Name: Joseph Hany
ID: 900182870

Assignment 6 Report

Before reporting the results of the experiments, we should have a clear definition for Spatial Locality and Temporal Locality:

Spatial Locality means that If an item is referenced, nearby items will tend to be referenced soon.
Temporal Locality means that specific data is likely to be reused over a relatively small time duration.

- Throughout the experiments in this report, we would hold some configuration variables constant in the program because we will assume that we are using a specific computer with specific capabilities. The following variables will be held constant:

  Cache-read-cycles = 2
  Cache-write-cycles = 4
  Memory-read-cycles = 8
  Memory-write-cycles = 16

- Variables that will be changed:

  - **cache mode**
  - **Cache size**
  - **Line size**

In each experiment**,** we will be changing the **Line Size** values (with **512, 1024, 2048, 4096** in each test) while leaving all the other configuration parameters constant, namely **Cache Line = (512, 1024, 2048, 4096,8192,16384 we pick on of these values during each different set of trials)** and **Cache Mode = (wt, wt we pick one of these values during each different set of trials)** to test the effect of changing the **Line Size**, **Cache Size, and Cache mode** on the **Miss Rate** with the rest of the configuration parameters constant**.**

- The rest of the output values will be considered as the affected variables, namely:

- **Total misses**
- **Total hits**
- **Miss Rate**
- **Memory Read Access Attempts**

- **Memory Write Access Attempts**
- **Total Memory Access Attempts**
- **Memory writes**
- **Memory reads**
- **Total # of Cycles for Cache Reads**
- **Total # of Cycles for Cache Writes**
- **Total # of Cycles for Cache access**
- **Total # of Cycles for Memory Reads**
- **Total # of Cycles for Memory Writes**
- **Total # of Cycles for Memory access**

There were two categories of experiments that were performed:

a) Spatial Locality Experiments:

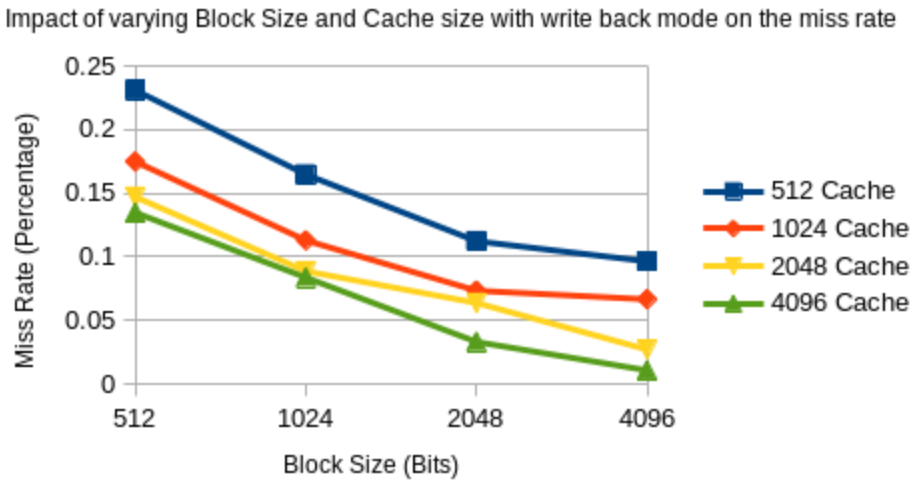
1) **<u>Horizontal Array Access Experiment</u> (**with two different cache modes) **:**

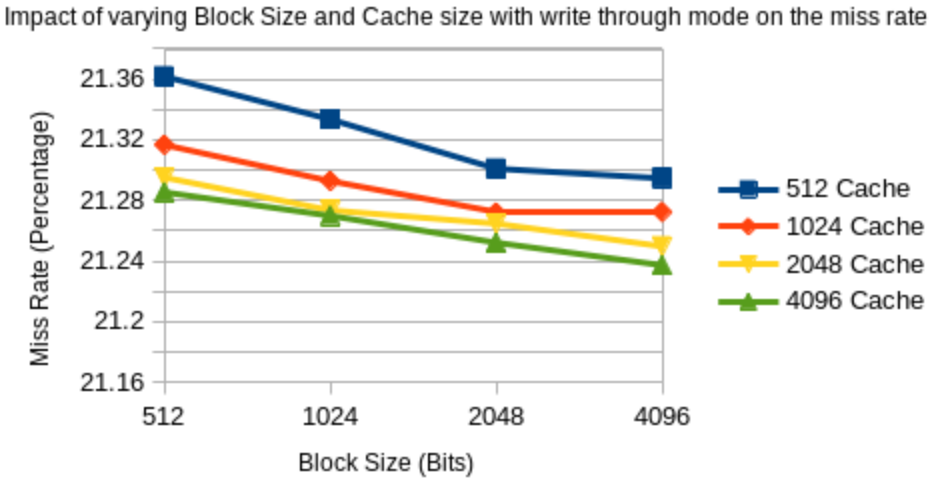
In this experiment we are mainly investigating the difference between wt and wb.

+ **Cpp File content:**

```
long row_sum (int ** array, long size)
{
        long s = 0;
        for(long i=0;i <size;i++)
                for ( long j= 0 ; j < size ; j++)
                        s += array[i][j];
        return s;
}
```

+ **Results:**





+ **Observation of the results (after running the simulation):**

In this experiment, we can make 2 distinct observations:

- Using Write through mode dramatically increases the miss rate in all different sized caches. To illustrate, write through mode increases the miss rate with about 95 times of the miss rate when we use write back mode.

- Increasing the block size in all different sized caches and different cache modes always leads to the decrease of the miss rate.

+ **Reason behind the results:**

- First, using write through mode increases the miss rate because by definition of write through mode data is simultaneously mirrored from cache to memory to achieve coherency. However, such technique causes a tremendous overhead, where every write to the cache means that there is a write to the memory. Writing to the memory is much slower than writing to the cache. Therefore, it is logical to have such dramatic miss rate difference between using write through and write back modes. Besides, write back mode only updates the memory if the cache block has been updated and the M bit has been set which makes it much faster in all cases than write through technique.

