# MARS Caching Evaluation Report

Author: Cristina McLaughlin

Professor: Peter-Michael Seidel

Class: EE 660

Date: December 1, 2019

## 1 Introduction

A *cache* in computer architecture is defined as memory with short access time that is used for storage of frequently or recently used instructions or data. Cache memories are small and fast SRAM-based memories that are managed automatically in hardware; they are more expensive than main memory, but still cheaper than register files. Today, caches are important because they improve the overall system throughput of computers. While processors have increased in speed, main memory and mass storage has remained slow in comparison; caching is used to close this gap by exploiting locality [1].

As an EE student I have taken several algorithms classes, and program optimization is always at the forefront. However, the focus is generally on computational and asymptotic complexity. In this project, I was interested in exploring how caching and algorithm structure can affect runtime performance. This was conducted by running two matrix programs with extreme versions of locality and observing the outcomes with the MARS simulator. After understanding locality, I tested iterative and recursive versions of the Fibonacci algorithm to see the cache performance and related the results to the perspective of computational complexity.

The following paper is formatted as follows: Section 2 contains background research on caching, Section 3 discusses the locality testing done in the MARS simulator, Section 4 analyzes the iterative and recursive versions of the Fibonacci algorithm on caching, and Section 5 contains my conclusions on this project.

## 2 Background Research

The first goal of this project was to gain a deeper understanding of cache memory, memory organization, and cache performance. Computer memory is split into a hierarchy where

smaller, faster, and costlier memory is kept closer to the processor, while larger, slower, cheaper memory is farther away. Figure 1 shows a diagram of common memory hierarchy; the fundamental idea is that each level k serves as a cache for larger slower devices at level k+1 [1]. Therefore, the register file holds words retrieved from the L1 cache and the L2 cache holds line retrieved from main memory. This creates a large pool of storage that can retain a lot of information at the bottom, but still serve data quickly to the processor at the top.

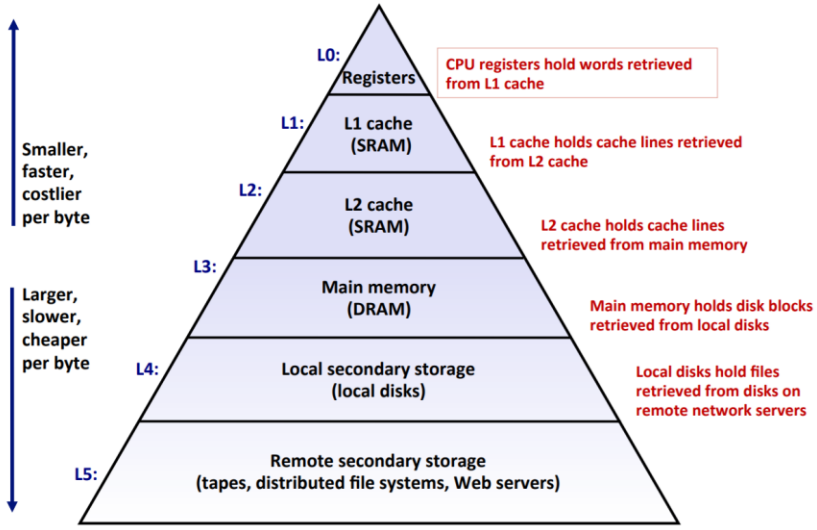



Figure 1. Memory hierarchy diagram [1].

The general concept of a cache is that when the processor needs to read or write to a location in memory, it first checks for the data entry in the cache. A *cache hit* occurs when the memory location is in the cache, and the data is then read from it. Figure 2 shows how the processor requests data, and how the block is in the cache, resulting in a hit [2]. A typical *hit rate* is from 95%-97% performance in the L1 cache in real life. *Hit time* is the time required to deliver a line in the cache to the processor, which is generally 1-2 cycles.

