CS6600 Assignment 1

Akshat Joshi - EE19B136

15th August 2022

1 Introduction

This assignment involves simulating different sorting algorithms on Cachegrind to check their miss rate behavior for different cache configurations

2 Running the code and output

- In the sorting folder the C code comprises of two files main.c and sort.c. Compile the code using the following command:

```
gcc main.c -lm -o main.o
```

- The syntax for running the sorting program is as follows:

```
./main.o <sorting_algorithm> <size_of_array>
```

- You can choose one of quick,radix,merge,bubble,selection for the sorting algorithm parameter.
- Example: To sort a random array of size 100 using quick sort run:

```
./main.o quick 100
```

- The python files analyze.py and plots.py have been provided for running cachegrind for the sorting algorithms and generating the plots using the cachegrind output.
- All Cachegrind output is stored in the dump folder and plots are stored in the plots folder.

3 Default Cache Configuration Statistics

The cache miss rate statistics for L1 (data) cache and Last Level (data) cache for various sorting algorithms for various sizes has been plotted below. From the bar graphs we observe that **Quick Sort** seems to be the most cache friendly algorithm, having the lowest L1 (data) and LLd cache Miss Rate across all sorting algorithms.

Note that for bubble sort plot is not done for 100000 as it was taking a lot of time. This is because it has a worst case time complexity of $O(n^2)$. For such large array sizes it would take very very long to simulate these inefficient algorithms on Cachegrind.

3.1 Cache Configuration

The default cache configuration is as follows:

- **L1 (Instruction) cache:** 65536 B, 64 B, 4-way associative
- **L1 (Data) cache:** 32768 B, 64 B, 8-way associative
- **Last Level cache:** 4194304 B, 64 B, 16-way associative

3.2 L1 (Data) Cache Miss Rate statistics

We can see that **Quick Sort** has the least L1 cache miss rate followed by radix sort, merge sort, bubble sort and selection sort.

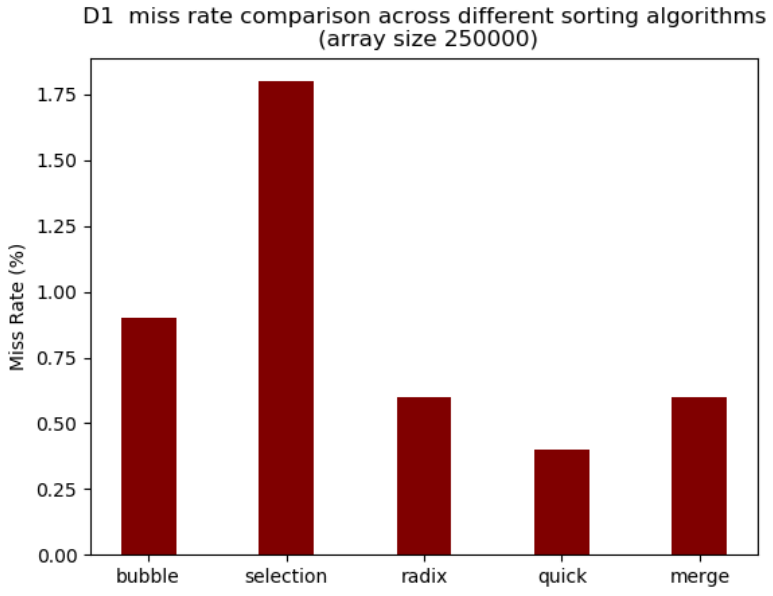

Figure 1: D1 Cache Miss Rate (array size 250000)

