Chuck Richardson

ECE 563 Microprocessor Architecture

ECE 563 Project 3 Cache Simulator

1. Introduction

The objective of this project was to implement a cache simulator that could mimic an L1 cache with a configurable capacity, cache line size, associativity as well as write hit/miss policies. This L1 Cache also used an LRU replacement policy for evictions. During the simulator a respective trace file was read in which contains a series of read and write operations. The students of ECE 563 were also tasked with completion of a matrix multiplication code to generate a respective trace file.

2. Results

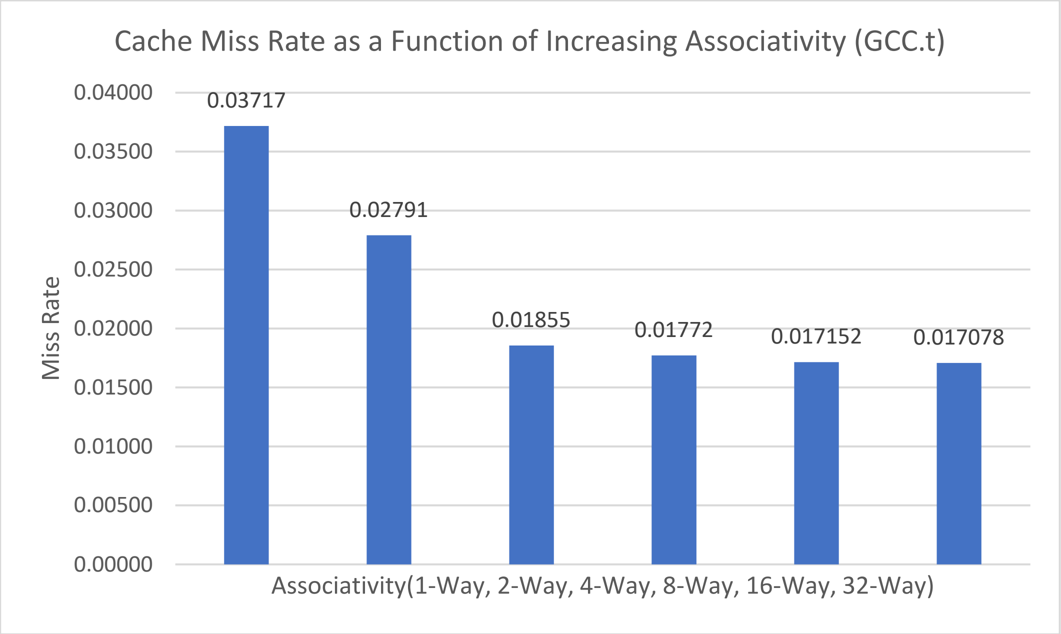
2.1. GCC Trace

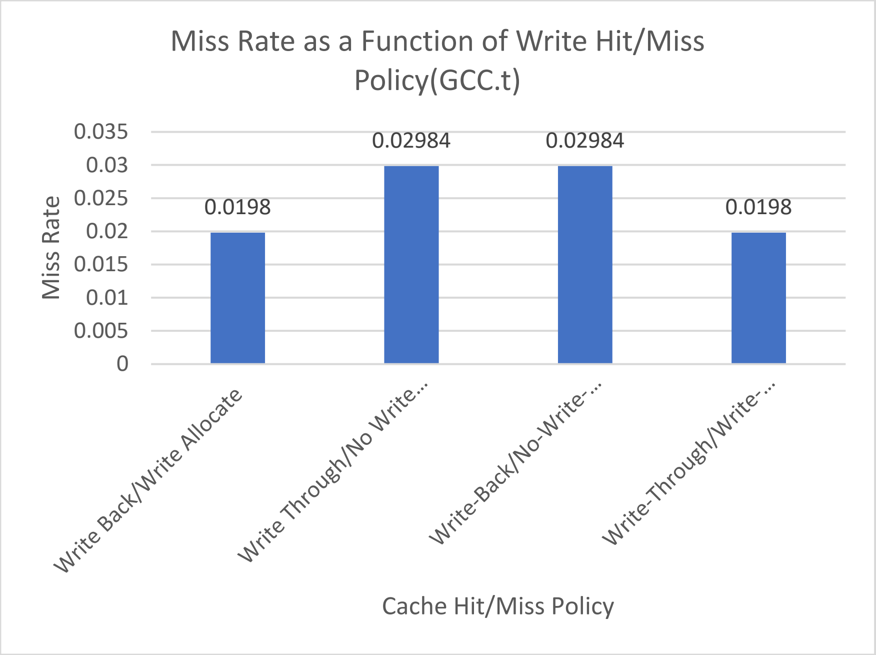
The following parameters of the cache were tested as shown in Table 1 for the GCC Trace.