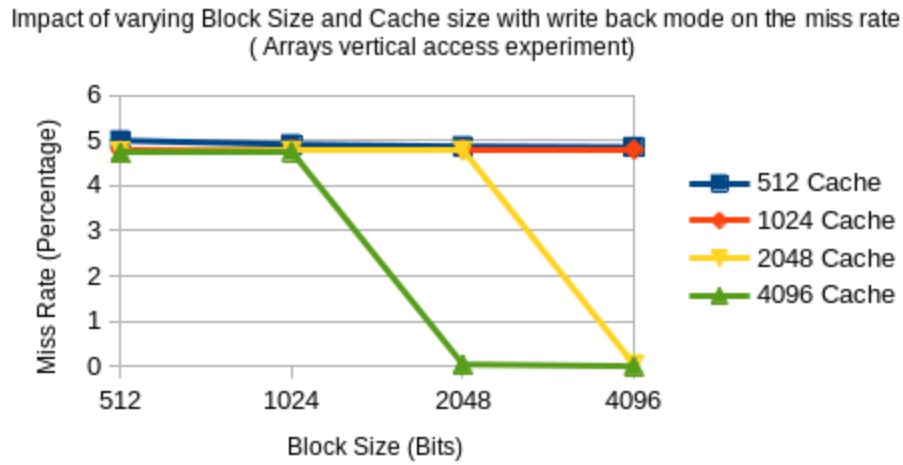
- Second, increasing the block size means that there can be more elements of the array (after being fetched from the memory) will be found in the same cache block which will consequently decrease the number of misses and increase the number of hits. In other words, we don't need to re-fetch the rest of the array elements from the memory. However, most of the elements (if not all, depending in the block size) will be found in the same block with the same tag.

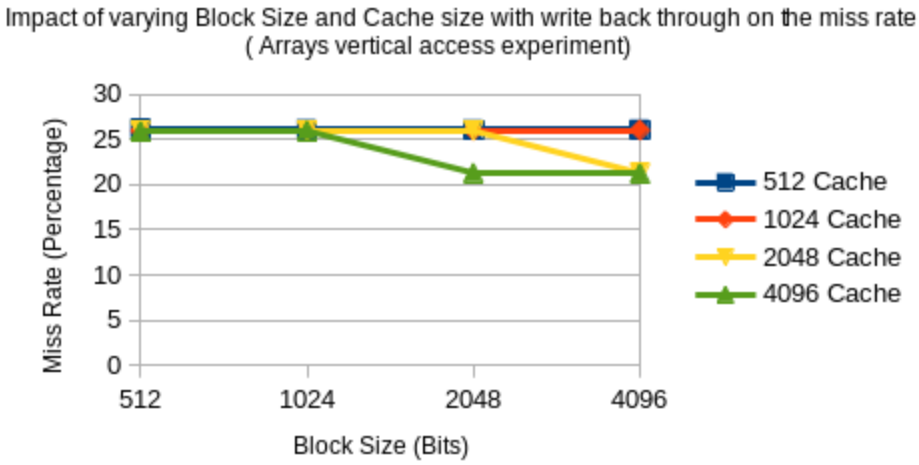
2) <u>Vertical Array Access Experiment</u>:

+ Cpp File content:

```
long row_sum (int ** array, long size)
{
        long s = 0;

        for ( long j= 0 ; j < size ; j++)
                for(long i=0;i <size;i++)
                        s += array[i][j];
        return s;
}
```

+ Results:





+ **Observation of the results (after running the simulation):**

In this experiment, we can make 3 distinct observations:

- Comparing this experiment with experiment one results, we can clearly infer that accessing the array vertically both in wb and wt cache modes increases the miss rate in all different sized caches.

- Using write through mode increases the miss rate in all different sized caches. To illustrate, write through mode increases the miss rate with about 5 times of the miss rate when we use write back mode. However, this factor is much more less than that of experiment #1.

- Increasing the block size in all different sized caches and different cache modes does not always lead to the decrease of the miss rate. Sometimes, the miss rate stays constant throughout the program.

+ **Reason behind the results:**

- First, accessing the array vertically causes compulsory misses. In other words, every time we want to access a new element in the cache, we have to fill a whole block of cache with data from which we will use only the first element. Thus, vertical arrays access distant elements and do not make use of spatial locality. Moreover, in this experiment, the **compulsory** miss rate is equal to 1 (100%). That in return increases the miss rate.

- Second, we already discussed the idea of write through mode and how it causes a useless overhead so it is not astonishing to have wb miss rate less than that of wt in this experiment. However, the astonishing part is where the factor between wt and wb in this experiment is much more less than that of experiment #1. The reason for this is that, as discussed above, accessing the array vertically leads to a dramatic increase in reads from main memory to fill the cache with the needed data. Besides, there is a miss every time we try to fetch an element from the cache block. Thus, the difference between wt and wb miss rate is relatively low because we are obligated to read from the main memory in both cases.

