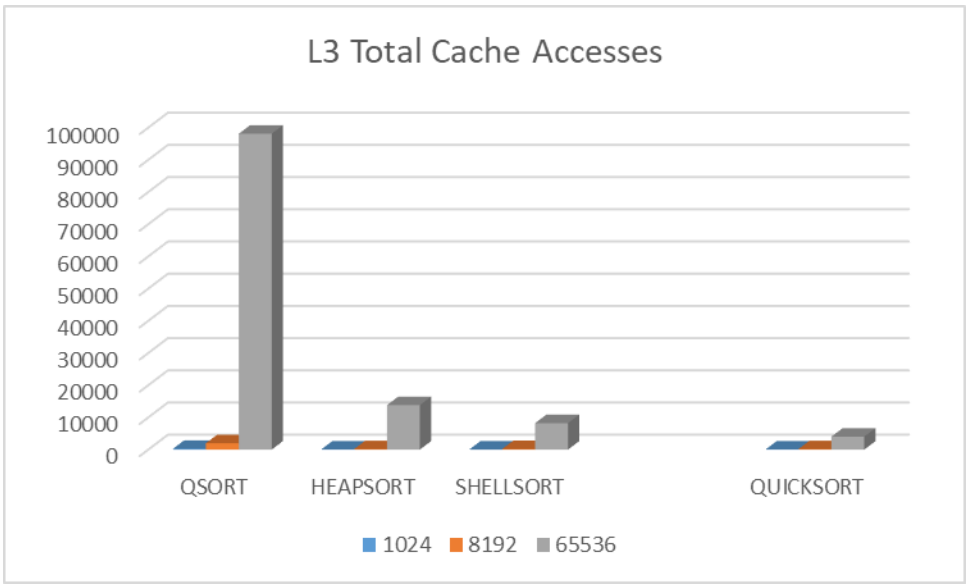
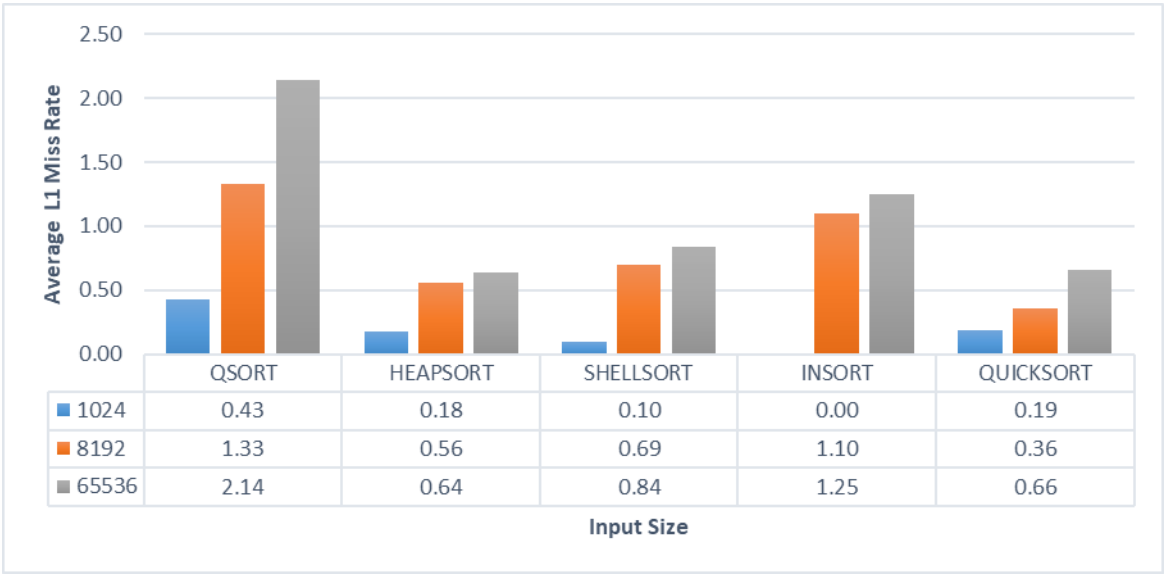


Fig. 8. L3 total cache access and total miss access comparison for different sorting algorithms with different input sizes.