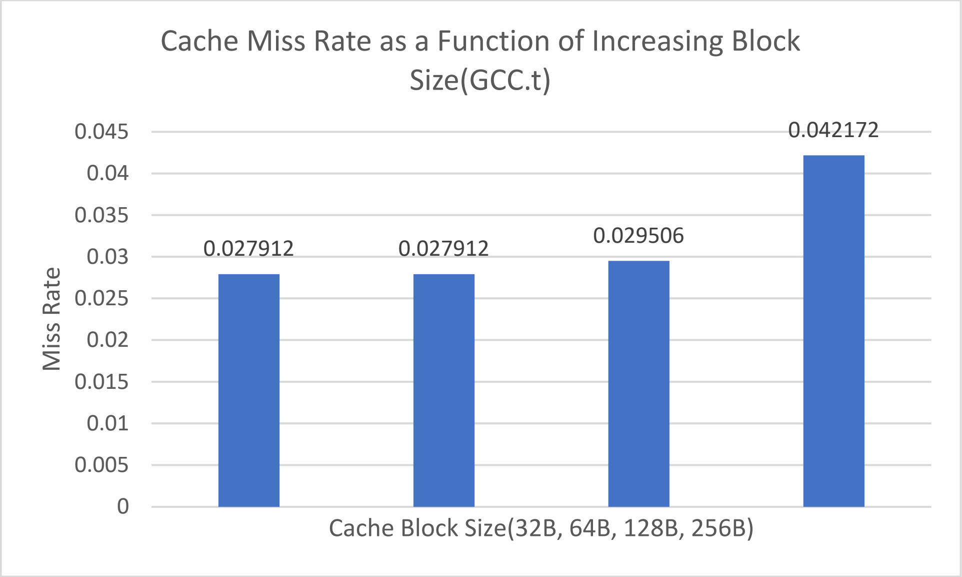
Cache Size	Cache Associativity	Block Size	Write Hit/Miss Policy
16KB	1-way	32B	Write-Back/Write-Allocate
32KB	2-way	64B	Write-Through/Write-No Allocate
64KB	4-way	128B	Write-Back/No-Write-Allocate
	16-way	256B	Write-Through/Write-Allocate

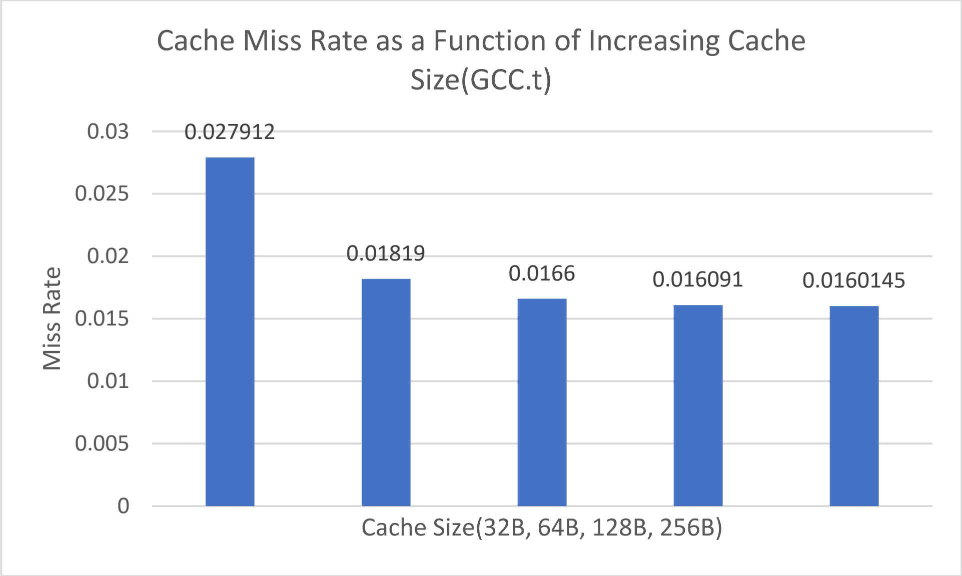
Table 1: Cache Parameters

Shown below is the Cache simulated outputs from the GCC.t trace fille.