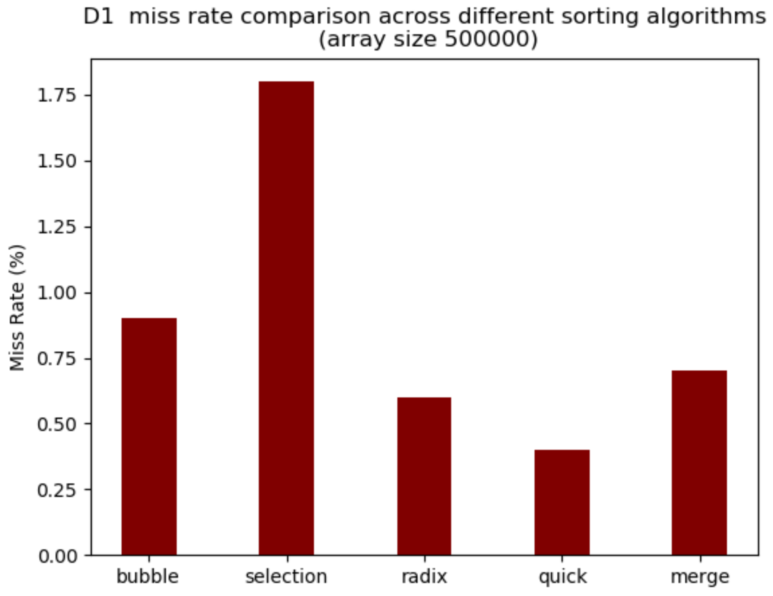

Figure 2: D1 Cache Miss Rate (array size 500000)

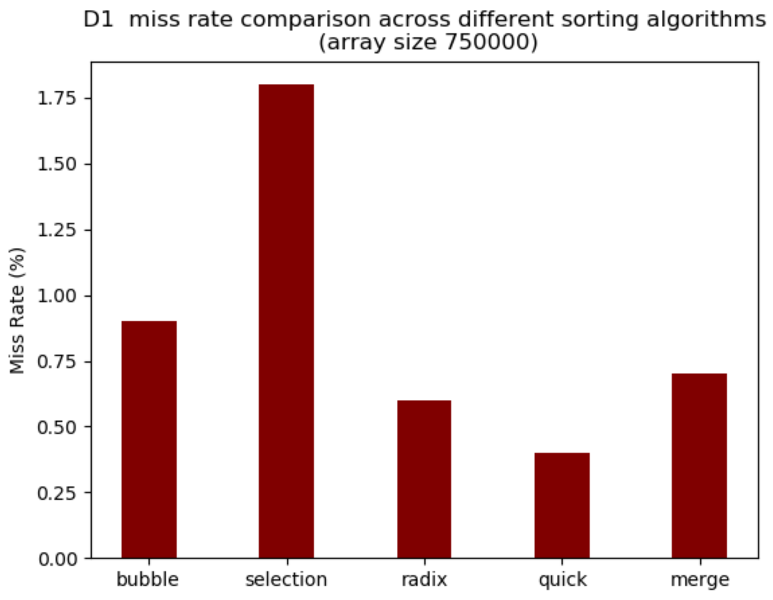

Figure 3: D1 Cache Miss Rate (array size 750000)

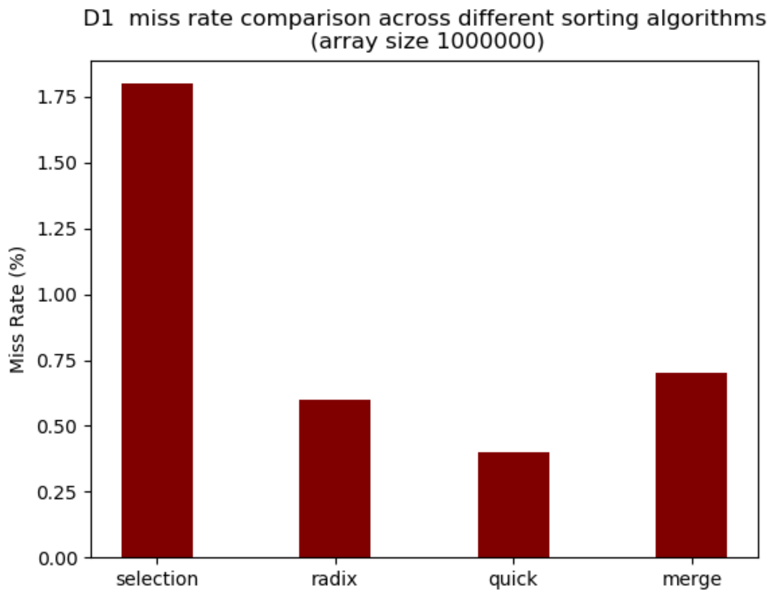

Figure 4: D1 Cache Miss Rate (array size 1000000)

3.3 Last Level (Data) Cache Miss Rate Statistics

Quick Sort again has the best cache performance. For array of size 250000 radix sort seems to be better than merge sort for last level cache miss rate. But for larger arrays radix sort performs worse than merge sort.

