Improved Algorithm for Sorting via Heap Data Structure

Ravi Prakash Rathore, Rohan Verma, Antriksha Somani, Sunny Bagga

Abstract— Software engineering dependably looks for better approaches for enhancing the execution and proficient usage of equipment. This is accomplished fundamentally by actualizing different sort of plans and information structures to the projects for making them fill in as proposed with less complexity. The planners dependably attempt to limit the equipment use however much as could be expected by composing the proficient calculations that suites both the equipment and in addition programming. In the initial segment of this paper, we have created a make-pivot algorithm which makes use of heap data structure to produce a PARTIALLY-SORTED LIST and in the second part we have made some changes that takes the efficiency a step further with the help of insertion sort algorithm and after all this we combined both of them to produce New sorting technique. In the later part, we see different graphs that show various cases in which our sorting technique just outperforms native heap-sort by doing better CPU utilization.

Index Terms—Heap Data Structure, Algorithm profiling, Data Structure and Alogrithm, Sorting Algorithm and Analysis..

1 Introduction

Data sorting is a crucial part in most of the real-time applications. There are various factors that are responsible for achieving results which are efficient as well as reliable in terms of hardware and software. These factors include time complexities, no. of swaps and comparisons at the first place. So, by holding this as a primary objective we focused on making an algorithm that is efficient in terms of both space and time. We took advantage of the heap data structure and its properties and created the algorithm in such a way that it forms a partially sorted list in linear-logarithmic time or O (n*log (n)) and further we did research on how we must perform steps so as to get an array of numbers that are sorted properly and we came across some algorithms to achieve the end result. We found that the sorting technique which is well suited is the insertion sort and if we apply the insertion sort on the partially-sorted list then it performs in its average-case scenario where the time complexity is:

$$O(n^2)$$

So, ultimately our new sorting technique's theoretical complexity in RAM model is

$$O(n^2 + n * \log(n))[4]$$

The reason for doing this is because insertion sort is a stable sort and performs extremely well in case of nearly-sorted list. So, in this way we extirpated two short comes, one is that our first algorithm which makes the use of heap data structure property performs optimally and produces a partially-sorted list in linear-logarithmic time and second is algorithm which applies insertion sort in a specific way on the partially-sorted list and produces a sorted list approximately in quadratic polynomial time. Result turned out to be positive and our sorting algorithm outperforms conventional heap sort technique practically despite of being having a higher theoretical complexity. Thus, we concluded that the practical performance metrics of our sorting algorithm is more efficient than conventional heap sort sorting technique.

- *Ravi Prakash Rathore, IES IPS Academy, Indore*
- *Rohan Verma, IES IPS Academy, Indore,*
- *Anatriksha Somani, IES IPS academy, Indore*
- *Sunny Bagga, IES IPS academy, Indore*

2 Resources we used

To make sure that our algorithm will work on different environment we used different compilers, these includes Intel C++ compiler and Microsoft Visual C++ compiler. For doing the profiling we mainly focused on some profound profiler tools one of them is Intel VTunes, the others are Microsoft VS Profiler and Gprof. These tests are performed on Microsoft windows. These are performed over the Intel Core 2 Duo processor with a 3072MB of RAM and Intel Core i5 Microarchitecture Haswell processor with 8192MB of RAM. We categorized these in 3 test beds that are given below:

(1) Test Bed 1 (CPU Time): Core 2 duo 2.10 Gigahertz, 3072MB of RAM, Microsoft Windows 8.1 Pro edition build 9600, Microsoft Visual C++ compiler.

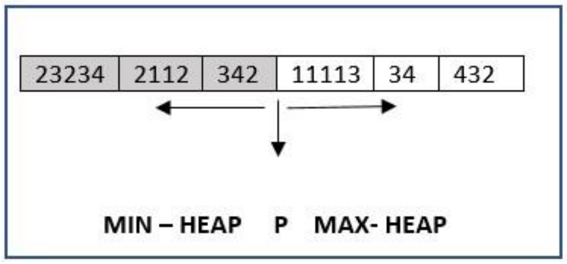
(2) Test Bed 2 (CPU Time): Core 2 duo 2.10 Gigahertz, 3072MB of RAM, Microsoft Windows 8.1 Pro edition build 9600, Intel C++ compiler.

(3) Test Bed 3 (Memory Analysis): Core i5 Haswell 3.20 Gigahertz, 8192MB of RAM (Quad core), Microsoft Windows 7 Ultimate edition build 7601, Intel Vtunes Amplifier 2017.