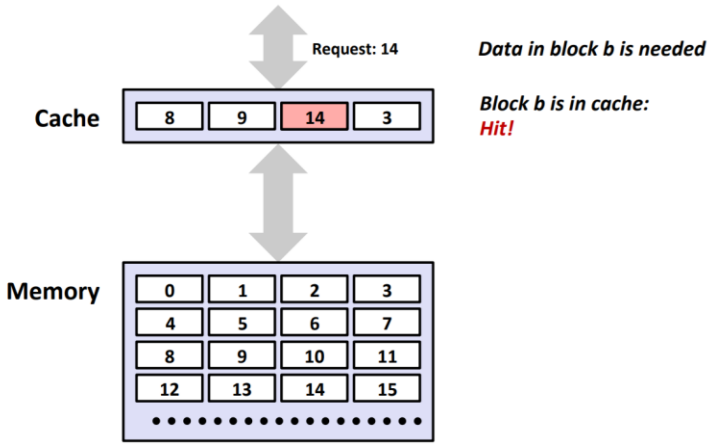



Figure 2. Diagram showing a cache hit [2].

On the other hand, a *cache miss* occurs when the processor *does not* find the memory location in the cache. If a miss occurs, the cache will allocate a new entry, then copy the data from the main memory into the cache and provide the data to the processor as shown in Figure 3 [2]. A cache miss comes with a *miss penalty*—because of the additional time required to get data from the main memory which is slow. A typical miss penalty is 50-200 cycles, but this trend is increasing [2]. This penalty is hundreds of times different from a hit time of 1-2 cycles. We can see that caching has a huge impact on system performance; a low hit rate can slow down throughput by hundreds to thousands of cycles depending on the length of the program.

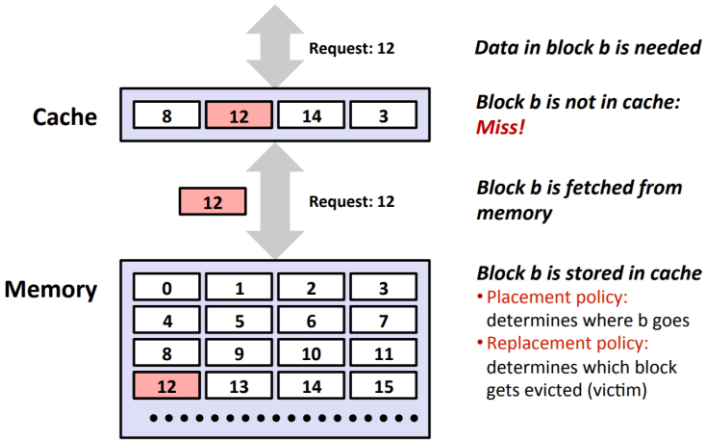



Figure 3. Diagram showing a cache miss [2].

The last caching concept is the *principle of locality*, which states that programs tend to use data and instruction with addresses near or equal to recently used ones. Locality is split into *temporal* and *spatial*. Temporal locality or "time" states that recently referenced blocks are likely to be called again in the near future. For example, suppose we have a for loop that sums the contents of an array. In terms of data, the sum variable is referenced in each iteration of the loop in short periods of time. In terms of instructions, the same instructions are called repeatedly when cycling through the loop. Spatial locality or "space" states that items within a neighborhood of addresses tend to be referenced close together in time. Within the same array example, there are also examples of spatial locality. For instance, the data within the array is accessed in succession, and the instructions within the loop are always referenced within the same sequence.

## 3 Locality Testing with MARS Simulator

The first portion of this project involved exploring extreme forms of locality by testing matrix algorithms. The first program that I ran was row.asm, which traverses a 16 by 16 element matrix in row major order, assigning each element a value from 0 to 255 in order [3]. The algorithm is as follows:

```
for (row = 0; row < 16; row++)
    for (col = 0; col < 16; col++)
         data[row][col] = value++;
```

The second algorithm that was testing was column.asm, which also traverses the matrix, but in column major order [3]. The column algorithm is as follows:

```
for (col = 0; col < 16; col++)
    for (row = 0; row < 16; row++)
        data[row][col] = value++;
```

From a normal algorithm analysis perspective there is no noticeable difference between these two functions. They both utilize double for-loops that run in $O(n^2)$ time. However, after running the cache simulator I observed very different results, which are discussed below.