Cache Miss Rate as a Function of Increasing Associativity (GCC.t)

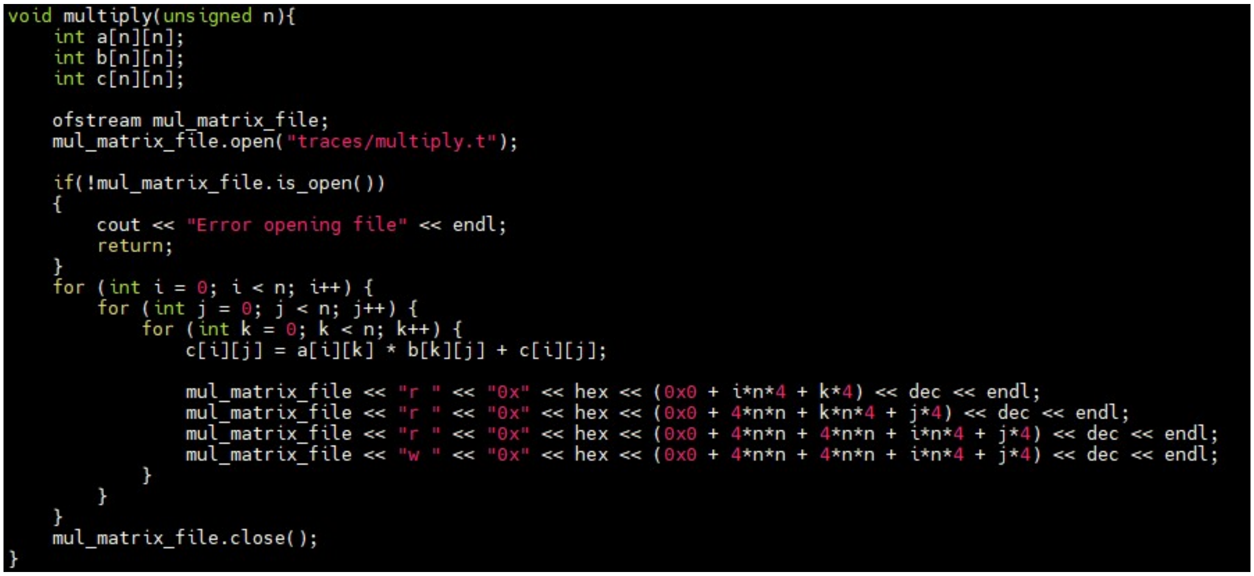

Miss Rate as a Function of Write Hit/Miss Policy(GCC.t)


Cache Miss Rate as a Function of Increasing Block Size(GCC.t)


Cache Miss Rate as a Function of Increasing Cache Size(GCC.t)

2.2. Naïve Matrix Multiplication Trace Generation

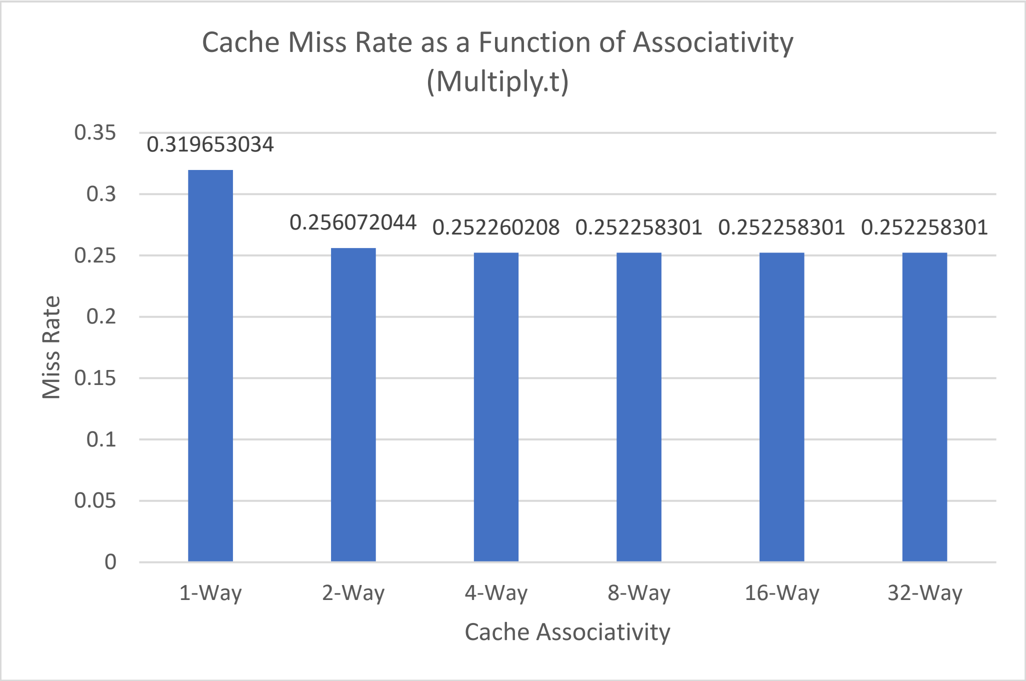
The following parameters of the cache were tested as shown in Table 1 for the Naïve Matrix Multiplication Trace. It is important to note that because array c will always be read and then written to, we will never have a write miss. This code is shown in Figure 1.