Cache Miss rate Analysis:

Level 1 Cache:



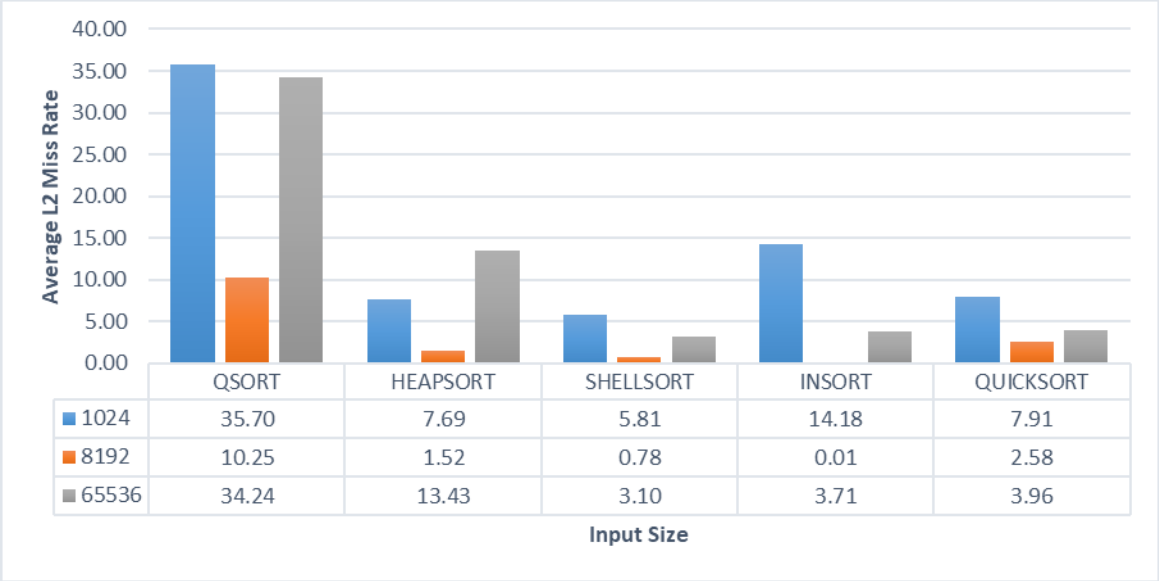
	QSORT	HEAPSORT	SHELLSORT	INSORT	QUICKSORT
1024	0.43	0.18	0.10	0.00	0.19
8192	1.33	0.56	0.69	1.10	0.36
65536	2.14	0.64	0.84	1.25	0.66

Fig. 9. Average L1 cache miss rate comparison for different sort algorithms and input sizes.

Level 1 Cache Miss rate Analysis:

1) Huge drop in L1 cache miss rate when using for input size of 1024.
2) Qsort has the highest L1 miss rate for all input sizes so it's the worst case for L1 cache.
3) Interesting observation about Insort algorithm miss rate for all input sizes. It has the zero L1 miss rate for smallest input size (1024). The total L1 cache access and misses in case of Insort algorithm take the highest values between all algorithms and that fact explains the relatively not very bad L1 miss rate of this algorithm when I compare it with Shell sort or Heap sort.
4) Quicksort has the lowest miss rate for both 8192 and 65536 input arrays.

Level 2 Cache:



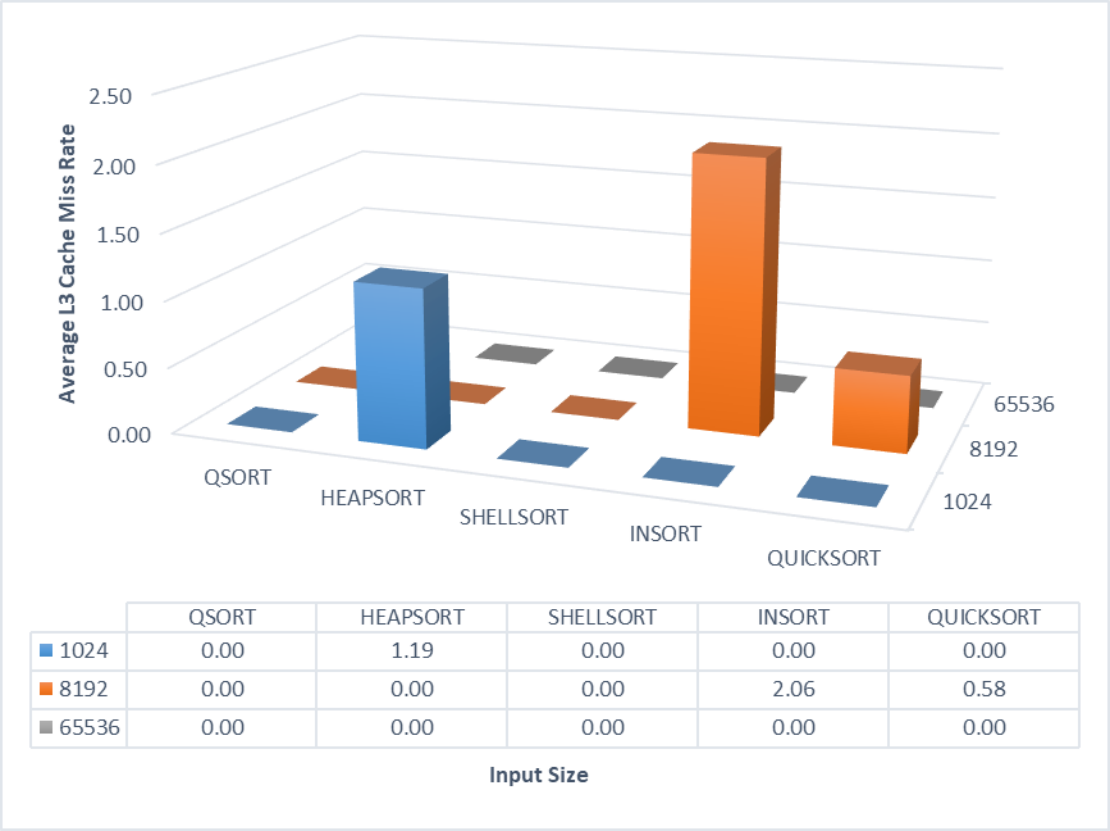
	QSORT	HEAPSORT	SHELLSORT	INSORT	QUICKSORT
1024	35.70	7.69	5.81	14.18	7.91
8192	10.25	1.52	0.78	0.01	2.58
65536	34.24	13.43	3.10	3.71	3.96

Fig. 10. L2 cache miss rate comparison for different sort algorithms and input sizes.

Level 2 Cache Miss rate Analysis:

1) We notice that miss rates for Shell sort is the lowest in L2 cache for all input sizes.
2) The input size 8192 results in the lowest L2 cache miss rate for any sorting algorithm. So, there should be a relation between size 8192 and L2 cache line size in our memory hierarchy
3) Insort algorithm results in the zero L2 cache miss rate for input size 8192.
4) Quick sort has the second lowest average miss rates for input sizes 1024 and 65536.
5) Input size 1024 is the worst input size in term of L2 cache miss rate.
6) Large miss rate of the L2 cache for Qsort explains the high number of total L3 cache accesses in figure 8.

Level 3 Cache:



	QSORT	HEAPSORT	SHELLSORT	INSORT	QUICKSORT
1024	0.00	1.19	0.00	0.00	0.00
8192	0.00	0.00	0.00	2.06	0.58
65536	0.00	0.00	0.00	0.00	0.00

Fig. 11. L2 cache miss rate comparison for different sort algorithms and input sizes.

Level 3 Cache Miss rate Analysis

1) L3 cache miss rate is almost zero for all the algorithms with input size 65536
2) L3 cache miss rate in case of using Heap sort is not the best case with input size of 1024
3) I noticed that Insort has higher miss rate in case of the input size 8192. When I compare this result with plots in figure 7, I can explain that a very high total number of L3 cache misses happens in case of input size 8192. While for the larger input size 65536, total number of misses in ¼ of that number for input size 8192. So, the input size 65536 is the best and input size 8192 is the worst in term of L3 cache miss rate.