To begin, I assembled each program within MARS and opened the Data Cache Simulator. I also set the run speed to 30 seconds per instruction so I could watch the cache performance animation. All default settings were kept for the first run.

The row major traversal ended with a cache hit rate of 75% and the memory visualization showed the matrix elements being accessed row-by-row as shown in Figure 4. The elements were accessed in the same order they were stored in. With each cache miss, a block of 4 elements was written into the cache, therefore the next 3 accesses were hits within the same block. When direct mapping mapped to the next cache block, another miss occurred, and the cycle repeated. Therefore, three out of four accesses were always hits. This matrix traversal utilizes spatial locality well because each element is accessed in order and the addresses are within the same neighborhood.

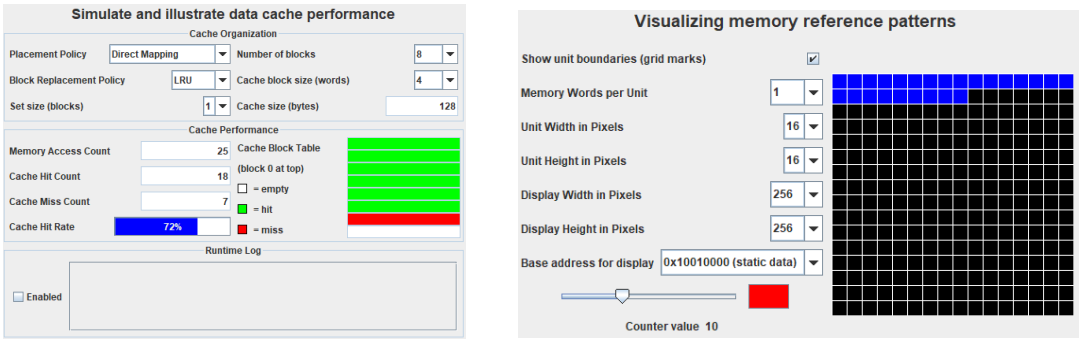



**Figure 4.** Row traversal cache performance and memory reference patterns.

In the column major traversal, with default settings the cache hit rate was 0% and the memory visualization showed access by column in Figure 5. The hit rate of 0% occurred because

elements were accessed 16 words beyond the previous access; since no two accesses occur in the same block, every access was a miss. This ended up being an extreme case of how locality plays a role in caching. In this code, since each cache access was a miss, every access also incurred the hit penalty. Column traversal ends up being hundreds of times of cycles slower than row traversal. This was a great example to show that while the asymptotic complexity of both algorithms are the same, the performance may be very different.

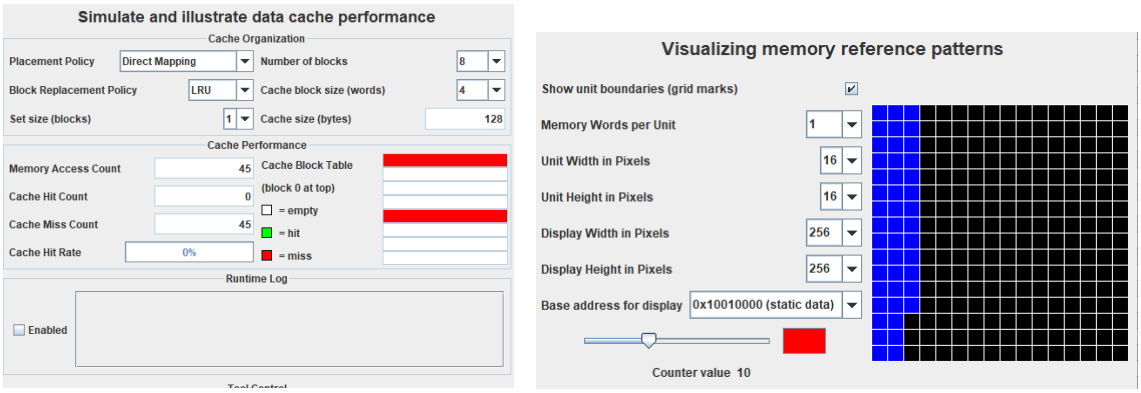



Figure 5. Column traversal cache performance and memory reference patterns.