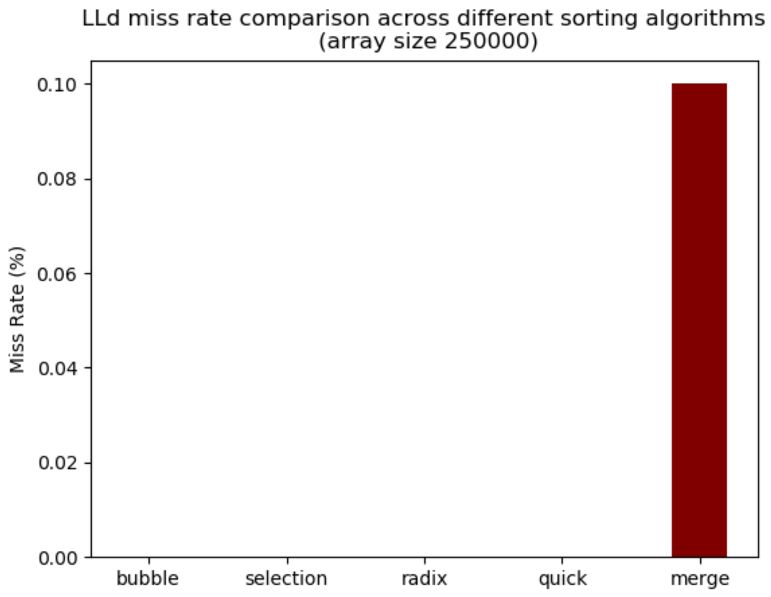

Figure 5: LLd Cache Miss Rate (array size 250000)

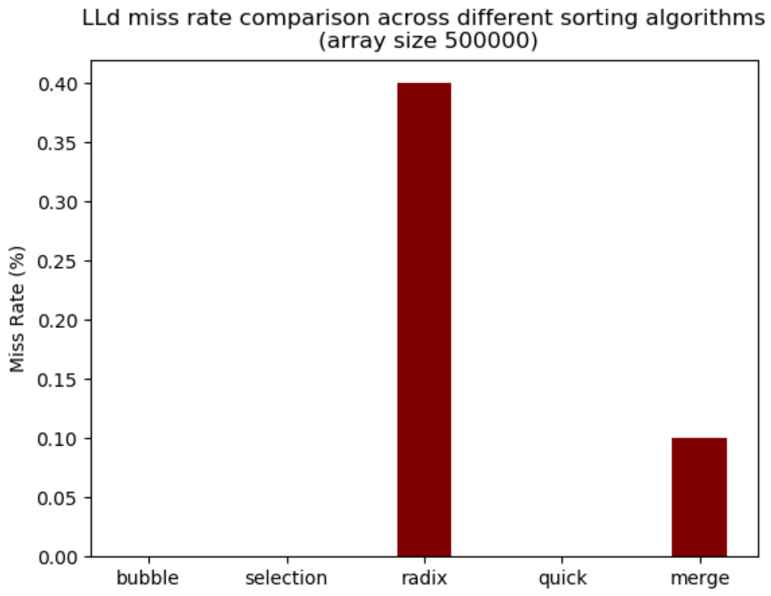

Figure 6: LLd Cache Miss Rate (array size 500000)

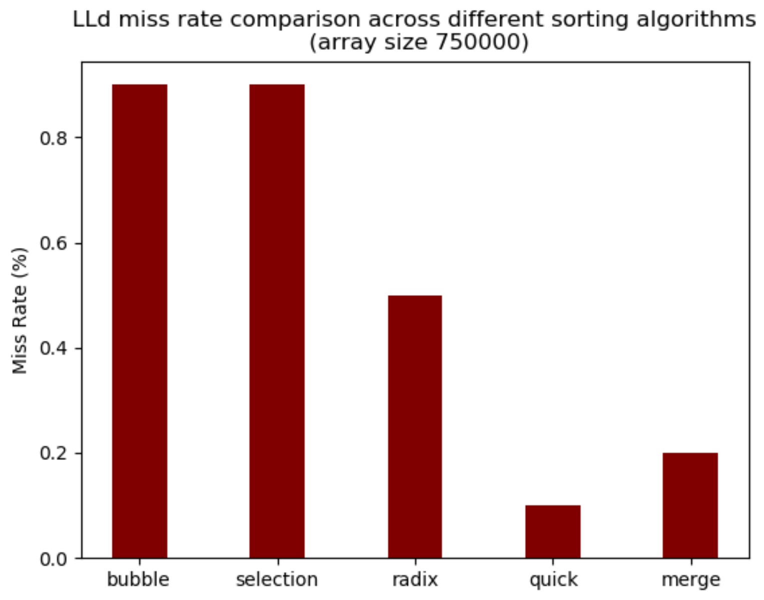

Figure 7: LLd Cache Miss Rate (array size 750000)