Takeaway:

1) Quicksort results in the lowest L1 cache miss rate for all the input sizes while is not the best (lowest) in terms of L3 cache miss rate in case of input size 8192 and it is average in term of L2 cache miss rate.
2) Qsort results in the lowest(zero) L3 cache miss rate for any input sizes while Qsort results in the highest L1 and L2 miss rates.
3) Shell sort is the lowest in term of L2 cache miss rate for all input sizes and in term of L1 and L2 cache miss rates is one of the lowest.
4) Heapsort and Insort are not the best options in term of L1 and L3 cache miss rate.

Project Part 2:

In order to generate an estimate of the cache usage of the original code (without PAPI calls in it) using the valgrind toolkit with cachegrind, I used the following command and since in the first part of project, I counted the data cache accesses and misses, I am also extracting the data cache measurements from the Valgrind report. This repost does not cover the Level 2 cache in our hardware configuration, since it only reports the L1 and last level cache metrics.

```
srun -p cosc6385 valgrind --tool=cachegrind ./sort-test 1024
```

I got the valgrind report for each sort algorithm and each input size separately. For example, for the heap sort report, I just kept this function in the code (removed the rest of the sort algorithms) and got the valgrind report. Then in the same code, I also removed the heap sort function itself and got the valgrind report again. For each parameter, I subtracted the value of that parameter in the second report from the value extracted from first report. This difference will reflect the parameter value exactly for heap sort. I completed all the subtraction process for all the algorithms and all input size and the results are reported in the table tables 7 and 8.

For example, from Heap sort valgrind report I have extracted L1 cache total accesses and total number of misses as well as the L3 cache misses and total accesses

First Report:

```
acct10@crill:/home2/acct10/sort-test> srun -p cosc6385 valgrind --tool=cachegrind ./sort-test 1024
==49622== D   refs:      1,088,229  (813,216 rd   + 275,013 wr)
==49622== D1  misses:       45,389  ( 28,302 rd   +  17,087 wr)
==49622== LLd misses:       38,390  ( 22,117 rd   +  16,273 wr)
==49622== D1  miss rate:       4.2% (    3.5%     +     6.2%  )
==49622== LLd miss rate:       3.5% (    2.7%     +     5.9%  )
==49622==
```

```
==49622== LL refs:        46,417  ( 29,330 rd   +  17,087 wr)
==49622== LL misses:      39,398  ( 23,125 rd   +  16,273 wr)
==49622== LL miss rate:      0.9% (    0.5%    +    5.9%  )
```

And the second report:

```
acct10@crill:/home2/acct10/sort-test> srun -p cosc6385 valgrind --tool=cachegrind ./sort-test 1024
==49691== D   refs:      921,698  (689,362 rd   + 232,336 wr)
==49691== D1  misses:     45,184  ( 28,289 rd   +  16,895 wr)
==49691== LLd misses:     38,199  ( 22,117 rd   +  16,082 wr)
==49691== D1  miss rate:     4.9% (    4.1%    +    7.3%  )
==49691== LLd miss rate:     4.1% (    3.2%    +    6.9%  )
==49691==
==49691== LL refs:        46,202  ( 29,307 rd   +  16,895 wr)
==49691== LL misses:      39,197  ( 23,115 rd   +  16,082 wr)
==49691== LL miss rate:      1.0% (    0.7%    +    6.9%  )
```

L1 cache total accesses = 1,088,229 - 921,698 = 166531 (This value is so close to the total L1 cache accesses from PAPI report which was 164846)

L1 data cache misses = 45389 – 45184 = 205 (This value is close to the total L1 cache misses from PAPI report - 298)

L1 Cache Miss Rate = 100*(205/ 166531) = 0.123 (in comparison with the PAPI L1 cache miss rate reported is 0.182)

I noticed that for L1 cache accesses and misses the values reported by PAPI are in the same range as the values extracted from Valgrind reports. Tables 7 and 8 and figures 12 and 13 are showing the comparison between the two reports on total number of accesses and misses and also the miss rate of the L1 and L3 caches. I noticed that the difference between values are very similar and the only difference is in the L3 cache misses and total cache accesses (miss rate) reported by valgrind as it is obvious from figures 14, 15 and 16. Specially for the input size of 65536, the difference increases.