3 Algorithm

The algorithm 1 has primary arguments as A - the list itself, N as number of elements in the list and rest are auxiliary arguments initialized by the algorithm itself in later stages i.e. there is no need of explicit initialization for these auxiliary argument. MAKE-PIVOT algorithm basically creates a pivot or inflexion point in list. This sorting technique is an in-place sorting technique so it does not need any extra space but a constant space to carry out intermediate operations. The pivot so created after applying MAKE-PIVOT algorithm separates the elements in list in such a way that the elements on left of the pivot are always smaller than elements on right of the pivot. Thus, we can apply any sorting technique independently on both sides, the pivot created is always positioned at the middle of the list so complexity does not depend on position at which pivot is created. Algorithm initially proceeds by dividing the list at the middle (afterwards known as pivot) there by creating a Max heap [5] on the right side of the pivot growing in the direction away from the pivot and a Min heap [5] on the left side of the pivot also growing in direction away from the pivot so ultimately looks like following figure:

Fig. 1. "Pivot Contruction"



```
ALGORITHM 1: MAKE-PIVOT (A, N, L, R, P, H1, H2)
L <- floor (N/2)
R <- N-1
REVERSE-MIN-HEAP (A [0:L-1], L)
MAX-HEAP (A [L: R], R)
P <- 0
H1 <- 0
H2 <- R
if A[L-1] > A[L]:
    SWAP(A[L], A[L-1])
while R>L:
    SWAP(A[L-1], A[P])
    H1 <- H1+1
    SWAP(A[L], A[R])
    H2- <- H2-1
    if A[L-1]>A[L]:
        SWAP(A[L], A[L-1])
    REVERSE-MIN-HEAPIFY (A, L-1,L, H1)
    MAX-HEAPIFY (A [L: R], 0, H2)
    R <- R-1
    P <- P+1
stop
```

Data structure that we are using to develop sorting technique is an Array because using that we can do random access and also we are only holding some numbers in memory (talking a fixed number of bits) not some big structure, thus compilers would not have any issue in requesting a new contiguous block of memory from MMU. But if the data structure is Linked List then we have to worry about a lot of pointers and also the Linked list is suitable when we are supposed to hold large or variable size structure but this is not the case meanwhile. Also, while traversing the Linked list we are doing random access continuously from one node to another node which maximizes something called cache misses or unavailability of the pages which would add a number of redundant CPU cycles and thus hampering the performance. Also, the heap data structure can be best represented as an Array so we always have advantage while using array.

Now in Figure 1 we can see that the pivot is represented as P and Min heap and Max Heap are well specified. Min heap can also be called as Reversed-Min heap if we observe carefully in Figure 1. Again, dissolve the Heap size for the Min heap i.e. set H1 = 0. So, Min heap is now a heap of size "0" positioned at the pivot and heap size for Max heap H2 remains as size of the right half. The construction of the Reversed-Min heap is done via algorithm 2 and corresponding heapify operation is performed via algorithm 3. The construction of Max heap [4] [5] [7] is done via default mechanism for creating Max heap. Now to proceed we first compare the root or first element of both heaps, for Reversed-Min heap this element is at immediate left of the pivot and for Max heap this element is at immediate right of the pivot. If the root element of the Reversed-Min heap is greater than root element of Max heap then swap these two elements and vice-versa, this is done to separate the smaller and bigger numbers to left and right of pivot respectively. Now we enter the core part of the algorithm, along with heap-sizes we maintain two pointers P and R where P is initialized as 0 and R is initialized as last element of the list. Now remember that we are using a constant space but not any extra space every time we are making modifications to list.

```
ALGORITHM 2: REVERSE-MIN-HEAP (A, N)
J <- floor(N-(N/2))
while J<N-1:
        REVERSE-MIN-HEAPIFY (A, J, N, 0)
        J<- J+1
stop
```

```
ALGORITHM 3: REVERSE-MIN-HEAPIFY (A, I, N, H1)
while 1:
        R <- (2*I)-(N+1)
        L <- R+1
        if R >= H1:
                M <- I
                if A[R] < A[M]:
                M <- R
                if A[L] < A[M]
                        M <- L
                            if M <> I:
                                SWAP(A[I], A[L])
                                I <- L
                                    continue
                            else:
                            stop
        if L<H1:
                stop
        if A[L]<A[I]:
                SWAP(A[L], A[I])
                stop
```

```
ALGORITHM 4: MAX-HEAPIFY (A, I, H2)
while 1:
        R <- (2*I) + 2
        L <- R-1
        if R<-H2:
           X <- I
           if A[R]>A[L]:
                  L <- R
           if A[R-1] > A[L]:
                  L <- R-1
                  if L <> I:
                      SWAP(A[I], A[L])
                      I <- L
                      continue
                  else:
                      stop
        if L>=H2:
            stop
        if A[L]>A[I]:
           SWAP(A[R-1], A[I])
stop
```

```
ALGORITHM 5: MAX-HEAP (A, N)
J <- floor ((N-2)/2)
while J>=0:
            MAX-HEAPIFY (A, J, N)
            J <- J-1
stop
```