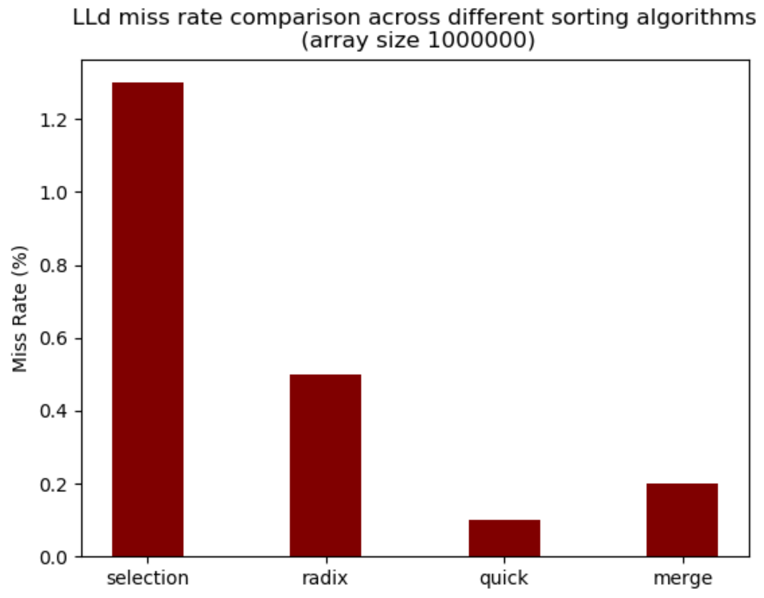

Figure 8: LLd Cache Miss Rate (array size 1000000)

4 Amount of Data Access comparison across sorting algorithms

It is observed that the number of data accesses are largest for Bubble Sort followed by Selection Sort, then by radix sort, merge sort and the least number of accesses by quick sort.

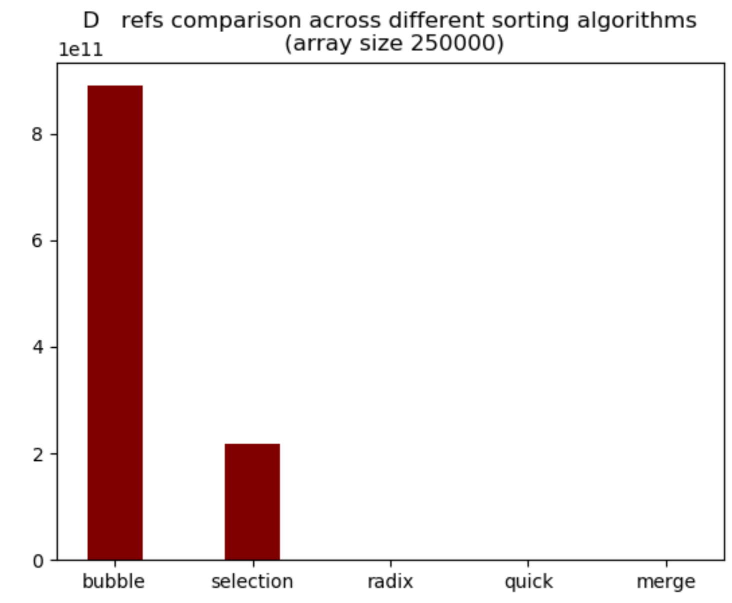

Figure 9: Data Refs (array size 250000)