	Sort	L1_miss	L1_miss_valgrind	L1_access	L1_access_valgrind	L3_miss	L3_miss_val	L3_access	L3_access_val
1024	QSORT	640	390	149448	152,849	0	425	249	425
	HEAP	298	205	163846	166,531	0	201	25	215
	SHELL	293	205	300532	288874	0	198	19	212
	INSORT	371	205	7876670	7,354,365	0	197	59	211
	QUICK	304	208	163172	165850	0	204	26	224
8192	QSORT	19237.0	18892	1448292.7	1480929	0.0	1779	1978.7	19,120
	HEAP	9201.3	9334	1655364.3	1,679,431	0.0	1545	140.3	9,344
	SHELL	22038.0	23135	3175034.7	3047551	0.0	1542	171.7	23142
	INSORT	5549993.0	5518293	503418570.3	469876733	21.7	1541	743.3	5518299
	QUICK	5809.0	7220	1612748.7	1636725	1.0	1548	150.7	7236
65536	QSORT	286393	297267	13405112	13996216	0.3	12551	98096	356,136
	HEAP	102789	102986	15989949	16185683	0.0	12298	13808	102996
	SHELL	265383	269547	31589434	30277780	0.0	12295	8225	269554
	INSORT	401988992	401931732	32213480084	30065688573	5.3	12294	14931480	401931738
	QUICK	101331	112277	15273419	15469351	0.0	12301	4015	112291

Table 7. Comparing Average of PAPI report and Valgrind for L1 and L3 caches using different sort algorithms with input array size

Input Size	Sort	L1_MissRate	L1_MissRate_val	L3_MissRate	L3_MissRate_val
1024.000	QSORT	0.428	0.042	0.000	0.904
	HEAP	0.182	0.123	1.190	0.433
	SHELL	0.097	0.071	0.000	0.427
	INSORT	0.005	0.003	0.000	0.424
	QUICK	0.186	0.125	0.000	0.439
8192.000	QSORT	1.328	1.276	0.000	2.655
	HEAP	0.556	0.556	0.000	2.781
	SHELL	0.694	0.583	0.000	6.663
	INSORT	1.102	1.174	2.063	0.028
	QUICK	0.360	0.441	0.581	21.393
65536.000	QSORT	2.136	2.124	0.000	3.524
	HEAP	0.643	0.636	0.000	8.243
	SHELL	0.840	0.890	0.000	4.561
	INSORT	1.248	1.337	0.000	0.003
	QUICK	0.663	0.685	0.000	10.955

Table 8. Comparing Average of PAPI report and Valgrind Miss Rates for different sort algorithms with input array size

Fig. 12. Total Number of L1 cache access comparison for different sort algorithms and input sizes between PAPI metrics and Valgrind report.