Now we repeat the following process till R becomes less than L (which is middle of list = N/2). One thing to notice is that leaf nodes of the Reversed-Min Heap which are far away from the pivot can be among the greater elements of the list which belong to right of the pivot and also the leaves of the Max heap can be among the smaller elements which belong to left of the pivot. This property of our augmented-list (having a Min Heap and Max heap created respectively on left and right side of pivot) is manipulated by the algorithm in further steps. For doing this we decrement R pointer and increment P pointer and also increment the size of Min heap H1 which is zero and decrement size of Max heap H2.

First, we swap immediate element on left of pivot and element pointed by P pointer and we increment the size of the Min heap H1 and also, we swap immediate element on right of pivot and element pointed by R pointer and we decrement the size of the Max heap H2 and then finally, we compare if the root element of the Reversed-Min heap is greater than root element of Max heap then swaps these two elements and vice-versa along with the decrement of R pointer and increment of P pointer. Repeat above process till condition turns false.
The result of the MAKE-PIVOT algorithm looks like following:

342 432 34 2112 23234 11113

```
ALGORITHM 6: NEW-SORT (A , N)
MAKE-PIVOT (A [0: N-1], N)
ISORT (A [0: floor (N/2) -1], floor (N/2))
ISORT (A [floor (N/2): N-1], N - floor (N/2))
stop
```

We can see that every element in left of the pivot is always smaller than every element present on right of the pivot. This is the sole purpose of MAKE-PIVOT algorithm and along with this we get an almost sorted pattern on both sides of pivot. Now we can apply insertion sort algorithm (ISORT) on both parts on left and right independently which is described in algorithm 6.
First step in this NEW-SORT algorithm is MAKE-PIVOT algorithm, we have already done that. The output of the MAKE-PIVOT algorithm is processed further by applying the insertion sort on left & right sides of pivot. The result we get is a sorted list.
Look carefully that array indexing used is in C-style indexing i.e. from 0 to N-1 not in FORTRAN style indexing i.e. from 1 to N. The floor, continue and stop functions used here have usual meaning as floor stands for Least Integer function, continue is the programming construct from C language and stop means terminate the algorithm. ISORT is standard insertion sort sorting algorithm.

4 Complexity

Max-Heap has the complexity [4] = O(n)
Max-Heapify has the complexity [4] = O(log(n/2))
Reverse-Min-Heap has the complexity [4] = O(n)
Reverse-Min-Heapify has the complexity [4] = O(log(n/2))
The worst-case complexity of MAKE-PIVOT algorithm is given below:

$$O(1) + O(n) + O(n) + O(1) + O(n) + O(n * \log(n)) \\ = O(n * \log(n))$$

The worst-case complexity of our sorting NEW-SORT algorithm is given below:

$$O(n * \log(n)) + O(n^2) = O(n^2)$$

Theoretical complexity of the sorting technique presented in this paper in RAM model is much higher than that of Heap Sort sorting technique. But practical performance of this algorithm outweighs the performance of Heap Sort.

5 Performance Metrics

The performance of the algorithm is assessed on following criteria:
(1) Time taken (in milliseconds).
(2) Memory utilization.
(3) Number of comparisons and swaps.

5.1 Time taken

The various algorithms are profiled with respect to time defined by CPU clock.