- Third, increasing the block size starts to have an effect on the miss rate when more elements can fit inside the same cache block. Then and only then we can have a considerable decrease in the miss rate

3) <u>Related Arrays Merging Experiment:</u>

+ **Cpp File content:**

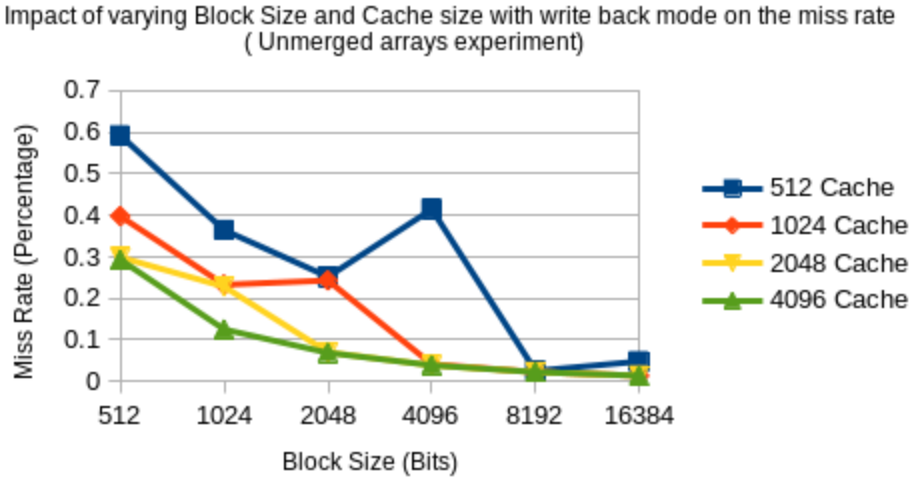
<u>Before merging:</u>

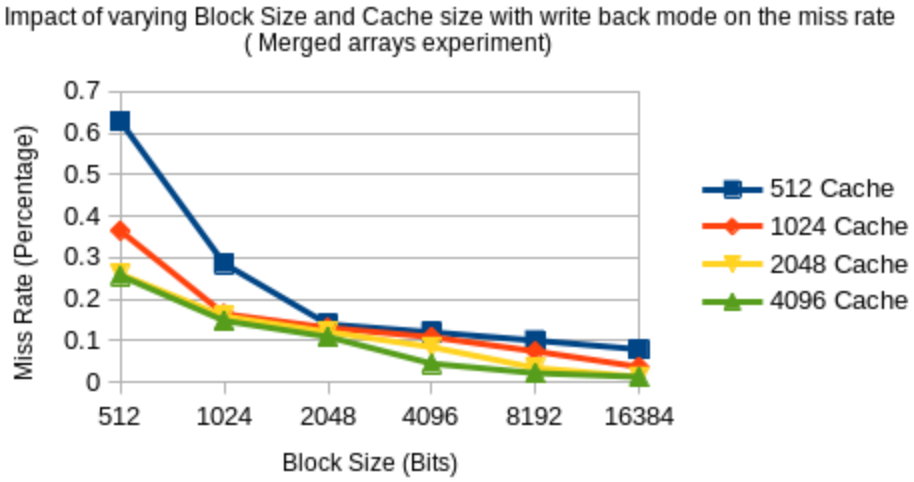
```
int * keys = (int *)calloc (size,sizeof(int));
int * values = (int *)calloc (size,sizeof(int));
```

<u>After merging:</u>

```
struct merged_elements * merged = (struct merged_elements *)calloc (size,sizeof(struct merged_elements));
```

+ **Results:**





+ **Observation of the results (after running the simulation):**

In this experiment, we can make 2 distinct observations:

- After merging the two dependent arrays together in one struct, it can be seen clearly from the graphs that instantaneously the miss rate decreases in all different sized caches.

- Increasing the block size in all different sized caches in the unmerged arrays experiment does not have a general trend. However, Increasing the block size in all different sized caches in the merged arrays experiment leads to the decrease of the miss rate.

+ **Reason behind the results:**

- First, merging the two arrays reduces the conflict misses that arise from the distance access of each of them independently throughout the loop. In addition, merging them creates spatial locality in terms of having both the key and the value after each other in the cache and we don't have to re-fetch the array of values when it is invoked by the keys array.

- Second, in the unmerged arrays experiment, when we increase the block size this may increase the distance between some elements in the key array and others in the values array so this interprets the instant rise ups in the graph of the unmerged array. However, in the merged array experiment, having larger blocks means having more keys and values beside each other in the same cache bloch, hence decreasing the miss rate.

4) <u>Matrix Multiplication Loop Interchanging:</u>

+ **Cpp File content:**

IJK:

```
int ** Matrix_Mult (int ** MatrixA,int ** MatrixB,int ** MatrixC,long size)
{
        int sum;
        for(long i=0;i <size;i++) {
                for(long j=0;j <size;j++) {
                        sum=0;
                        for(long k=0;k <size;k++) {
                                sum+=MatrixA[i][k]*MatrixB[k][j];
                        }
                        MatrixC[i][j]=sum;
                }
        }
        return MatrixC;
}
```

IKJ:

```
int ** Matrix_Mult (int ** MatrixA,int ** MatrixB,int ** MatrixC,long size)
{
        long x;
        for(long i=0;i <size;i++) {
                for(long k=0;k <size;k++) {
                        x=MatrixA[i][k];
                        for(long j=0;j <size;j++) {
                                MatrixC[i][j]+=x*MatrixB[k][j];
                        }
                }
        }
        return MatrixC;
}
```

JIK:

```
int ** Matrix_Mult (int ** MatrixA,int ** MatrixB,int ** MatrixC,long size)
{
        int sum;
        for(long j=0;j <size;j++) {
                for(long i=0;i <size;i++) {
                        sum=0;
                        for(long k=0;k <size;k++) {
                                sum+=MatrixA[i][k]*MatrixB[k][j];
                        }
                        MatrixC[i][j]=sum;
                }
        }
        return MatrixC;
}
```

JKI:

```
int ** Matrix_Mult (int ** MatrixA,int ** MatrixB,int ** MatrixC,long size)
{
        long x;
        for(long j=0;j <size;j++) {
                for(long k=0;k <size;k++) {
                        x=MatrixB[k][j];
                        for(long i=0;i <size;i++) {
                                MatrixC[i][j]+=x*MatrixA[i][k];
                        }
                }
        }
        return MatrixC;
}
```

KIJ:

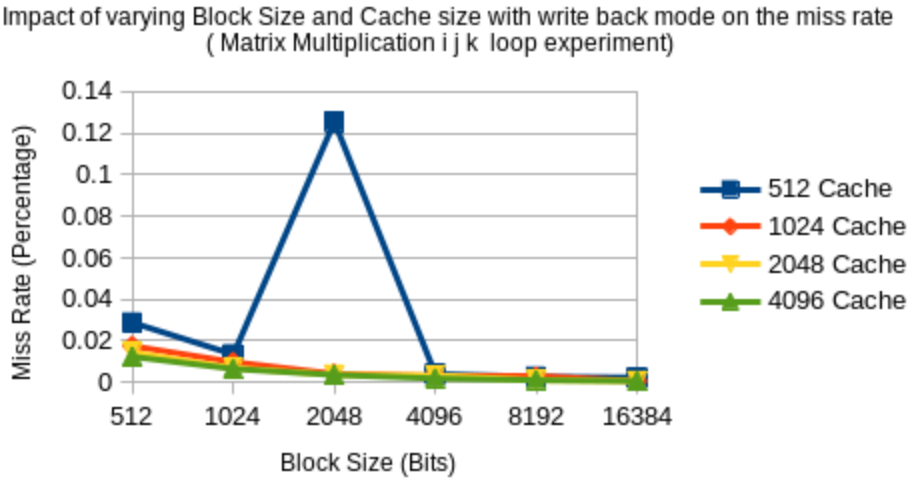
```
int ** Matrix_Mult (int ** MatrixA,int ** MatrixB,int ** MatrixC,long size)
{
        long x;
        for(long k=0;k <size;k++) {
                for(long i=0;i <size;i++) {
                        x=MatrixA[i][k];
                        for(long j=0;j <size;j++) {
                                MatrixC[i][j]+=x*MatrixB[k][j];
                        }
                }
        }
        return MatrixC;
}
```

KJI:

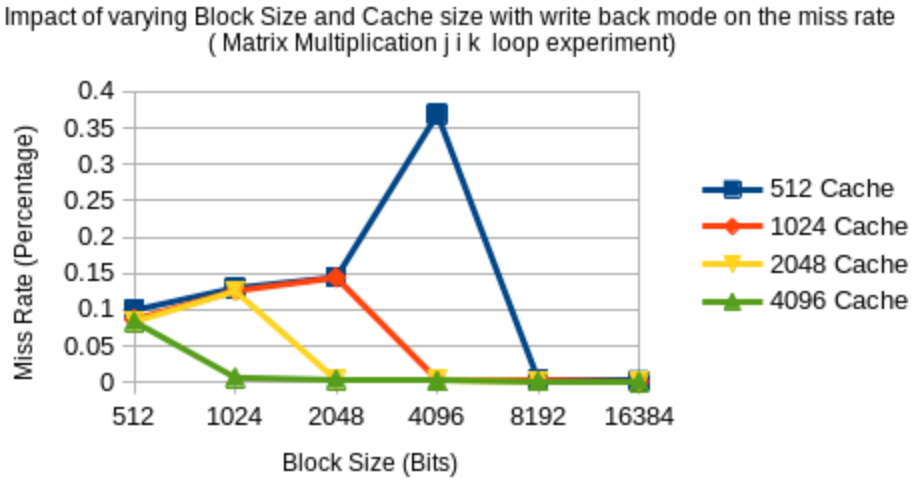
```
int ** Matrix_Mult (int ** MatrixA,int ** MatrixB,int ** MatrixC,long size)
{
        long x;
        for(long k=0;k <size;k++) {
                for(long j=0;j <size;j++) {
                        x=MatrixB[k][j];
                        for(long i=0;i <size;i++) {
                                MatrixC[i][j]+=x*MatrixA[i][k];
                        }
                }
        }
        return MatrixC;
}
```

+ **Results:**

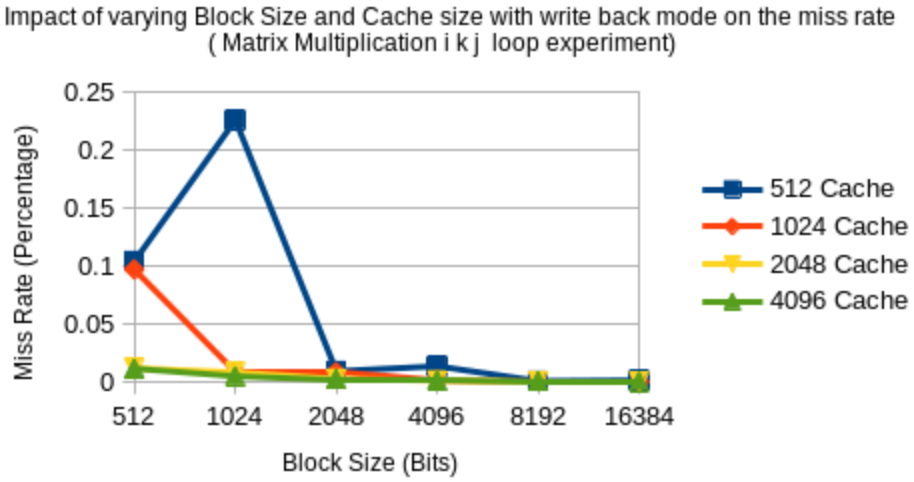
IJK loop:



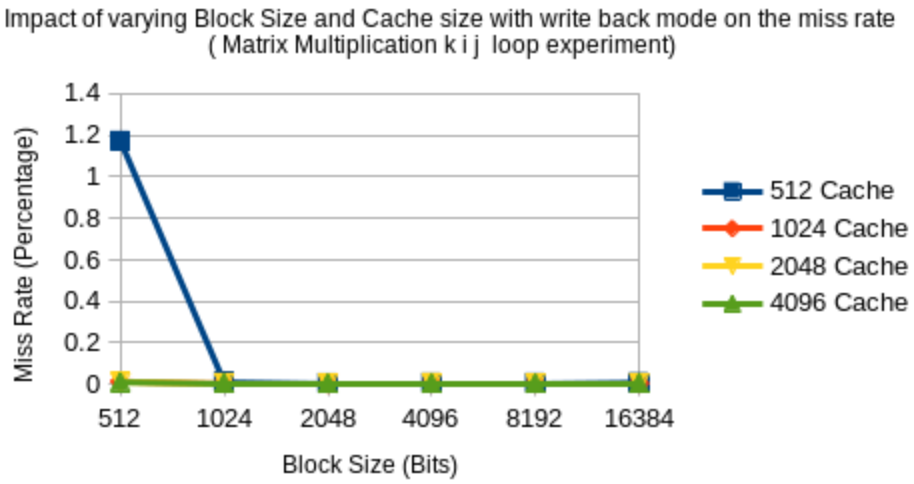
JIK loop:



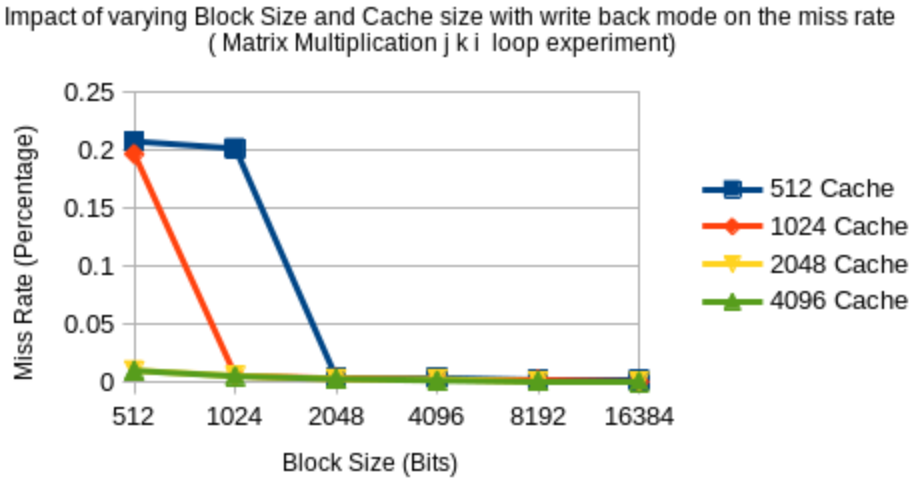
IKJ loop:



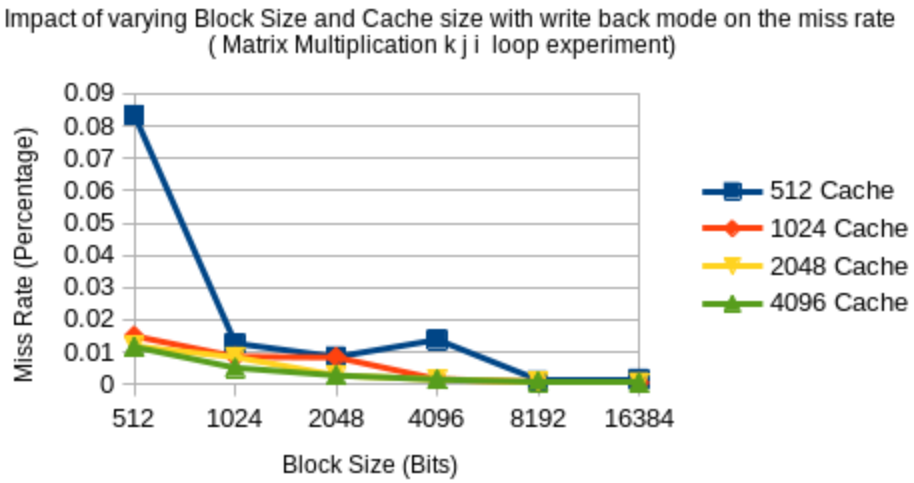
KIJ loop:



JKI loop:



KJI loop:



+ **Observation of the results (after running the simulation):**

In this experiment, we can make 1 distinct observations:

- If we sum up all the different sized caches overall miss rate, we will find that Ikj loop & jik loop have the least overall miss rate. Afterwards, ijk loop & jik loop have the second least overall miss rate. Finally, jki loop & kji loop have the highest overall miss rate of all the combinations.

+ **Reason behind the results:**

- Performing cache analysis on each loop variable combinations, we get that in both loop combinations ijk and jik, we face row accessing caused misses in matrix A and vertical access caused misses in matrix B. Second, in both loop combinations kij and ikj we face row accessing caused misses in both matrices B and C. Third, in both loop combinations jki and kji we face column accessing caused misses in both matrices A and C matrices. Therefore, we can recognize that two array vertical accessing caused misses should have the highest miss rate, while two array horizontal accessing caused misses should have the lowest miss rate.