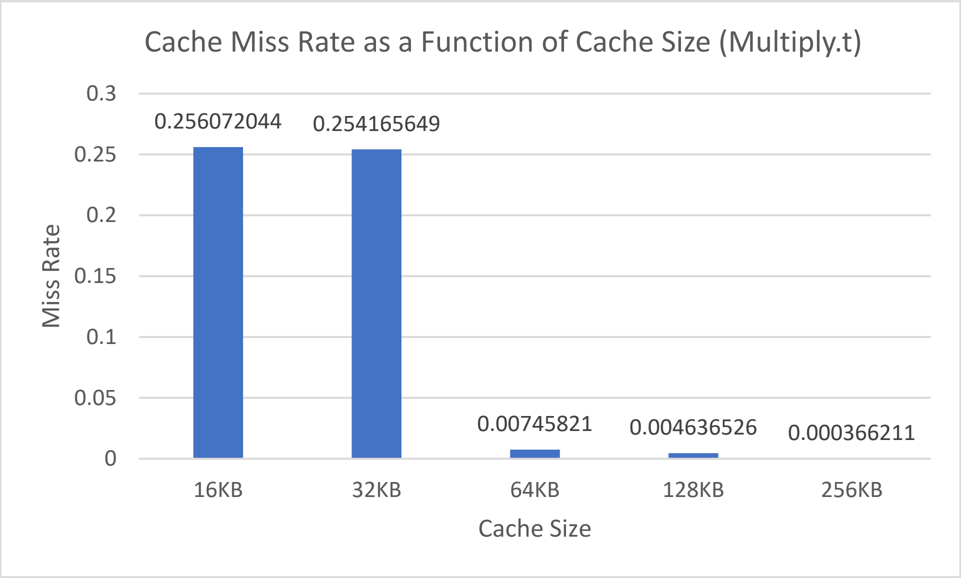


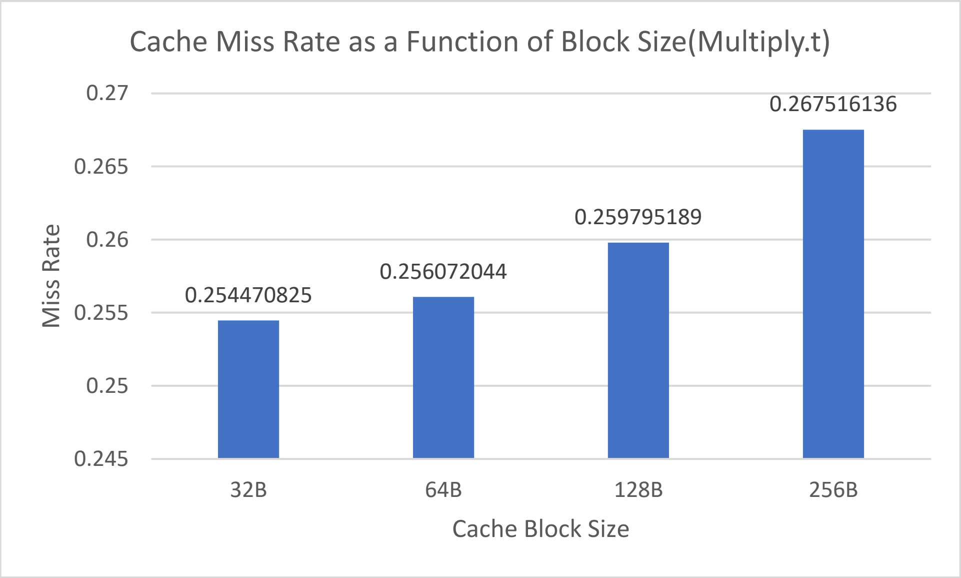
```
void multiply(unsigned n){
    int a[n][n];
    int b[n][n];
    int c[n][n];

    ofstream mul_matrix_file;
    mul_matrix_file.open("traces/multiply.t");

    if(!mul_matrix_file.is_open())
    {
        cout << "Error opening file" << endl;
        return;
    }
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            for (int k = 0; k < n; k++) {
                c[i][j] = a[i][k] * b[k][j] + c[i][j];

                mul_matrix_file << "r " << "0x" << hex << (0x0 + i*n*4 + k*4) << dec << endl;
                mul_matrix_file << "r " << "0x" << hex << (0x0 + 4*n*n + k*n*4 + j*4) << dec << endl;
                mul_matrix_file << "r " << "0x" << hex << (0x0 + 4*n*n + 4*n*n + i*n*4 + j*4) << dec << endl;
                mul_matrix_file << "w " << "0x" << hex << (0x0 + 4*n*n + 4*n*n + i*n*4 + j*4) << dec << endl;
            }
        }
    }
    mul_matrix_file.close();
}
```