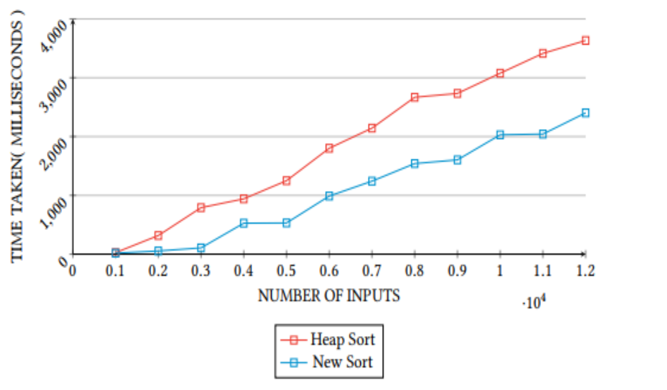


Fig. 2. Heap Sort vs New Sort (Unsorted , Visual C++ Compiler).

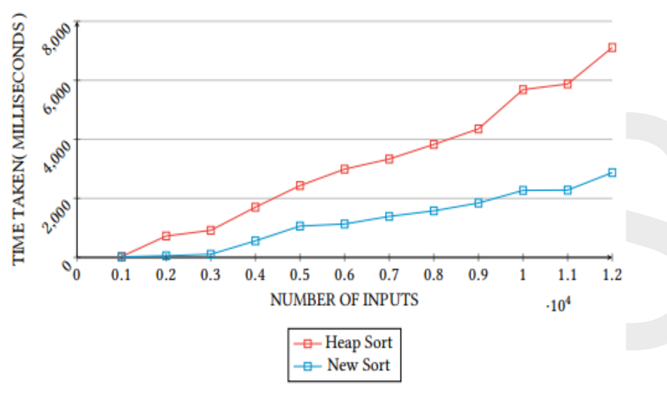


Fig. 3. Heap Sort vs New Sort (Unsorted , INTEL C++ Compiler).

As we can see in the depicted graphs in fig 2, 3, the profiling is performed on an unsorted array on both the compilers. It is clearly visible with this that the performance of our sorting algorithm is far better than Heap Sort.

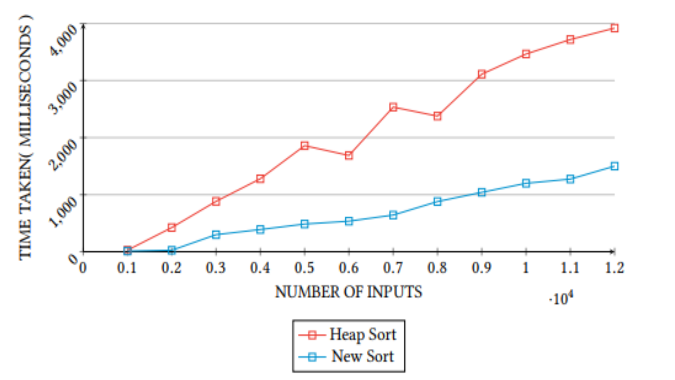


Fig. 4. Heap Sort vs New Sort(Almost sorted , Visual C++ Compiler).

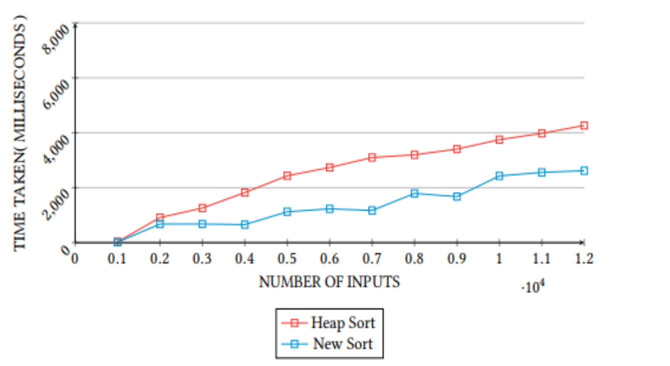


Fig. 5. Heap Sort vs New Sort
(Almost sorted , INTEL C++ Compiler).