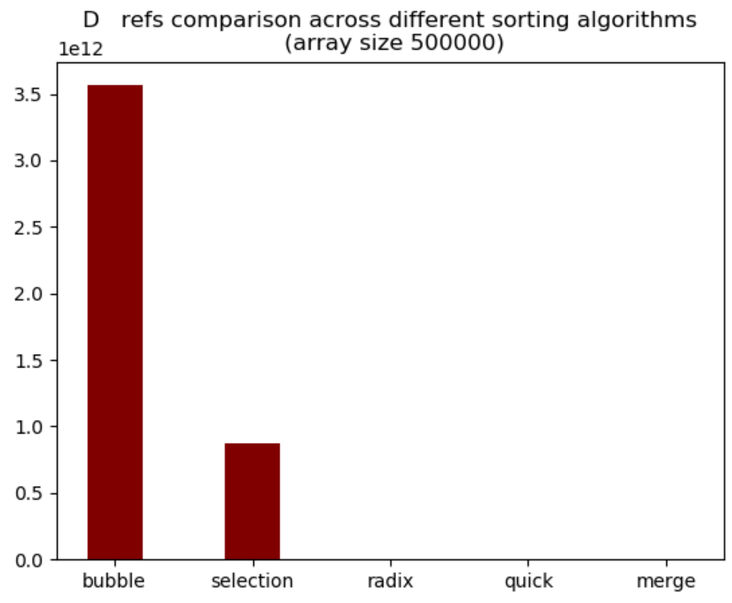

Figure 10: Data Refs (array size 500000)

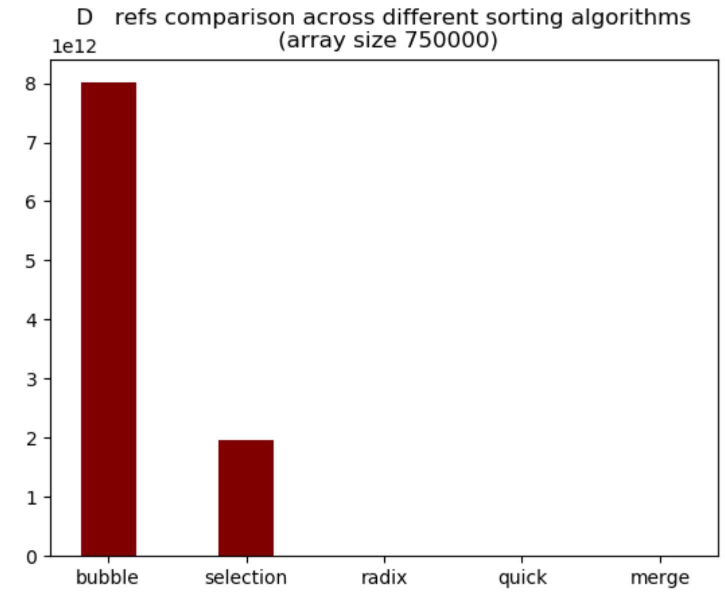

Figure 11: Data Refs (array size 750000)

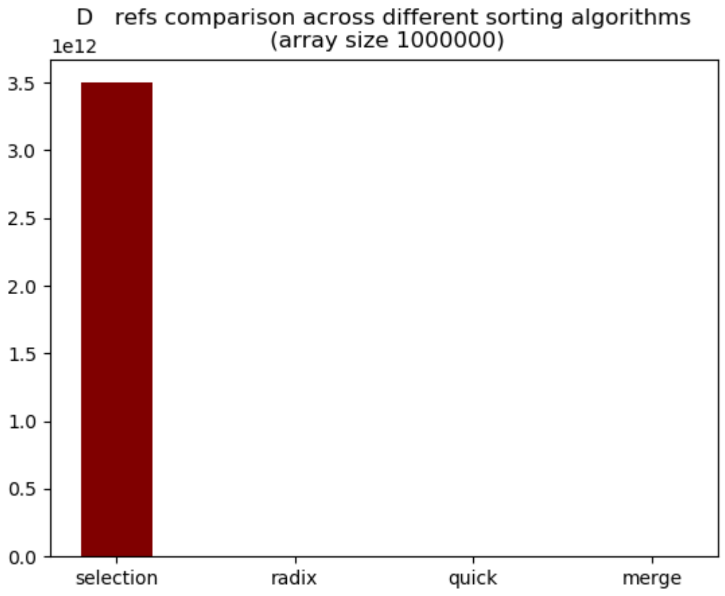

Figure 12: Data Refs (array size 1000000)

5 Comparison across cache configurations

This comparison has been illustrated only for the efficient sorting algorithms (quick, merge, radix) as bubble and selection take a long time to simulate for given array sizes. Across all these sorting algorithms we observe a common trend:

- Increasing capacity reduces miss rate. In this case L1(data) cache capacity was increased from 32768 Bytes to 65536 Bytes.
- Reducing block size increases miss rate to a large extent. In this case block size was halved from 64 Bytes to 32 Bytes for L1(data).
- Reduced Associativity does not seem to affect the cache miss rate significantly. In this case miss rate was reduced from 16-way to 4-way associative for L1(data).

