Introduction

- Description of Simulation:

  The whole process of this cache simulation is to read in addresses from trace files, convert the hex data into binary, then depending on the cache design, direct mapped, fully associative, or set associative, and depending on the different parameters, the simulator will separate address into tag and offsets and insert tag into cache. As the simulator keeps reading more addresses from the trace file, it will check if the current tag is already in the cache, if it is in the cache, it will register a hit which will be used for hit rate calculation later on. On the other hand, if the tag is not in the cache, and if the cache is already full, before we add the tag into cache, the simulator will execute either FIFO, or LRU replacement policy to free up space.

Description of Tests

- What were the parameters for each test?

  I used parameters like set counts, blocks per set, bytes per block, vector<string> that stores the trace data, boolean variables for the Fully Associative and Set Associative test cases.

- Why did you choose these parameters?

  First of all, the set counts, blocks per set, and bytes per block are the essential part of the cache memory and this whole project, these three parameters determine the cache size of the cache simulator and allow us to do different test cases by changing them with measurements. All my tests take the vector string which stores the trace data in binary form which I read from the trace files before tests. Without this, no test will be able to execute due to missing information. And because Direct mapped cache design does not require replacement policy, I only included boolean variables as parameters for my tests for Fully Associative and Set Associative, and depending on the value of the boolean, tests will be executed via FIFO or LRU replacement policy.

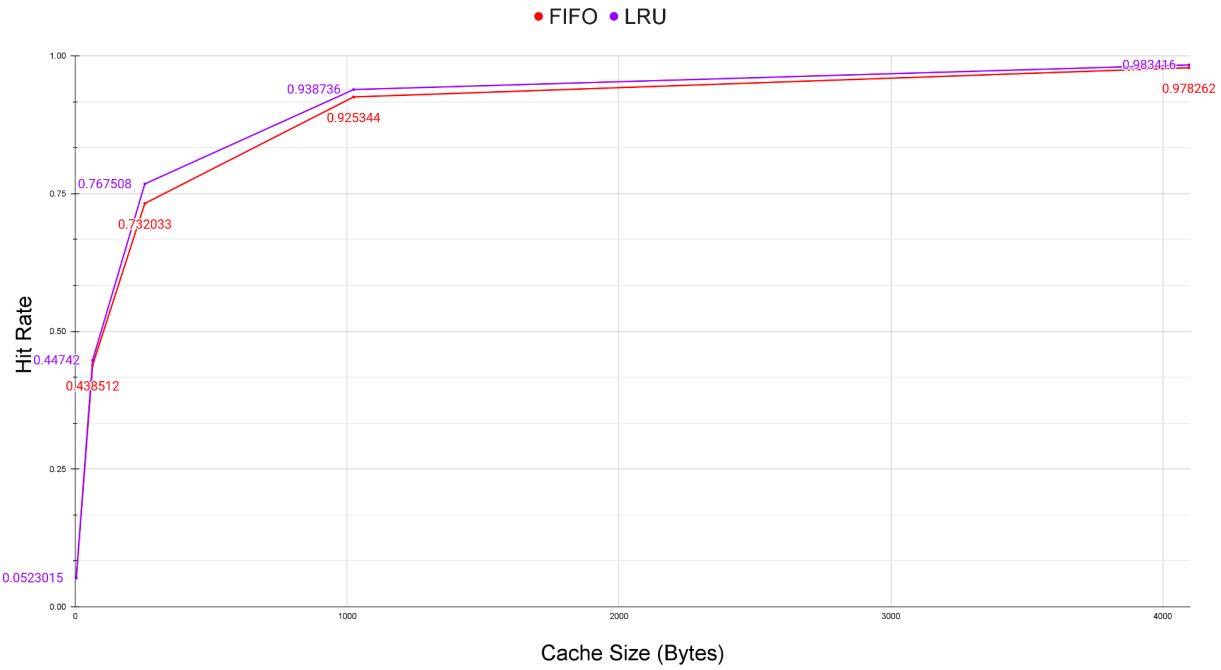
Results

- What were the hit rates for the different configurations?