Figure 1: Naive Matrix Multiplication Code

The results of the cache simulation is shown in the graphs below.

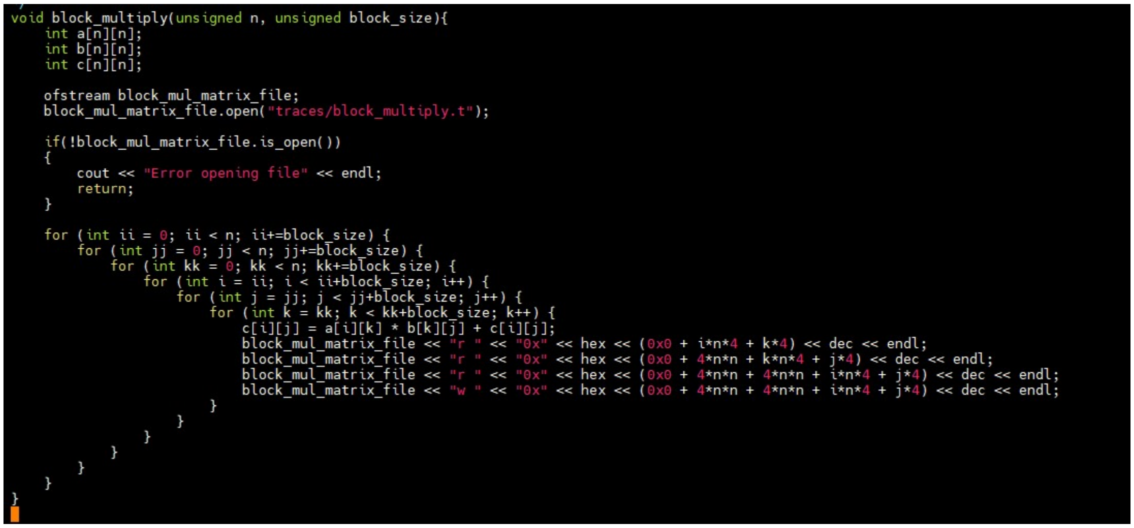

Cache Miss Rate as a Function of Associativity (Multiply.t)


Cache Miss Rate as a Function of Cache Size (Multiply.t)


Cache Miss Rate as a Function of Block Size(Multiply.t)

2.3. Cache Friendly Matrix Multiplication Trace

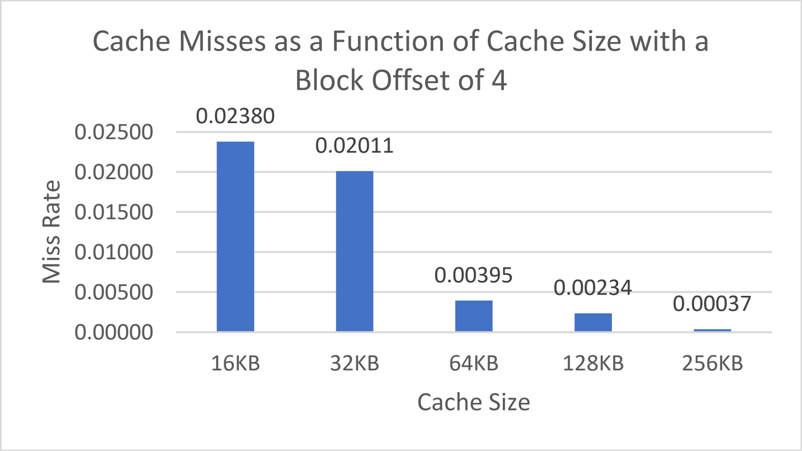
The following parameters of the cache were tested as shown in Table 1 for the Cache Friendly Matrix Multiplication. This code is shown in Figure 2. The result of the simulation is shown in the graphs below with their respective block offsets.