We applied same concept on the partially sorted array and the results are shown in graph 4, 5

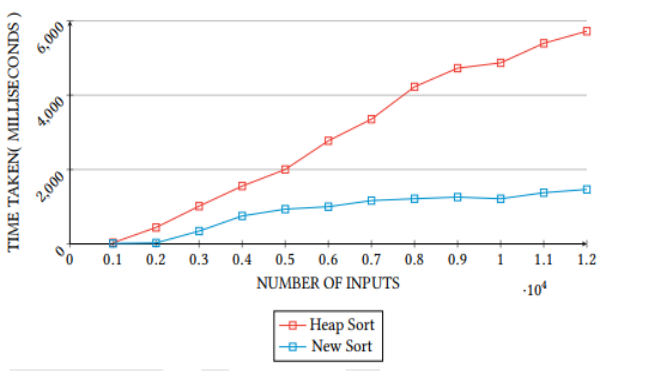


Fig. 6. Heap Sort vs New Sort (Sorted , Visual C++ Compiler)

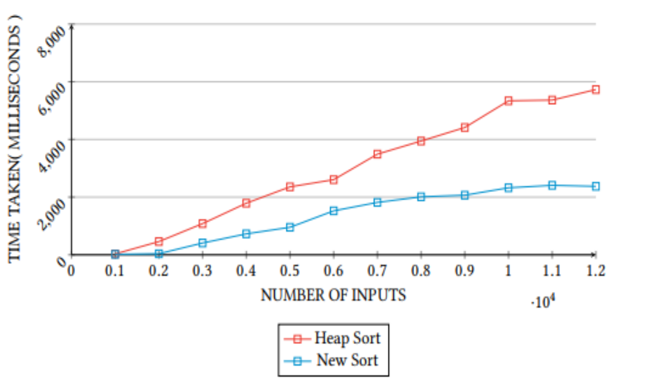


Fig. 7. Heap Sort vs New Sort (Sorted , INTEL C++ Compiler).

Again, in the above graph 6, 7 we have taken the observations on the sorted array and we observed with the result that the heap sort takes a lot of time on the sorted array as compared to our sort. So, it is clear visible from the results on various test beds that our sort performs much more efficiently in all the environment where the array might be sorted, unsorted or partially sorted. The time taken by our “New Sort” is much less than the Heap sort with the same input values. We can take the gradient or slope of our sorting algorithm and heapsort in graphs and can deduce that our sorting algorithm is approaching a finite value on a very large input.

5.2 Memory Utilization

In order to verify the feasibility of algorithm, we perform several tests using Intel VTune Amplifier 2017 performed over Intel Core i5 processor. To prove that the result obtained from our new sorting technique is much better than conventional heap sort algorithm, we basically used hotspot analysis and memory analysis on one of the implementation of our sorting technique. Before comparing the performance we need to understand the platform on which the test is performed. Intel Core i5 haswell microarchitecture has three levels of caches namely L1, l2 and L3. The L1 has latency cycle of about 4 Cycles, it has both the instruction and data cache. The L1 cache only demands the cache line from higher level cache such as L2 and L3 only and only if there is a cache miss in L1 cache itself. The L1 cache writes to L2 cache only with write-through operations. L2 and L3 both share code and data with each other spontaneously. DRAM also called as main memory loads the data/instruction from storage and is only accessed if there is a cache line missing in L1, L2 and L3. Intel Core i5 haswell microarchitecture is a load-store architecture in which instructions are pre-fetched either in L1 cache or SRAM before they get executed and this includes branch prediction and data forwarding, and also the CPU might execute more or less number of instruction than it actually have to execute. This technique is also called as speculative execution. Modern processors have pipeline execution to support this speculative execution. The following memory analysis has been done on both Heapsort and our NEW Sort sorting technique. The memory analysis test [1] [2] has been carried out on input of 1 million numbers(unsorted), the observations made to justify the less CPU time consumed by our NEW Sort sorting technique as compared to Heapsort are:
Elapsed Time -: The total time used by profiler (Intel Vtunes Amplifier) to execute the program/application and to profile it [3].
CPU Time -: Total time for which the program is actively executed by CPU.This is important because the operating system does schedule the programs/process and in order to calculate the CPU time, the process context switching time and CPU time for which the program is idle has to be removed. This is typically calculated in seconds [3].
Memory Bound -: Memory bound measures a fraction of slots where pipeline could get stalled because of store instructions. This accounts for incomplete transient memory demand loads that overlap with waiting time of program in addition to less trivial cases where stores could imply back-pressure on the pipeline [3].
L1 Bound -: This metric shows how often machine was stalled without missing the L1 data cache. This is expressed in percentage of clock ticks. The L1 cache has profoundly least access time. However, in certain cases like data dependency, high latency can be observed despite being satisfied by the L1 [3].
L2 Bound -: This metric shows how often machine was stalled on L2 cache. This is expressed in percentage of clock ticks [3].
L3 Bound -: The L3 cache is inclusive cache i.e. shared by all processor cores. This metric shows how often CPU was stalled on L3 cache, or contended with a sibling Core (in this case we have 4 cores). This is expressed in percentage of clock ticks [3].
LLC Miss Count -: The LLC (last-level cache) is the last, and longest-latency, level in the memory hierarchy before main memory (DRAM). Any memory requests missing here must be serviced by local or remote DRAM, with significant latency. The LLC Miss Count metric shows total number of demand loads which missed LLC. Misses due to HW prefetcher are not included [3].
DRAM Bound -: This metric shows how often CPU was stalled on the main memory (DRAM).
This is expressed in percentage of clock ticks [3].
The Memory Bound is expressed as percentage of pipeline slots. A pipeline slot is the hardware resource that is needed to process one micro operation.
If we carefully compare the Memory Bound of Heapsort and NEW sort, Memory Bound of our sorting technique represents that more than 11% of CPU resources are wasted waiting for memory operations to complete as in case of Heapsort just 0.1% of CPU resources are wasted waiting for memory operations to complete. But this is not a negative impact, we further will show why exactly the result is like this in next section.