5.1 Quick Sort

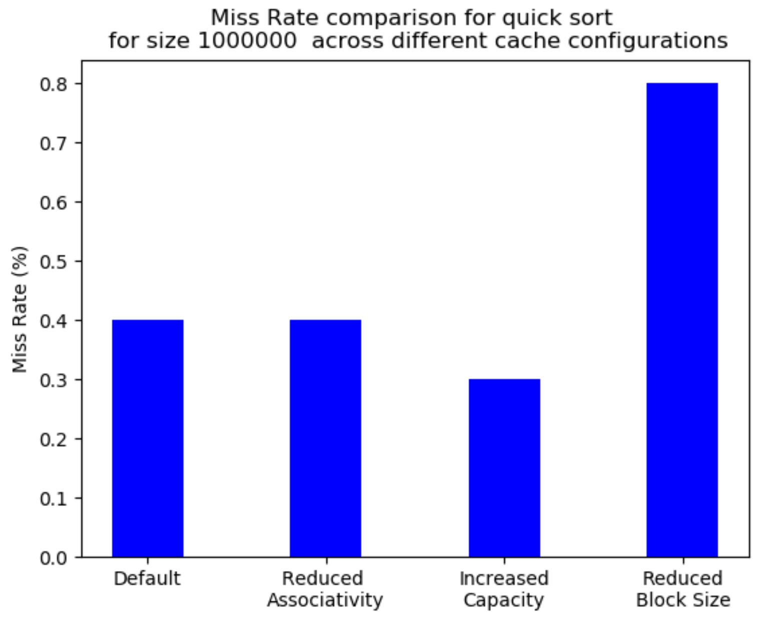

Figure 13: Cache Configuration Miss Rate comparison for Quick Sort

5.2 Merge Sort

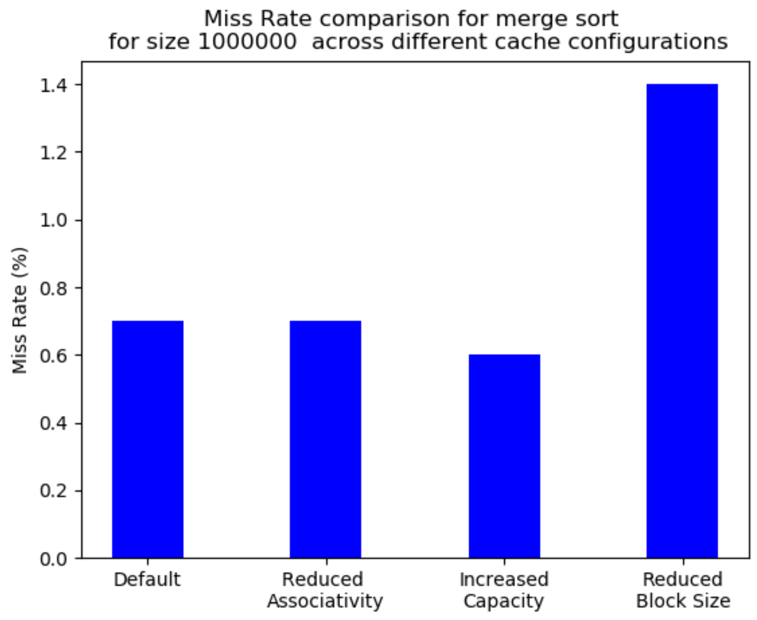

Figure 14: Cache Configuration Miss Rate comparison for Merge Sort

5.3 Radix Sort

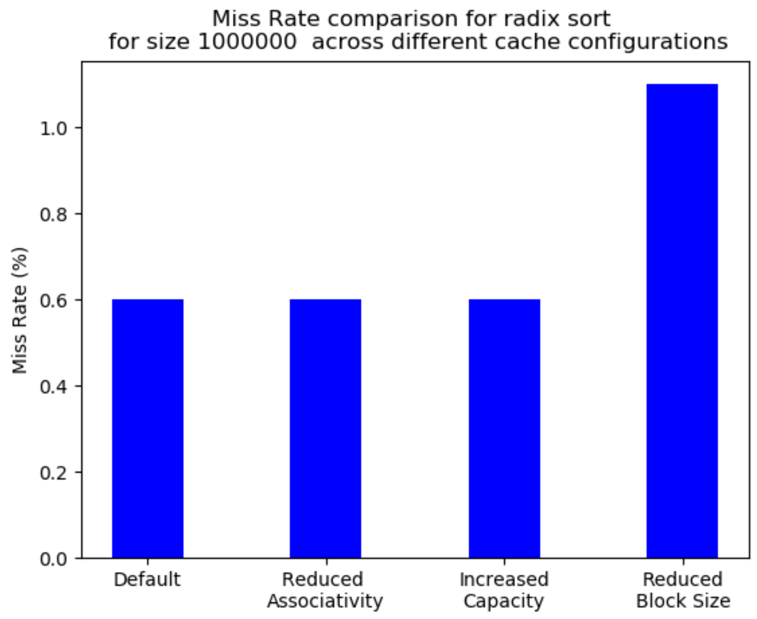

Figure 15: Cache Configuration Miss Rate comparison for Radix Sort

6 IPC Analysis

$$IPC = N/C$$

where N = Number of Instructions
and C = CPU Clock Cycles taken to execute those instructions.

This IPC analysis has been done for array of size 250000 as this is the only size for which instruction count is known for all sorting algorithm (higher sizes take a lot of time to run, but the analysis is similar)

Let us assume a hypothetical system for which every instruction fetch with Cache Hit takes 1 clock cycle, with Cache Miss on L1 takes 10 additional clock cycles, and with Cache Miss on LL takes 100 additional clock cycles. The IPC can be calculated for this system as follows:
$IPC = Ir/(Ir + 10 * L1m + 100 * LLm)$

6.1 Bubble Sort

$Ir = 1763477342587$
$L1m = I1m + D1m = 7796561012 + 1076 = 7796562088$
$LLm = 35147$
Hence we get,
$IPC = 0.9576$

6.2 Selection Sort

$Ir = 375063674410$
$L1m = I1m + D1m = 3905915932 + 1077 = 3905917009$
$LLm = 35147$
Hence we get,
$IPC = 0.9056$

6.3 Quick Sort

$Ir = 235767775$
$L1m = I1m + D1m = 1083 + 470673 = 471756$
$LLm = 35152$
Hence we get,
$IPC = 0.9662592$

6.4 Radix Sort

$Ir = 1649951043$
$L1m = I1m + D1m = 2836689 + 1132 = 2837821$
$LLm = 98066$
Hence we get,
$IPC = 0.97738$

6.5 Merge Sort

$Ir = 300150468$
$L1m = I1m + D1m = 1121 + 974334 = 975455$