  Table:

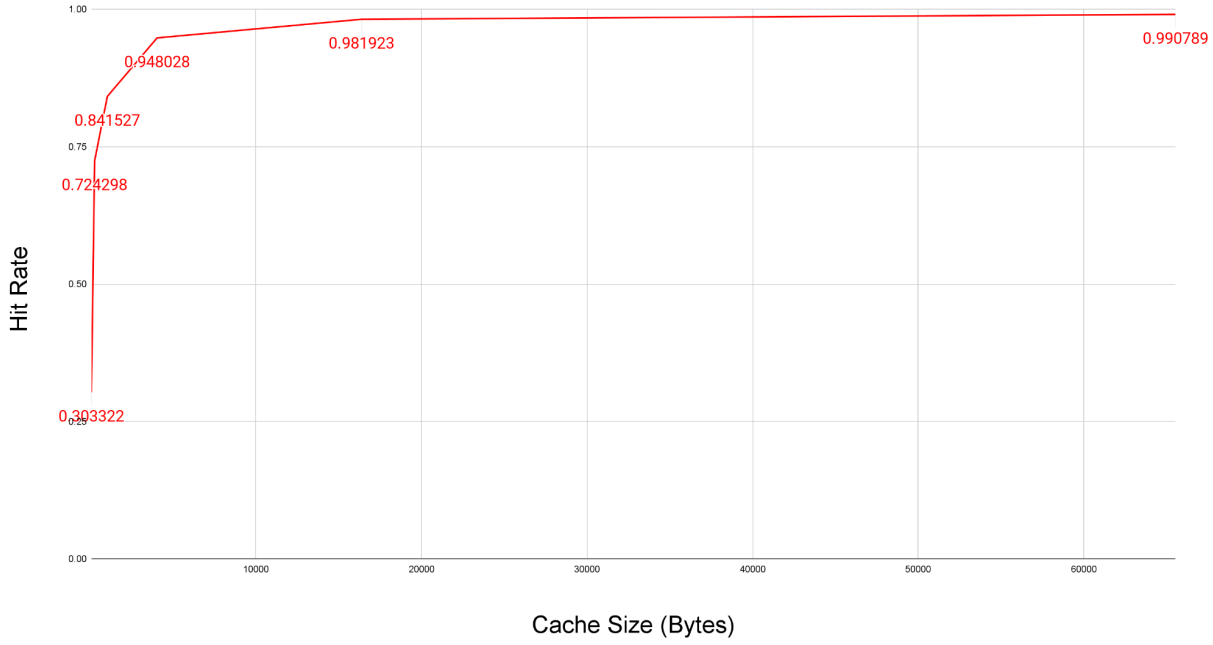
Fully Associative				
Sets	Blocks/Set	Bytes/Block	Replacement	Hit Rate
1	1	4	FIFO	0.0523015
1	1	4	LRU	0.0523015
1	16	4	FIFO	0.438512
1	16	4	LRU	0.44742
1	16	16	FIFO	0.732033
1	16	16	LRU	0.767508
1	64	16	FIFO	0.925344
1	64	16	LRU	0.938736
1	64	64	FIFO	0.978262
1	64	64	LRU	0.983416

Fully Associative



Direct Mapped				
Sets	Blocks/Set	Bytes/Block	Replacement	Hit Rate
16	1	4	N/A	0.303322
16	1	16	N/A	0.724298
64	1	16	N/A	0.841527
64	1	64	N/A	0.948028
256	1	64	N/A	0.981923
256	1	256	N/A	0.990789