Lastly, I was interested in conducting quantitative tests of varying the cache block size and observing the resulting hit rate. The results are shown in the graph in Figure 6. In this experiment the number of blocks was restricted to 8, while the cache block size was changed. The row traversal performance increased smoothly as the cache block size was increased. This was expected because as the block size increased, more than 4 words would be held in the same block and the sequential accesses would result in more hits. On the other hand, the column traversal did not improve until block size reached 32, where the entire matrix fit into the cache and no replacements were needed (32x8 = 256).

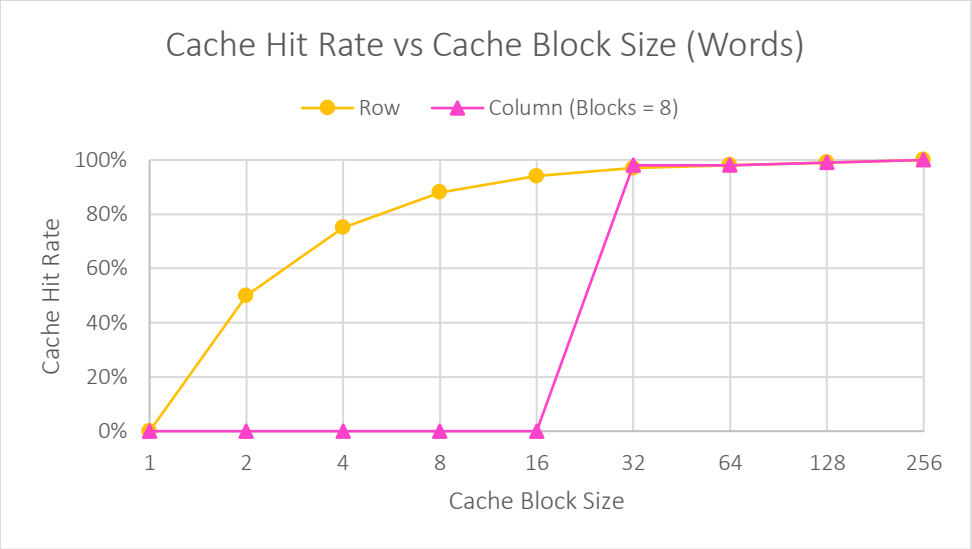



**Figure 6.** Cache hit rate when varying cache block size for row and column traversal.

I was also interested in varying the number of blocks and seeing the affect of the cache hit rate. Since row traversal implements spatial locality, changing the number of blocks did not change the performance. Figure 7 shows the data collected after varying the number of blocks during the column traversal. With each number of blocks, the trendline is the same as the original but offset by one cache block size each. With an increased number of blocks, a decreased cache block size is necessary in order to reach a hit rate higher than 0% where the entire matrix is stored into the cache. I also found it interesting that the hit rate follows the trend of the row traversal line.