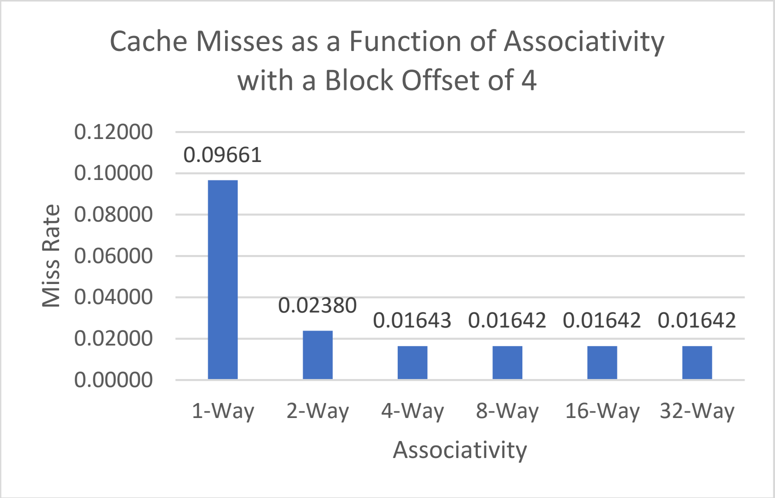


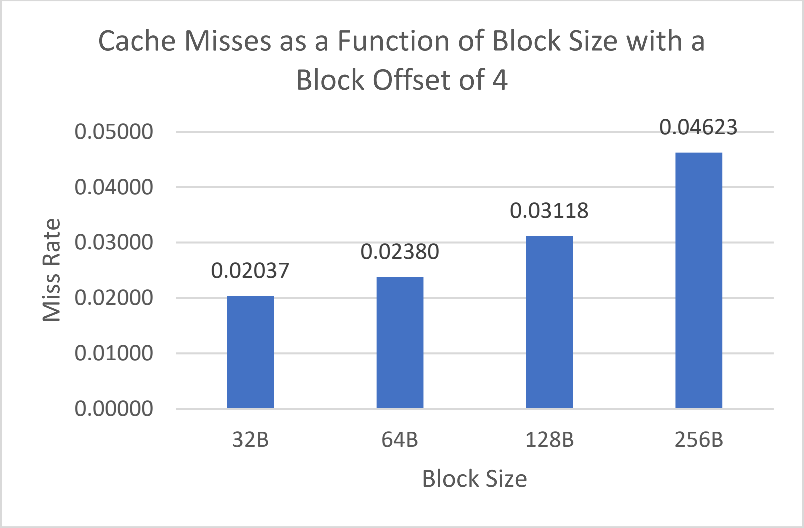
```
void block_multiply(unsigned n, unsigned block_size){
    int a[n][n];
    int b[n][n];
    int c[n][n];

    ofstream block_mul_matrix_file;
    block_mul_matrix_file.open("traces/block_multiply.t");

    if(!block_mul_matrix_file.is_open())
    {
        cout << "Error opening file" << endl;
        return;
    }

    for (int ii = 0; ii < n; ii+=block_size) {
        for (int jj = 0; jj < n; jj+=block_size) {
            for (int kk = 0; kk < n; kk+=block_size) {
                for (int i = ii; i < ii+block_size; i++) {
                    for (int j = jj; j < jj+block_size; j++) {
                        for (int k = kk; k < kk+block_size; k++) {
                            c[i][j] = a[i][k] * b[k][j] + c[i][j];
                            block_mul_matrix_file << "r " << "0x" << hex << (0x0 + i*n*4 + k*4) << dec << endl;
                            block_mul_matrix_file << "r " << "0x" << hex << (0x0 + 4*n*n + k*n*4 + j*4) << dec << endl;
                            block_mul_matrix_file << "r " << "0x" << hex << (0x0 + 4*n*n + 4*n*n + i*n*4 + j*4) << dec << endl;
                            block_mul_matrix_file << "w " << "0x" << hex << (0x0 + 4*n*n + 4*n*n + i*n*4 + j*4) << dec << endl;
                        }
                    }
                }
            }
        }
    }
}
```