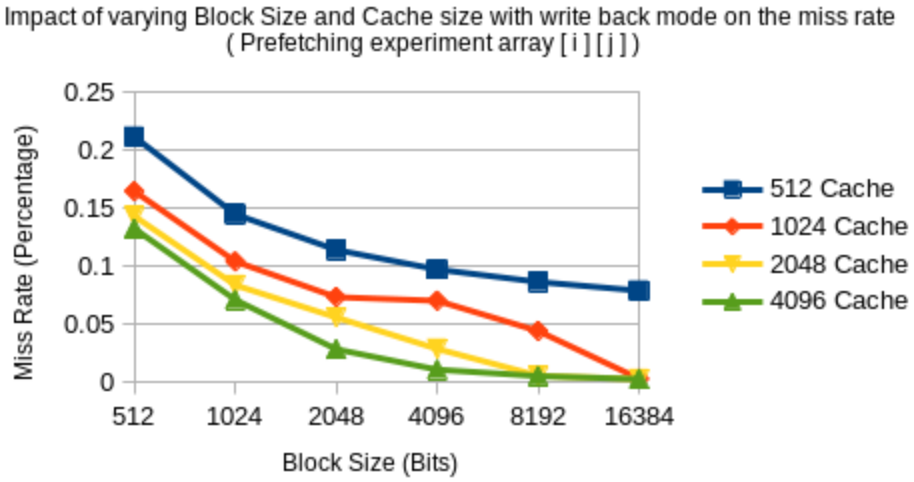
5) <u>Prefetching Experiment:</u>

+ **Cpp File content:**

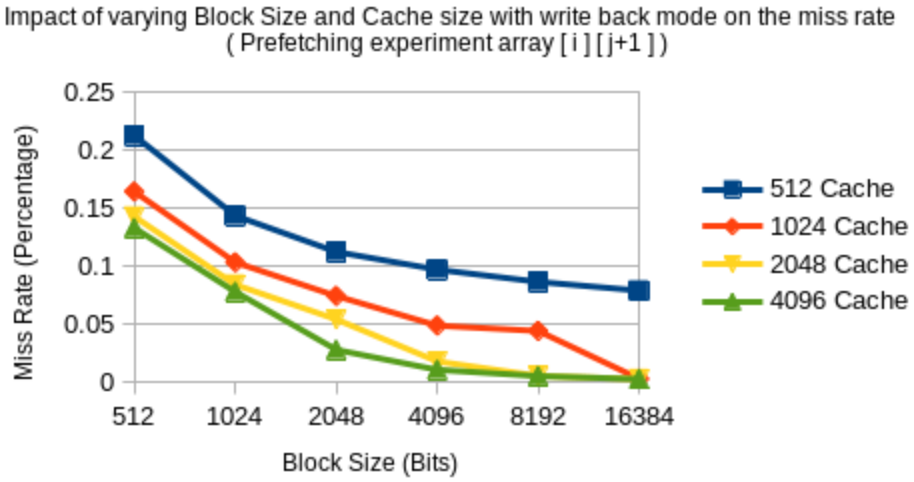
```
for(long i=0;i <size;i++) {
        for ( long j= 0 ; j < size ; j++){
                array[i][j+4];
                s += array[i][j];
        }
}
```

+ **Results:**

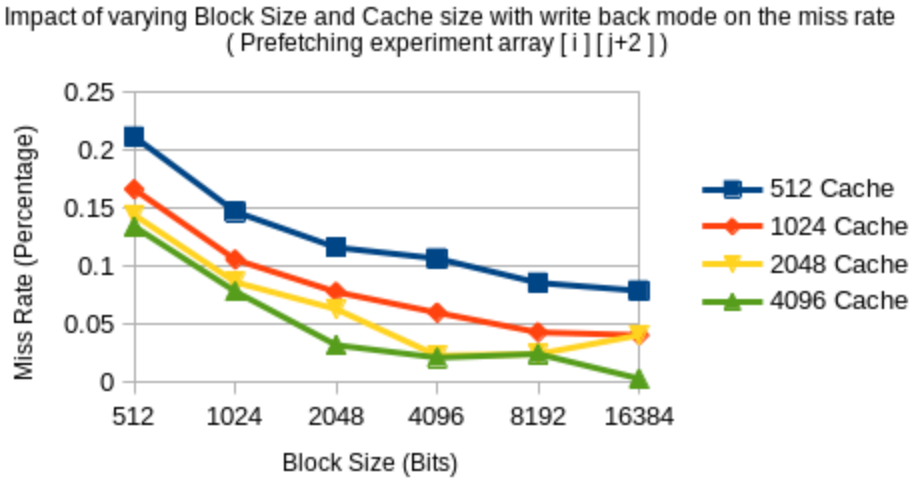
J+0:



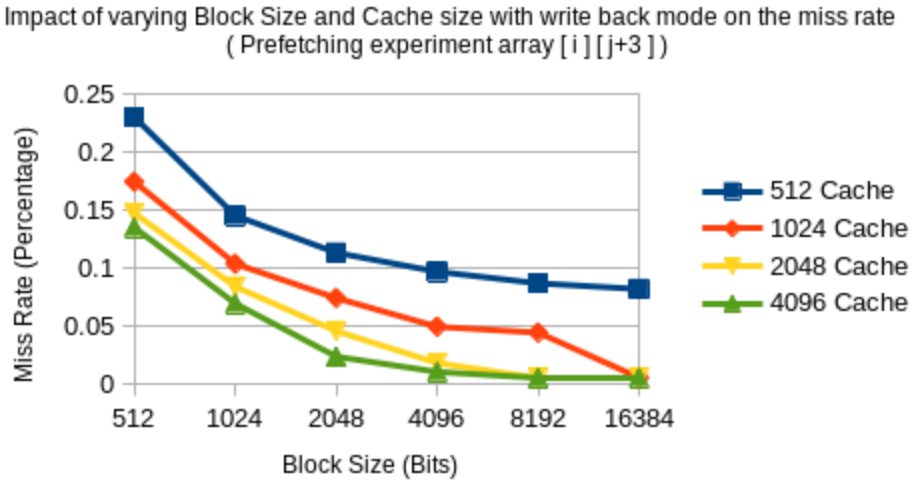
J+1:



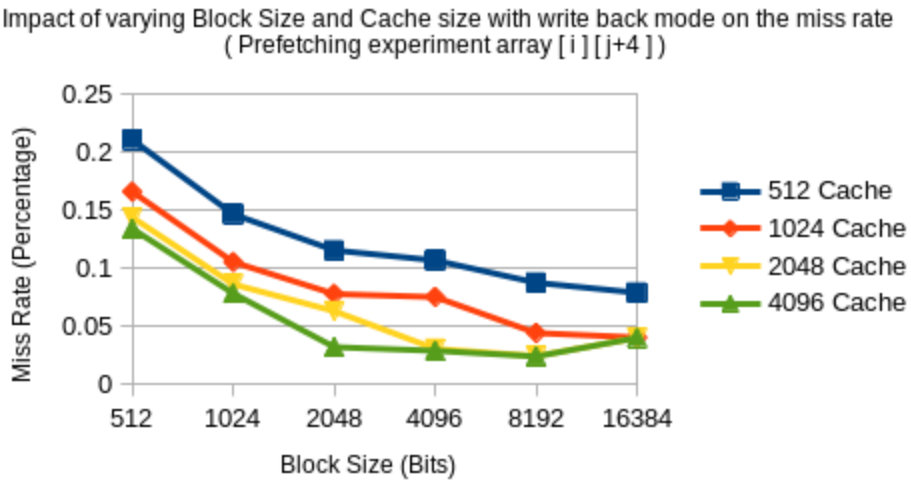
J+2:


Impact of varying Block Size and Cache size with write back mode on the miss rate
(Prefetching experiment array [i] [j+2])
Miss Rate (Percentage)
0.25
0.2
0.15
0.1
0.05
0
512
1024
2048
4096
8192
16384
Block Size (Bits)
512 Cache
1024 Cache
2048 Cache
4096 Cache

J+3:


Impact of varying Block Size and Cache size with write back mode on the miss rate
(Prefetching experiment array [i] [j+3])
Miss Rate (Percentage)
0.25
0.2
0.15
0.1
0.05
0
512
1024
2048
4096
8192
16384
Block Size (Bits)
512 Cache
1024 Cache
2048 Cache
4096 Cache

J+4:



+ **Observation of the results (after running the simulation):**

In this experiment, we can make 1 distinct observations:

- The only prefetching index that considerably decreases the miss rate is “array [i][j+1]” while the other offsets generally increase the miss rate.

+ **Reason behind the results:**

- First, figuring out the exact offset that optimizes the data prefetching is <u>not</u> constant for all computers as this depends on the clock rate and hardware capabilities of each computer. However, in our case, array [i][j+1] was the only perfect offset that brought the needed data while other operations were executing. Thus, eliminating wasted time in waiting for the memory read to finnish and fetch the data in the cache. Consequently, offsets more than one resulted in cache pollution where we had data inside the cache that was not used. Besides, this cache pollution has overwritten the data that could have been used in a short time. Thus, when the right offset is figured out data prefetching is very useful in terms of spatial locality.

6) **<u>Loop Size Increasing (increasing size of data being processed) experiment:</u>**

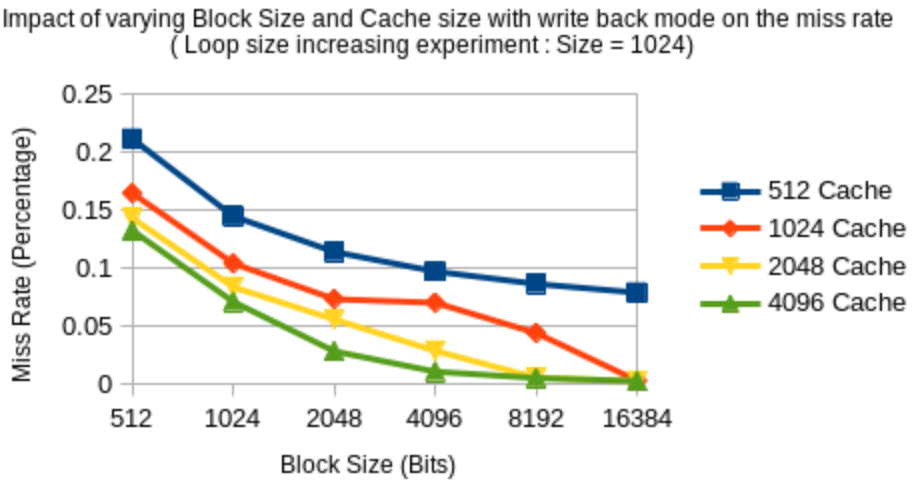
+ **Cpp File content:**

```
int size=1024;  int size=1500;  int size=2048;
```

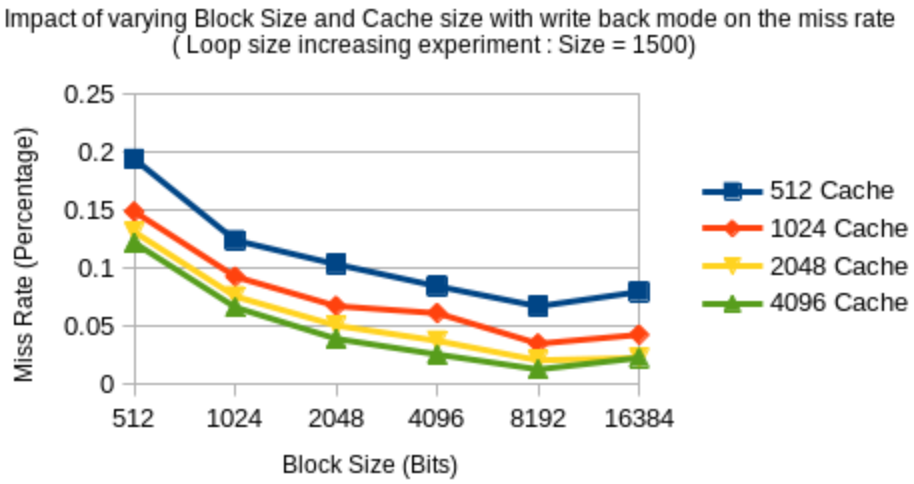
```
long row_sum (int ** array, long size)
{
        long s = 0;

        for(long i=0;i <size;i++) {
                for ( long j= 0 ; j < size ; j++){
                        s += array[i][j];
                }
        }
        return s;
}
```