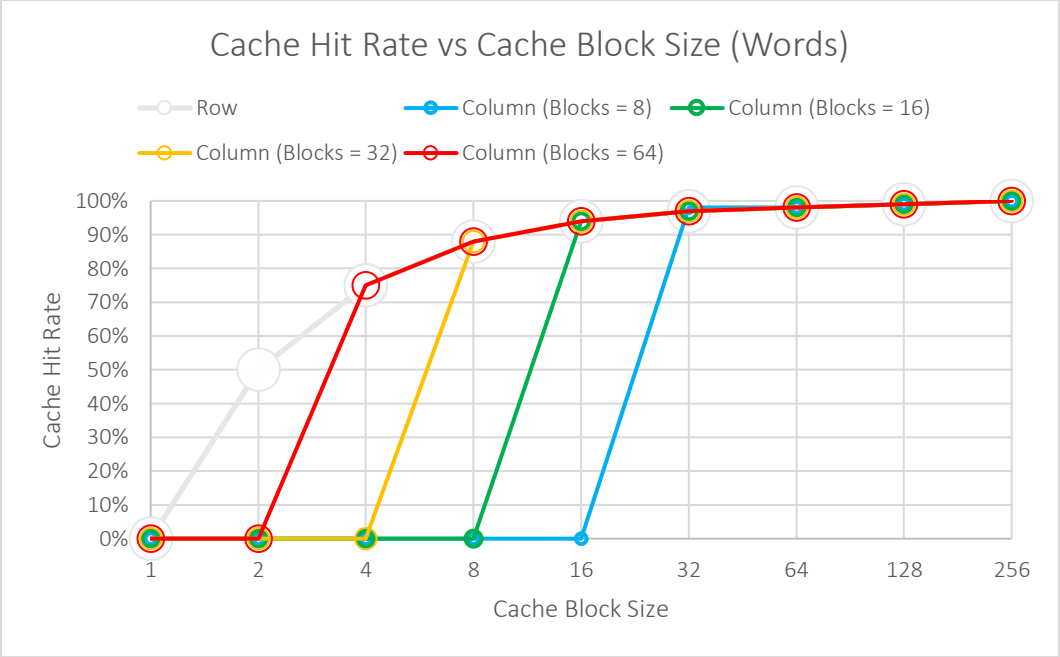



**Figure 7.** Cache hit rate when varying block size and number of blocks.

## 4 Iterative and Recursive Fibonacci Algorithm in MARS

The last quantitative test I wanted to conduct was comparing iterative and recursive programs and observing their cache performance. In an algorithm analysis class, we learned that the Fibonacci algorithms iterative and recursive versions perform drastically different in terms of O-notation. The dynamic programming version, or space optimized iterative version, only stores the two previous numbers calculated. It has a time complexity of O(n) and a space complexity of O(1) which is constant. The recursive version calculates a single Fibonacci number multiple times according the recursion tree. It has a time complexity of $O(2^n)$ and a space complexity of O(n).

I ran both programs for 1 to 20 Fibonacci numbers and observed the instruction counts and cache results. The instruction counts were as expected and closely followed the time complexity perspective, the results are shown in Figure 8. The iterative version remained linear,

increasing by 7 instructions per $n^{th}$ number, while the recursive function increased exponentially. By the $20^{th}$ Fibonacci number the iterative version had 157 instructions, while the recursive version was well over 350,000. The difference was so significant there was noticeable lag when the recursive version was running—setting MARS to max speed did not immediately finish the program.