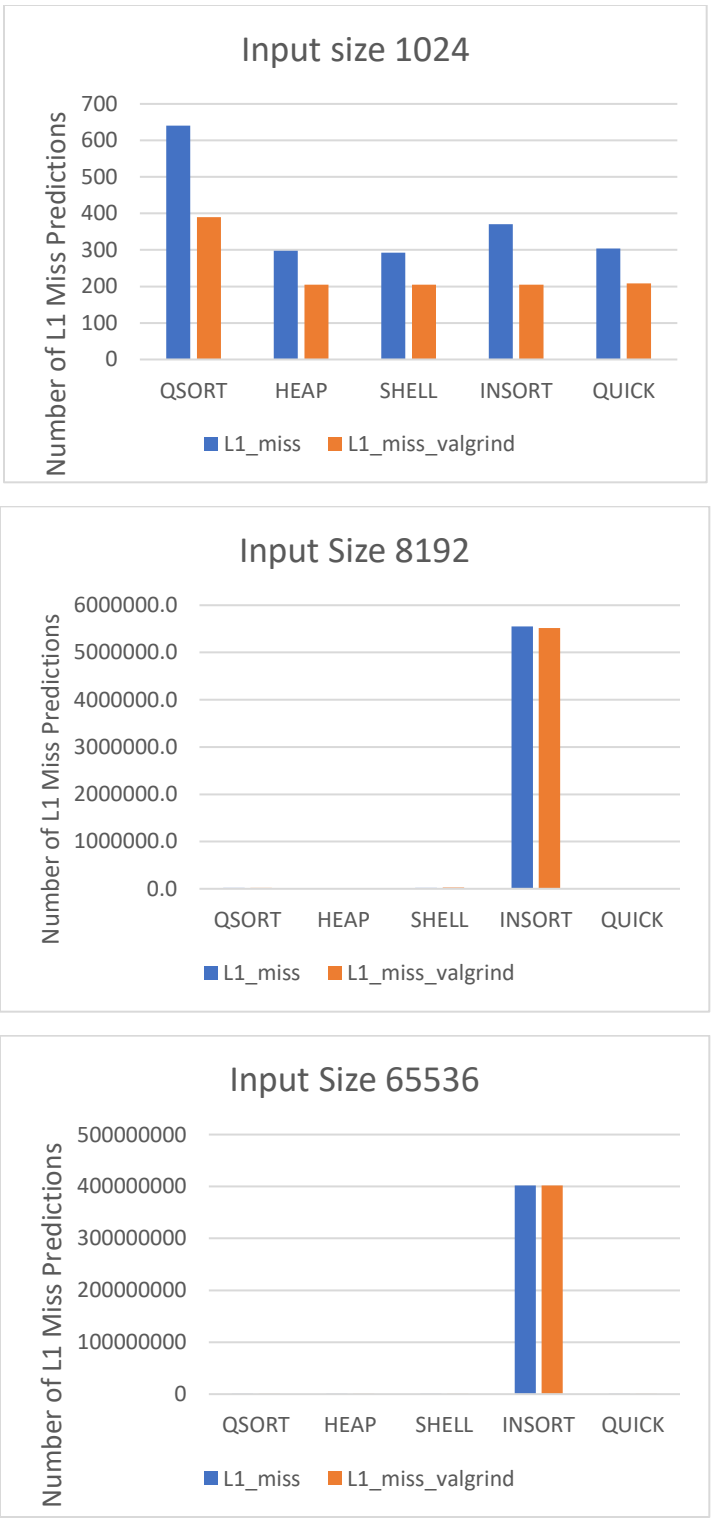


Fig. 13. Total Number of L1 cache miss comparison for different sort algorithms and input sizes between PAPI metrics and Valgrind report.

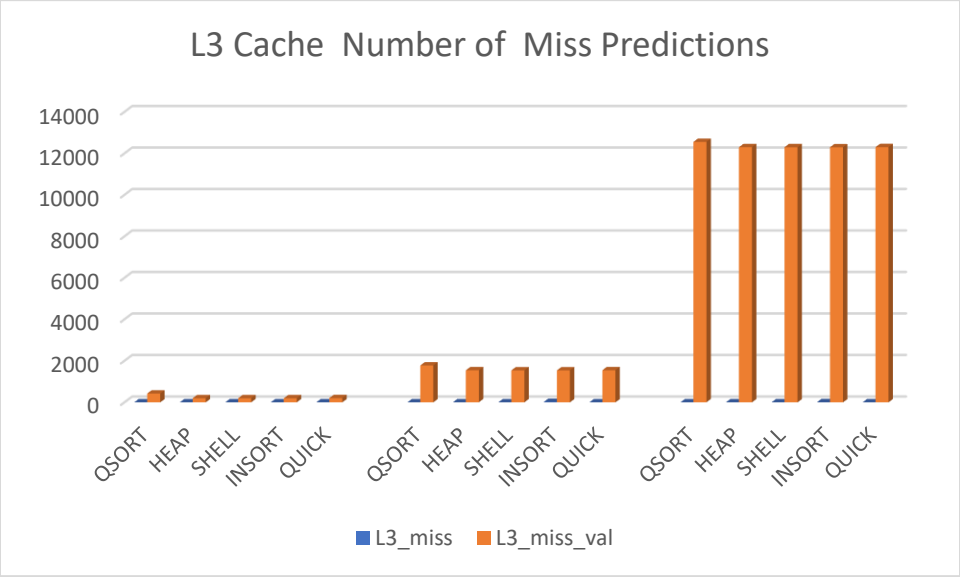


Fig. 14. Total Number of L3 cache miss comparison for different sort algorithms and input sizes between PAPI metrics and Valgrind report.