Figure 2: Cache Friendly Matrix Multiplication Code

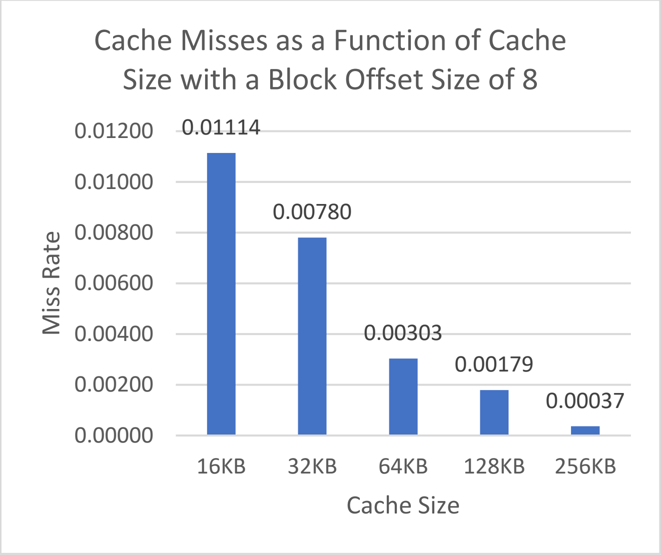
2.3.1 block_multiply Trace Simulated Cache Parameters with a Block Offset = 4

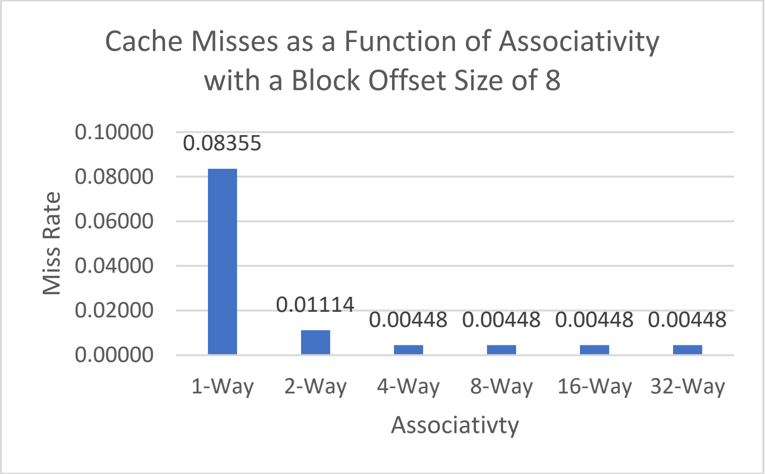

Cache Misses as a Function of Cache Size with a Block Offset of 4

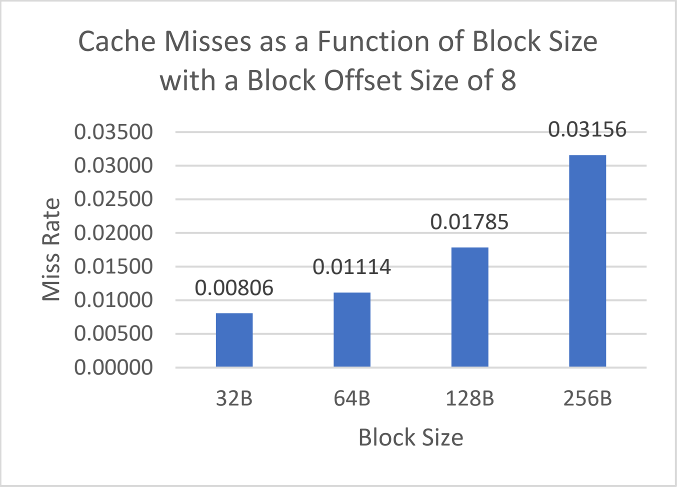

Cache Misses as a Function of Associativity with a Block Offset of 4


Cache Misses as a Function of Block Size with a Block Offset of 4

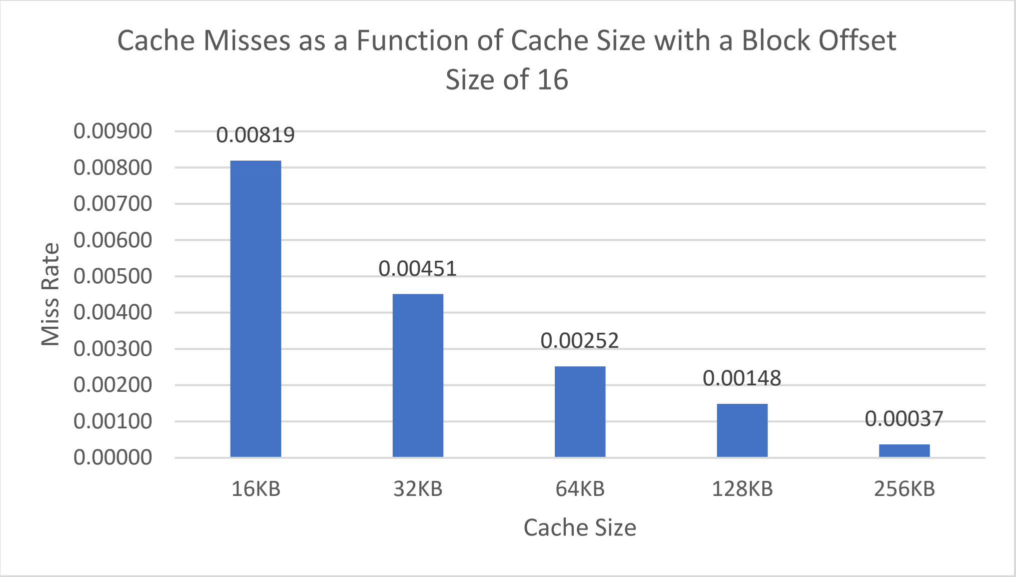
2.3.2. block_multiply Trace Simulated Cache Parameters with a Block Offset = 8