+ **Results:**

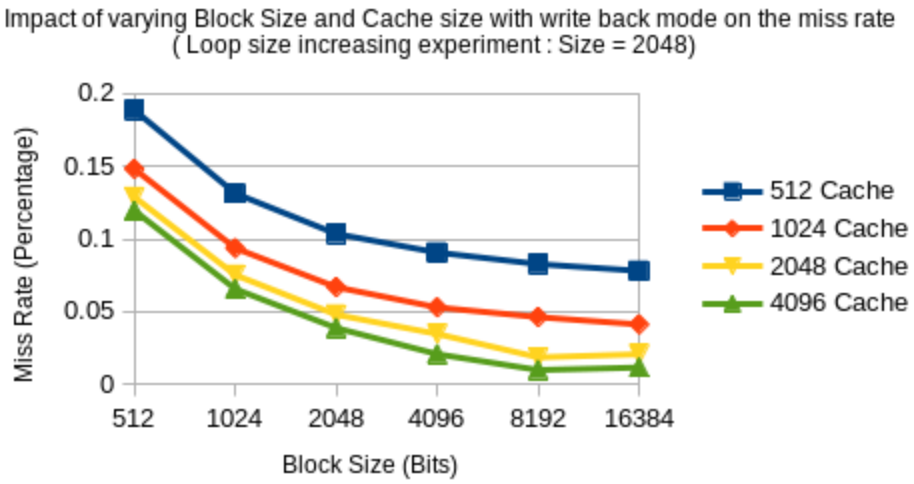
1024:



1500:



2048:



+ **Observation of the results (after running the simulation):**
  - The miss rate decreases a little bit (with no significant amount) as we increase the loop size in all different cases

+ **Reason behind the results:**
  - As we increase the koop size that does not affect how our cache acts as it does not have a significant impact on our data and graphs.

b) Temporal Locality Experiments:

1) **Loop Fusion Experiment:**

+ **Cpp File content:**

No Fusion:

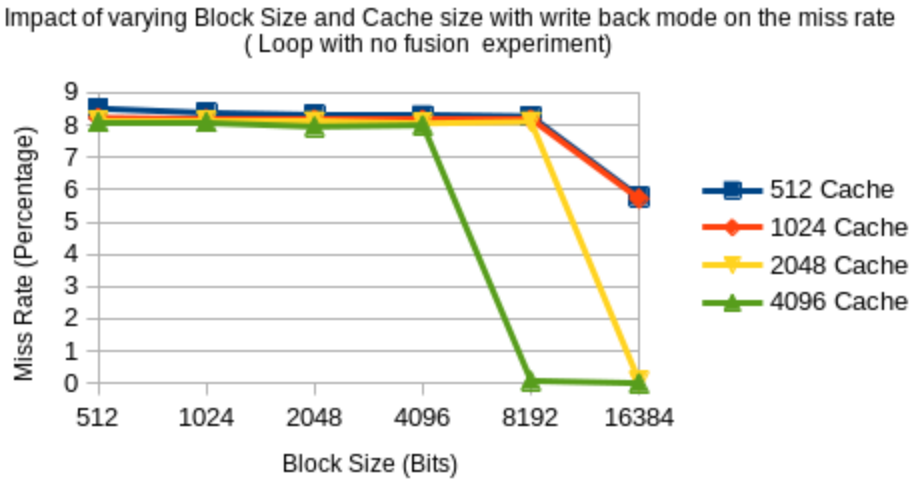
```
for ( long j= 0 ; j < size ; j++){
        for(long i=0;i <size;i++){
                MatrixB[i][j]=2*MatrixA[i][j];
        }
}
for ( long j= 0 ; j < size ; j++){
        for(long i=0;i <size;i++){
                MatrixC[i][j]=MatrixB[i][j]+MatrixD[i][j];
        }
}
```

With Fusion:

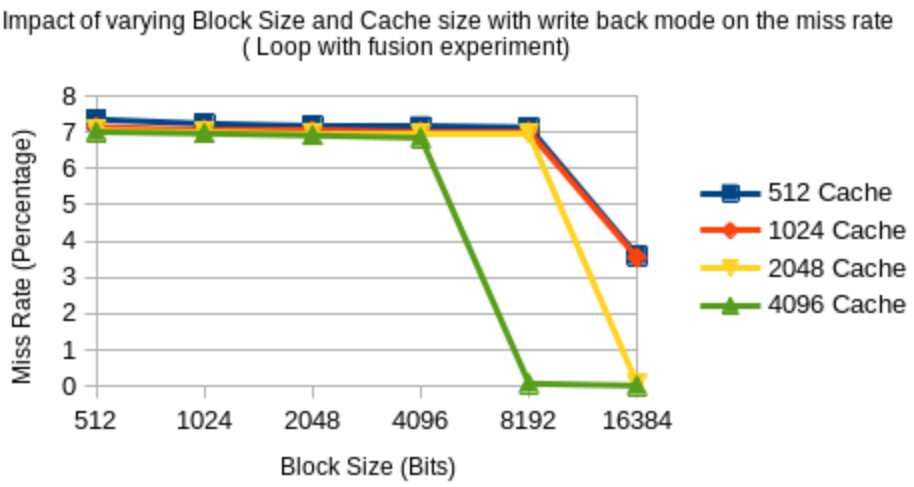
```
for ( long j= 0 ; j < size ; j++){
        for(long i=0;i <size;i++){
                MatrixB[i][j]=2*MatrixA[i][j];
                MatrixC[i][j]=MatrixB[i][j]+MatrixD[i][j];
        }
}
```

+ **Results:**

No Fusion:



With Fusion:



+ **Observation of the results (after running the simulation):**

In this experiment, we can make 1 distinct observation:

- Fusion of the two loops that have dependent variables used inside them decreases the miss rate with a constant ratio across all different sized caches.

+ **Reason behind the results:**

- In this experiment, instead of having two misses for not finding MatrixB in the cache, we would only have one miss. Thus, creating temporal locality and decreasing the miss rate as MatrixB is frequently used so fusion would decrease the need of fetching its data rwo times from the memory to the cache.