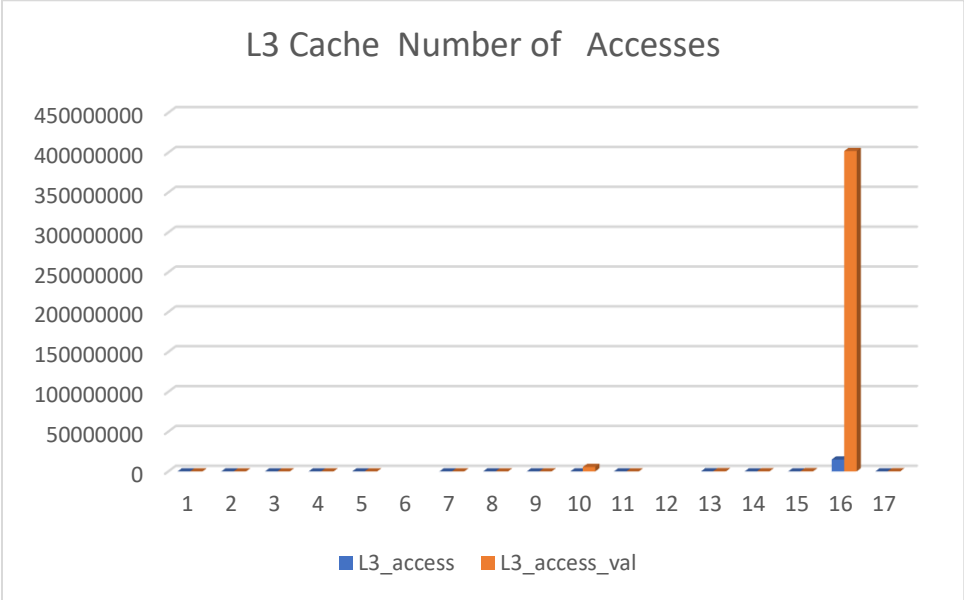


Fig. 15. Total Number of L3 cache access comparison for different sort algorithms and input sizes between PAPI metrics and Valgrind report.

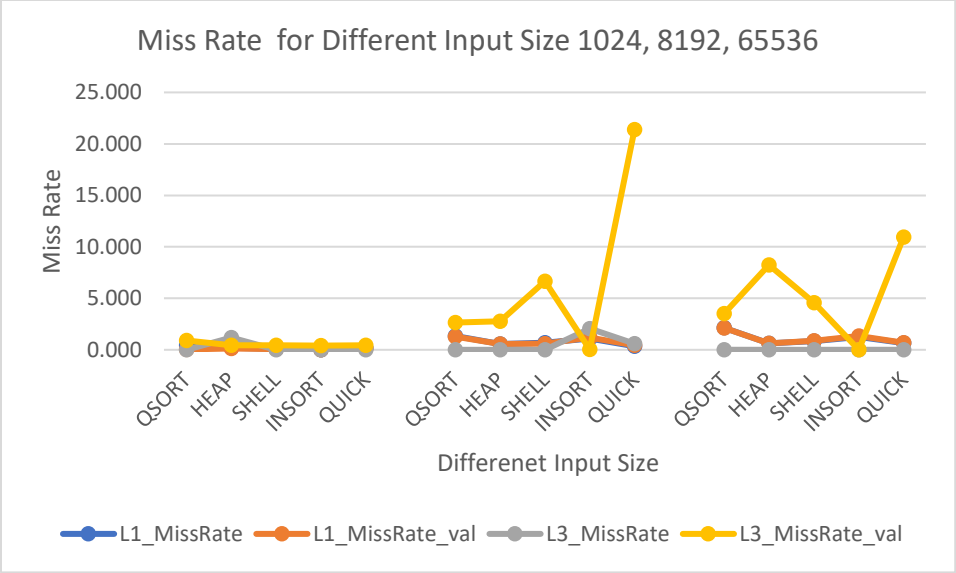


Fig. 16. L1 and L3 Cache Miss Rate comparison for different sort algorithms and input sizes between PAPI metrics and Valgrind report.