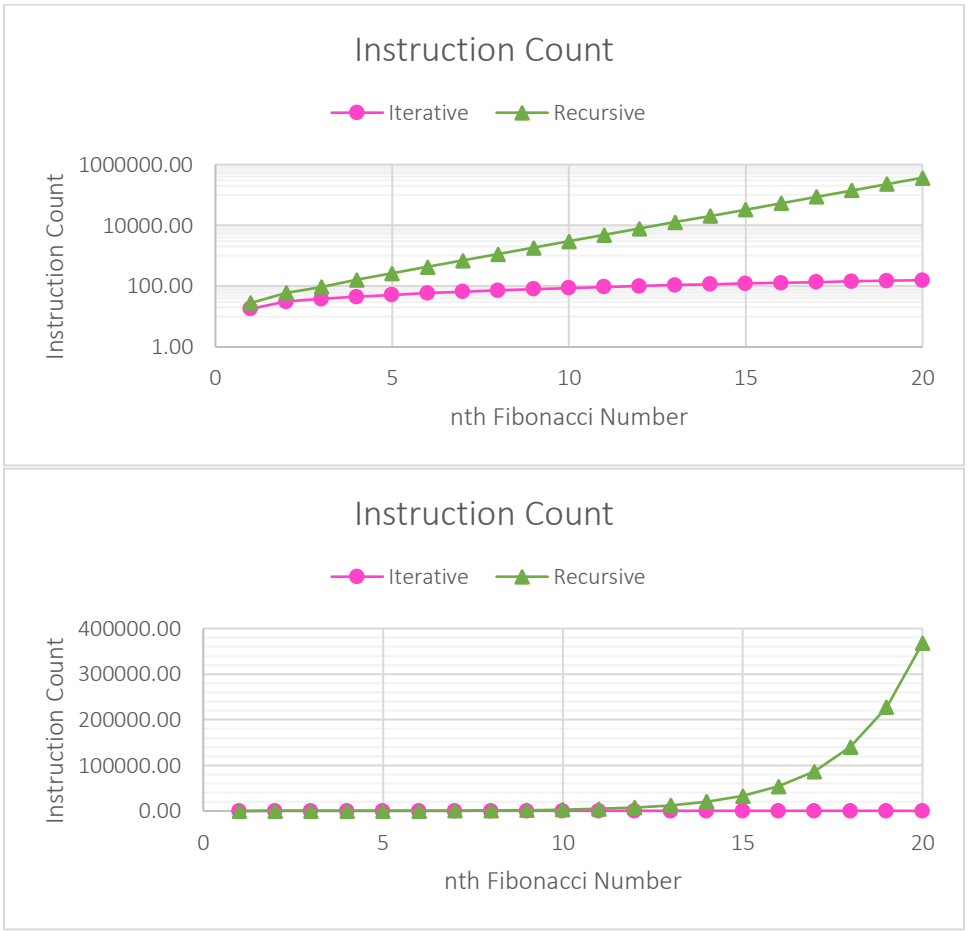
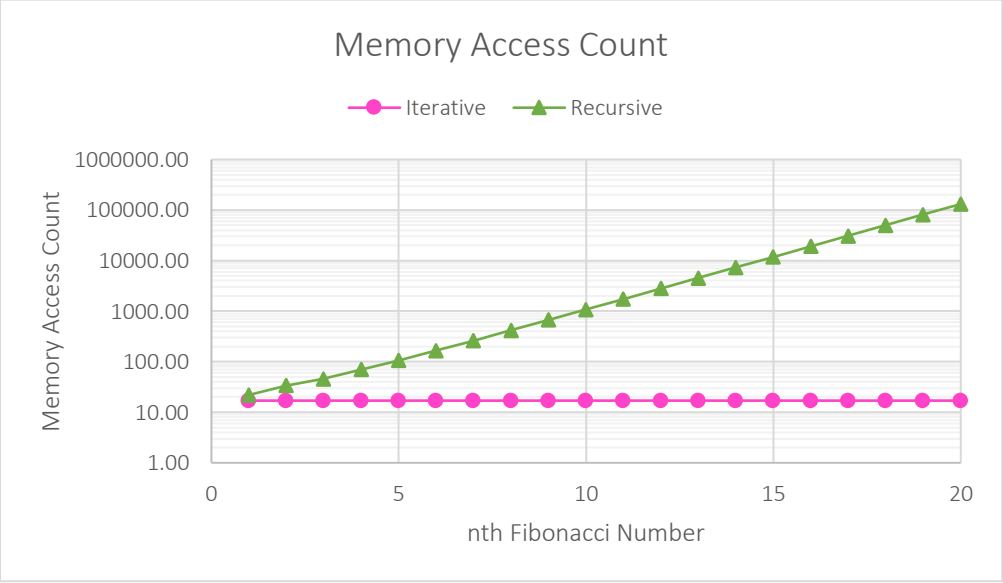



**Figure 8.** Instruction count of Fibonacci programs shown on logarithmic scale (top) and linear scale (bottom)

The memory access count and cache hit rate also followed the space complexity perspectives. The iterative version had 17 memory accesses and this remained constant for each number which agrees with the O(1) complexity. This also meant that the cache hit rate remained at 88% for every test. While 88% is not considered a great hit rate, the number of

accesses for the entire program was minimal and constant, so the miss penalty does not affect the overall runtime performance greatly. The accesses are so minimal because the iterative Fibonacci algorithm uses locality well. Variables a, b, and c which are used to store and calculate the next number are kept in immediate registers in each iteration of the loop, therefore the cache is not accessed.