$LLm = 87418$
Hence we get,
$IPC = 0.94195$

7 Observations

- Quick Sort is the most cache friendly algorithm out of all. The reason is that it is an in-place algorithm. Unlike merge/radix sort it does not involve dynamic memory allocation thus preserving locality in memory access .
- Increasing capacity reduces miss rate, which is trivial as we have to access the higher level caches/memory for lesser data.
- Reducing block size increases miss rate to a large extent.
- Reduced Associativity does not seem to affect the cache miss rate significantly. But in general it does tend to increase the miss rate.
- Amongst the fast sorting algorithms the number of memory accesses is the highest in radix sort as book keeping of the count of the keys have to be done while executing counting sort as a part of the radix sort algorithm. This is followed by merge sort which has to allocate extra memory while performing the merge operation. The least number of memory accesses is in Quick Sort as it is an inplace sorting algorithm and does not require extra memory.
- The LLd miss rate is negligible in bubble and selection sort for small array size. This is because these algorithms repeatedly access the same memory again and again. Hence, data if not found in L1 typically hits the Last Level Cache. For Quick Sort, since it is an inplace sorting algorithm it has a lower Last Level cache miss rate than radix and merge sort which require extra memory allocations. However, for large array sizes bubble sort and selection sort have poorer LLd miss rate compared to the other sorting algorithms.