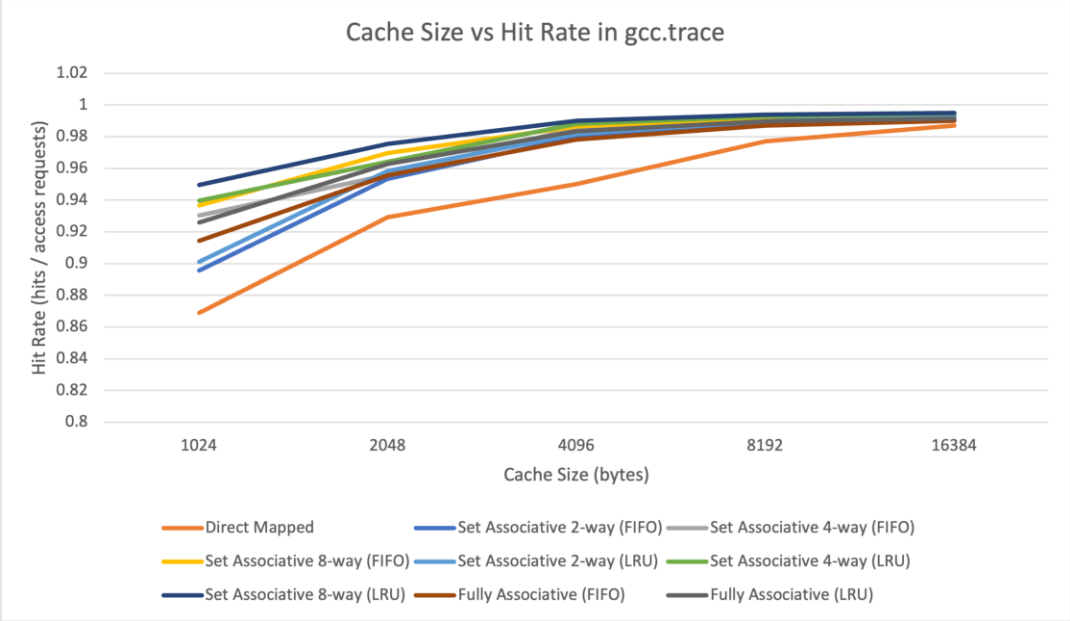
My simulation is a program written in C++ that utilizes user input to determine what implementation to use and under what parameters to implement a cache based on a provided text file. The program prompts the user to specify what cache implementation, the number of lines per set or the replacement policy to use if applicable, the block size, and the cache size. It then determines which function to run based on this information. There are 5 available and each is a different combination of a cache implementation and a replacement strategy. For example, there is a set associative first-in first-out function, but there is also a set associative least recently used function. Although each function works using a different data structure to emulate a cache, they each have the same basic steps. The functions will start by calculating the necessary variables such as the size of the tag and set fields. It will then proceed to run the rest of the code according to a while loop which will read each line of a provided text file and execute the rest of the function within it. For each line of text, the address is separated from any unneeded characters, translated into binary, split into the appropriate fields, and stored. Each function then defers on how to logically determine when an input registers a hit or miss but will record the number of hits and the total number of access requests. At the end the function then calculates the hit rate based off of these values and prints them. One notable feature of this program is that each implementation of a cache and replacement type utilizes different data structures or combinations of data structures. For example, direct mapped caches are simulated using a map of (string, int) but a set associative map using LRU uses a map of strings and vectors of pairs.

I conducted multiple tests using the provided file “gcc.trace” and different parameters. Each cache implementation and replacement strategy corresponded to a specific function in my program. This meant that I tested multiple different parameters in direct mapped caches, set associative using FIFO, set associative using LRU, fully associative utilizing FIFO, and fully associative caches using LRU. For set associative caches, I specifically tested 2, 4, and 8-way set associative for each replacement strategy. I chose to do 2, 4 and 8 because they are a good range of lines per set, where a noticeable difference between the performance of them can be discerned under the same parameters otherwise. The reason I tested every combination of

replacement strategy and cache implementation to be able to determine the differences in performance between implementations utilizing different approaches replacements. For each of these caches I tested using 1024, 2048, 4096, 8192, and 16384 cache size. This was important to be able to observe the trends as cache size increased, and provides a healthy range to ensure that results can be observed at both low and relatively high cache sizes. Block size was kept the same for each of the tests, this was in order to be able to more easily isolate variables in testing and to be able to better identify what changes affect any trends.

For the different configurations, there are some obvious trends and patterns present in my data from testing the above constraints.



This is a plot of cache sizes tested and the respective hit rates of different configurations of cache implementation, replacement strategy, and associativity. The most obvious trend amongst every cache implementation, regardless of replacement strategy or associativity is that as cache size increases, the hit rate increases until cache size gets relatively large. At larger cache sizes every cache shows a plateau behavior. Cache size also highlights an important point, which is that the hit rates of all the configurations eventually converge. The graph shows that direct mapped has the lowest hit rates consistently at all cache sizes. With set associative cache

implementations, as associativity increases, the hit rates at all cache sizes also increases. Another notable pattern is that for both full and set associative caches, those implementing LRU have higher hit rates than those using FIFO. The data also supports that as associativity increases, the difference in hit rate from one associativity (for example, 2 and 4 or 4 and 8) decreases. So, the difference between the hit rates of 2-way set associative caches and 4-way set associative caches is noticeably larger than the difference between 4 and 8-way set associative caches for both LRU and FIFO implementations. One point that my data does not fare well on is the specific hit rates of the full associative tests. Their patterns seem good, as in they follow the patterns shown in other implementation, however based on knowledge of the cache, their hit rates should start somewhere around where the 8-way set associative hit rates start (at 1024 bytes for cache size). This is because full associative caches in general should perform better (have a higher hit rate) than any other type of cache. This is not seen in my data and is most likely due to discrepancies in my program implementation that is either skewing set associative caches hit rates high or full associative caches hit rates low.

This data soundly supports multiple conclusions based on patterns seen across tests. In regard to cache size and its effects, the data shows that as cache size increases, hit rates across every combination of implementation, associativity, and replacement method also increase. However, they do all also show signs of plateau as cache size gets relatively high (16384 bytes). It is also evident that at higher cache sizes the hit rates of every combination seem to converge. These all support the conclusion that as cache size increases, hit rates increase, but also as cache size increases, the differences between each implementation in terms of hit rates diminishes to almost negligible fractions of a percentage. Meaning that at higher cache sizes it theoretically should be less important as to what implementation is used if the only criterion is hit rate. The data also shows clear patterns in terms of replacement policies. For both full and set associative, LRU performs better (has a higher hit rate) than FIFO. Implying that LRU implementations will achieve higher hit rates than FIFO implementations until larger cache sizes as discussed previously. Regarding cache design, a few patterns reveal themselves. One being that set associative caches will consistently have higher hit rates than direct associative caches. Another being that associativity increases for set associative caches, its hit rate also increases

(regardless of replacement policy). However, associativity also shows a pattern of diminishing returns, meaning that as associativity increases, the amount of improvement in hit rate decreases. Lastly, theoretically, fully associative should have higher hit rates than the other cache implementations (this is not seen in. My data due to error). These points all support the conclusion that the order from worst (in terms of hit rate) to best of the cache implementations is direct mapped, set associative with increasing associativity, and fully associative. And as associativity increases, there are diminishing returns of hit rate improvement. Overall, the simulation clearly showed a lot of patterns that support previous knowledge regarding the cache, while encountering some hiccups, still maintains a level of consistency regarding patterns over time, between replacement policies, and cache implementations.