The recursive version reached a cache hit rate of 91% to 100% as the calculated Fibonacci number increased, but the memory access count was also exponential as shown in Figure 9. The principle of locality is not followed well in this version. Temporally, the working set becomes so huge that similar addresses are not accessed within short time of each other. In addition, spatially, large strides are made between addresses when recursing back up through the tree. Therefore, while the cache hit rate does reach 100% by the 20$^{th}$ Fibonacci number, the memory access count still exponentially increases the runtime compared to the iterative version, because each access (even if it is a hit) still requires 1-2 cycles.



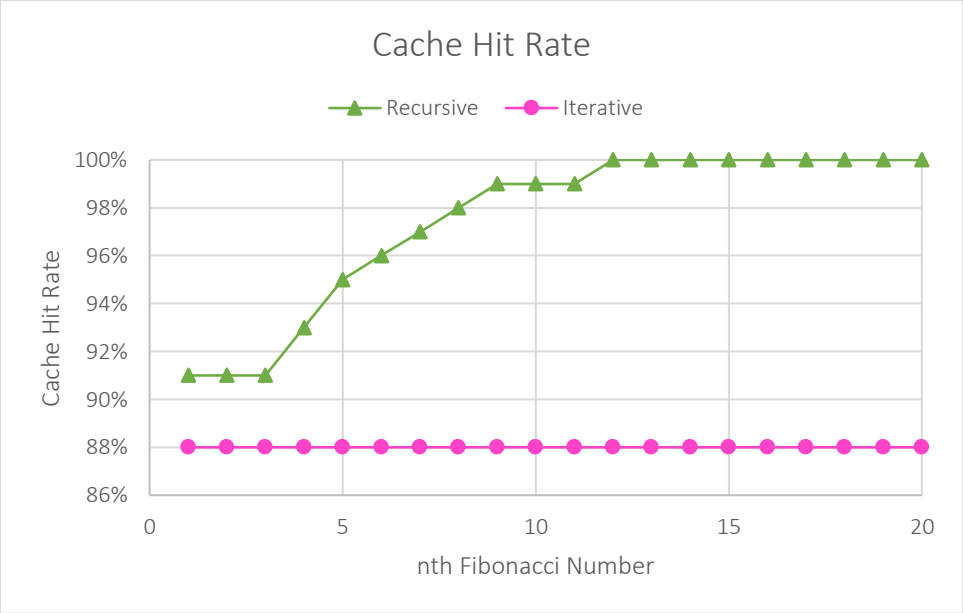



**Figure 9.** Memory access count and overall cache hit rate.

## 5 Conclusion

In conclusion, this project provided me with a deeper understanding of caching and how it affects the runtime performance of programs. I observed how the MIPS caching simulator works by testing extreme cases of locality using matrix traversal. I also tested recursive and iterative functions and observed the cache performance of each. My takeaway from this project is that there are multiple perspectives to look at for program complexity and optimization. Computational complexity may state one thing, while cache miss penalties say another. It is important to write cache friendly code, and there are multiple layers of optimization that can be done on a program.

To extend this project I would test other forms of the Fibonacci algorithm and look at the cache and runtime performance of those. For instance, the memorized recursive version would be interesting to test to see if the cache hit rate improves by using a single array to keep track of the already calculated numbers. There are other solutions that involve matrix multiplication or

using the Fibonacci formula directly. It would be interesting to see how these could be implemented in MIPS and the final cache performance.

## References

[1] G. Kesden and A. Rowe, "The Memory Hierarchy", Carnegie Mellon, 2011.

[2] G. Kesden and H. Pitelka, "Cache Memories", Carnegie Mellon, 2011.

[3] I. Raharja, "Cache Performance", King Fahd University of Petroleum and Minerals, 2018.