TABLE 1
PROFILING SUMMARY OF HEAPSORT

Elapsed Time	7.912s
CPU Time	0.150s
Memory Bound	0.10%
L1 Bound	10.70%
L2 Bound	0.00%
L3 Bound	0.00%
DRAM Bound	3.00%
Loads	120,903,627
Stores	237,003,555
LLC Miss Count	0
Average Latency (cycles)	10
Total Thread Count	1

Our profiling results in table 1 and table 4 shows that L1 Bound of NEW sort technique is lower than Heapsort (clock ticks for both can be calculated by CPU Time given), for analysis of this we have to find which are exactly the hot zones (parts of program which takes maximum CPU time) in both Heapsort and NEW sort. Heapify Operation and Swap operation in Heapsort are hot objects/spots in it. And only one hot spot in NEW sort which works for most of the time is insertion sort operation because the function is responsible for producing a sorted list.

TABLE 2
LATENCY CYCLE FOR HOTSPOTS OF HEAPSORT

Function	CPU Time	L1 Bound	Average Latency (cycles)
heapify	0.059s	0.032	8
swap	0.039s	0.099	9

In case of Heapsort, we see that most of CPU time is spent in two function heapify and swap (hot spots), we can also see that both functions are L1 and DRAM bound. The number of elements in list is 1 million and so this should not fit fully into the L1 cache but the “Average Latency” metric shows a latency of 8 cycles in case of heapify operation. This exceeds the normal L1 access latency of 4 cycles.In table 2, we can see that the swap operation also has “Average Latency” metric and shows a latency of 9 cycles.

Again, we exceeded the L1 normal access latency of 4 cycles, which often means that we have some contention issues that could be either true or false sharing.

TABLE 3
Memory bandwidth (Heapsort)

Function	Memory Latency	Memory Bandwidth
heapify	3.20%	69.9%
swap	30.90%	69.1%

True sharing can be easily avoided by adding padding so that threads always access different cache lines. But in case of only one thread, true sharing may not be possible. A cache line size greater than a word is also a reason for False-Sharing. This can further be attributed by looking at DRAM bandwidth for both operations and these are 69% of total clock ticks. So, in conclusion Heapsort uses the memory extensively and thus have higher memory traffic than NEW sort technique. That is in Heapsort we have used more swap operations than in our sorting technique.

TABLE 4
PROFILING SUMMARY OF NEW SORT

Elapsed Time	0.192s
CPU Time	0.140s
Memory Bound	11.90%
L1 Bound	7.60%
L2 Bound	0.00%
L3 Bound	0.00%
DRAM Bound	1.30%
Loads	697,520,925
Stores	125,401,881
LLC Miss Count	0
Average Latency (cycles)	12
Total Thread Count	1

Now coming to our NEW sort technique the L1 Bound and DRAM bound are both lower than Heapsort technique.

TABLE 5
LATENCY CYCLE FOR HOTSPOTS OF NEW SORT

Function	CPU Time	L1 Bound	Average Latency (cycles)
Insertion Sort	0.136s	0.039	12

TABLE 6
MEMORY BANDWIDTH (NEW SORT)

Function	Memory Bandwidth	Memory Latency
Insertion Sort	27.30%	48.20%

Again we see that insertion sort is only hot spot in our sorting technique and we can observe that memory bandwidth is 27.30 % of total clock ticks in accordance with the table 6. Thus, we concluded that our sorting algorithm is using less memory bandwidth or have less memory traffic because we have used less number of swaps operation than Heapsort. We have done this by making additional number of comparisons.