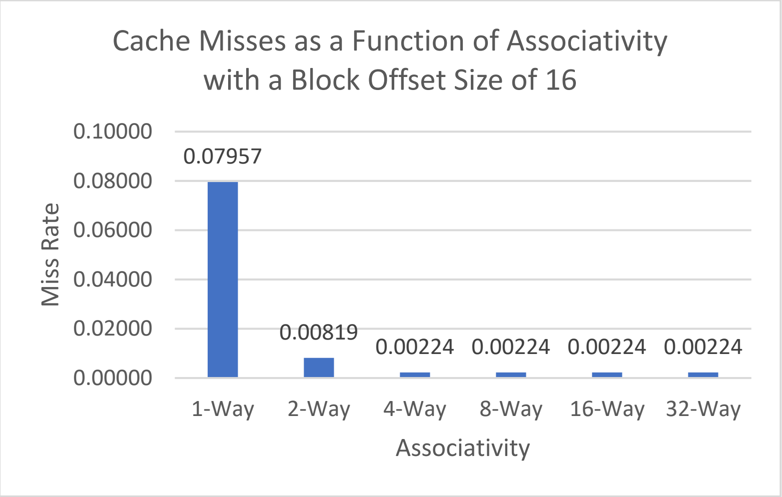


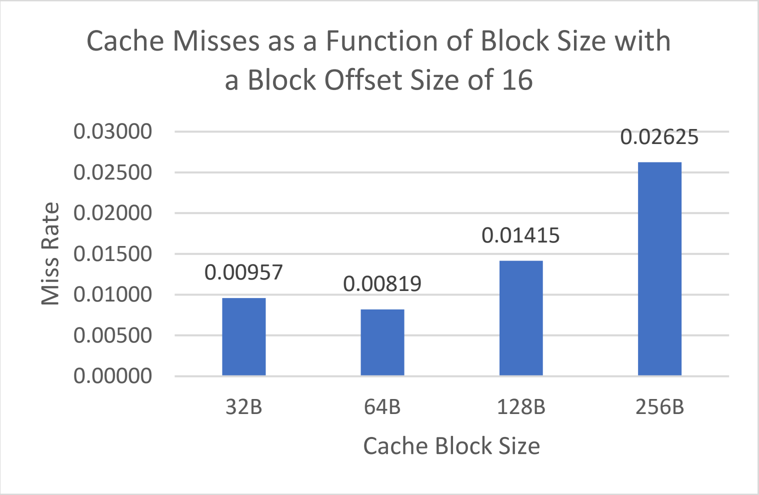




2.3.3 block_multiply Trace Simulated Cache Parameters with a Block Offset = 16







3. Discussion

Some notable conclusions in this project is that associativity and cache size are all inversely proportional to cache miss rate, while cache block size is directly proportional. For example, as cache associativity increases then the cache miss rate decreases.

Regarding the naïve cache matrix implementation one can see that the 2-way to 32-way associativity cache miss rate is almost always the same, because most of the time we will only have two variables a and b being used in the cache lines. Therefore, increasing associativity offers no beneficial results.

The cache friendly implementation is better than that of the naïve implementation because it ensures that the variables that are put into the cache are fully used prior to them being evicted. This is ensured by the lower 3 loops. In the naïve code implementation, we might evict cache blocks before they are utilized. We also notice that the cache miss rate is at a minimum when the block offset is set to 16 which is the Block Size 64/4.

4. Conclusion

In conclusion, the cache simulator worked as expected and lets us see the results of varying different cache parameters such as associativity, cache size, cache block size, and cache hit/miss policies.