Recall that we noticed that memory bound of our sorting algorithm is 11 % which is much higher than Heapsort which is 0.1 %. This attributes to the previous deduction of lower bandwidth usage of memory by our sorting algorithm. Efficient sorting algorithm can be designed if we can make additional comparisons to reduce number of swaps happening which is the approach adopted while designing this algorithm.

5.3 Number of Comparisions and Swaps

To verify the analysis in previous section, we can see in table 7 that the NEW sort algorithm is clearly taking more number of comparisons than the Heapsort but also taking less number of swaps than Heapsort.

To stipulate the efficiency of our sorting algorithm over Heapsort we have following facts: Swaps must be taking more CPU time because it fetches data from memory to cache and then cache to registers and finally back to cache.

Comparisons must be taking less CPU time (compared to Swaps) because it accompanies following (in general):

(1) Fetching data from memory to cache [6].
(2) Fetching daat from cache to registers [6].
(3) Executing single compare operations on two registers (which should be a little fasster than writing two integers into a chache) [6].

Additionally, a conditional jump after a comparison may not be taking more cycles because of: branch prediction and L1 code cache. The first one will save from cleaning the pipeline due to jump, the second one will save from memory access and operations decoding.

TABLE 7
COMPARISONS AND SWAPS BASED ON THE NUMBER OF INPUTS

Algorithm	Inputs	Comparisons	Swaps
NEW Sort	1000	41081	7106
Heapsort		53028	9467
NEW Sort	2000	155564	23020
Heapsort		170910	30382
NEW Sort	3000	384888	48176
Heapsort		358845	63636
NEW Sort	4000	727454	83479
Heapsort		619108	109596
NEW Sort	5000	1239961	129187
Heapsort		954379	168647
NEW Sort	6000	1944863	185232
Heapsort		1366343	241080
NEW Sort	7000	2876701	251925
Heapsort		1855886	326966
NEW Sort	8000	4097471	329050
Heapsort		2423759	426581
NEW Sort	9000	5594457	417136
Heapsort		3072053	540307
NEW Sort	10000	7373700	516828
Heapsort		3802530	668367
NEW Sort	11000	9423673	628051
Heapsort		4615511	810769

6 CONCLUSION

The sorting algorithm presented is based on the fact that we can make additional comparisons to reduce the number of swaps and consequently less memory traffic. A similar approach can be taken by using a k-ary heap data structure instead of a binary heap data structure. All these kind of approaches give a more efficient performance as compared to conventional Heapsort sorting technique.

REFERENCES

[1] INTEL. 2017. Events for Intel Microarchitecture codename Haswell. (June 2017). Retrieved April 2, 2017 from https://software.intel.com/en-us/node/597060.

[2] INTEL. 2017. Intel optimization manual, Appendix A Application Performance Tools, Appendix B Using Performance Monitoring Events. (June 2017). Retrieved June 21, 2017 from http://www.intel.com/content/dam/www/public/us/en/ documents/manuals/64-ia-32-architectures-optimization-manual.pdf

[3] INTEL. 2017. Intel VTune Amplifier Tutorials. (June 2017). Retrieved June 21, 2017 from https://software.intel.com/ en-us/articles/intel-vtune-amplifier-tutorials

[4] Donald Knuth. 1998. The Art of Computer Programming (2nd. ed.). Sorting and Searching, Vol. 3. Addison-Wesley Professional, New York, NY.

[5] R. Schaffer and R. Sedgewick. 1993. The Analysis of Heapsort. J. Algorithm (1993). https://doi.org/10.1007/ 978-3-540-31856-9_52

[6] Robin Skafte. 2015. Memory consistency in the Haswell multicore architecture. (Dec. 2015). Retrieved June 27, 2017 from http://www.eit.lth.se/fileadmin/eit/courses/edt621/Rapporter/2015/robin.skafte.pdf

[7] I. Wegener. 1993. Bottom-Up-Heapsort, a new variant of Heapsort beating, on an average, Quicksort (if n is not very small). Theoretical Computer Science 118 (1993), 81–98. https://doi.org/10.1016/0304-3975(93)90364-Y