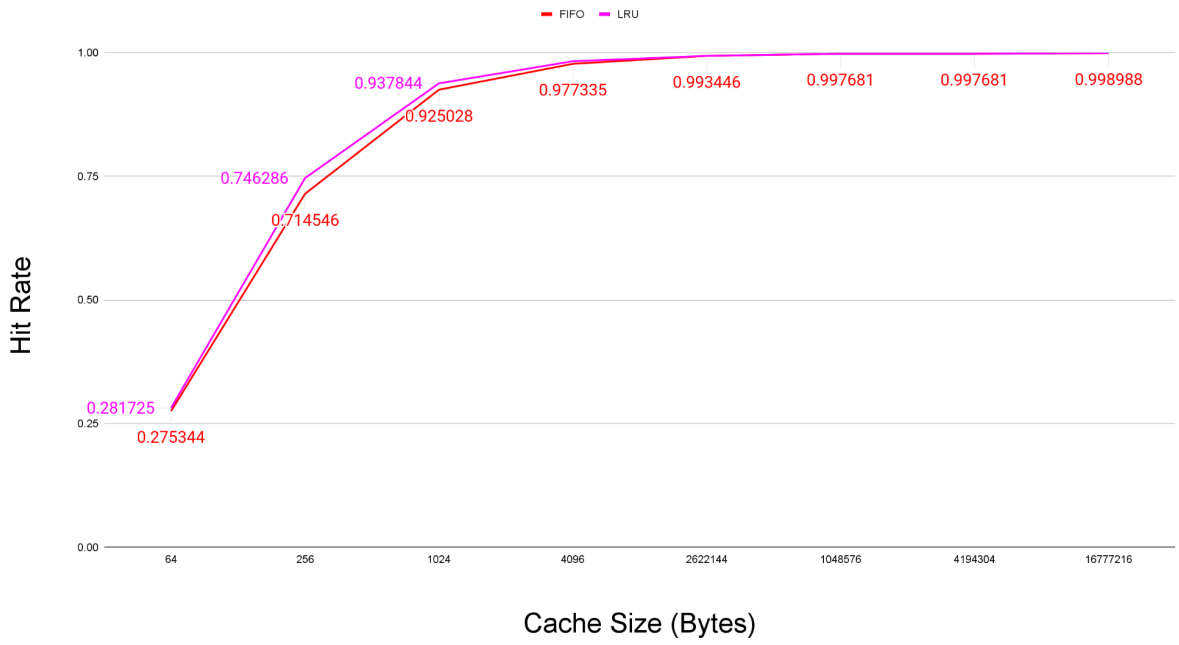
Direct Mapped



Set Associative				
Sets	Blocks/Set	Bytes/Block	Replacement	Hit Rate
4	4	4	FIFO	0.275344
4	4	4	LRU	0.281725
4	4	16	FIFO	0.714546
4	4	16	LRU	0.746286
4	16	16	FIFO	0.925028
4	16	16	LRU	0.937844
4	16	64	FIFO	0.977335
4	16	64	LRU	0.982604
64	64	64	FIFO	0.993446
64	64	64	LRU	0.993461
64	64	256	FIFO	0.997681
64	64	256	LRU	0.997681

64	256	256	FIFO	0.997681
64	256	256	LRU	0.997681
64	256	1024	FIFO	0.998988
64	256	1024	LRU	0.998988

Set Associative



Conclusions

- What can you say about cache design
  - Direct Mapped
    Compared to the other two cache designs, under the same cache size, direct mapped have the least amount of hit rate due to it having no replacement policy. With that being said, Direct mapped cache design is arguably the fastest and cheapest cache design to execute with that same reason.
  - Fully Associative
    Fully associative is the second cache design I worked on after direct mapped, with it being required using replacement policies, it is definitely more expensive to execute than direct mapped, but also more accurate.

According to the data points I have gathered, Fully associative cache design has the highest hit rate among the three with the same parameters.

- Set Associative
  As I was working on the implementation of Set associative cache design, I realized this cache design is the combination of the previous two, will have a setID similar to the lineID directly mapped requires, set associative also requires the replacement policies which fully associative requires. In summary, set associative is the number of fully associative sets combined together with setIDs. But surprisingly, this cache design is still not out performing the fully associative on the hit rate accuracy, but it does have way more cache size capability than the other two.

- What can you say about replacement policies
  - FIFO
    First in first out replacement policy is very straightforward, as the cache size being full, the system will remove the first element entering the cache to free up space for feature tags. This policy is cheaper to execute than the LRU, but due to our advanced hardware, the difference is not that noticeable. While the cache size gets bigger and bigger, the hit rate accuracy of FIFO policy will be closer and closer to LRU and eventually be identical.
  - LRU
    Least recently used replacement policy removes the element in the cache that was least recently used in order to free up space for feature tags. It is not very different from the FIFO, with a couple lines of additional code, this policy gives an edge over FIFO with smaller cache sizes, but as I mentioned earlier, as the cache size gets bigger and bigger, these two replacement policies return identical performances.

- What can you say about cache size

  As the cache size gets bigger and bigger for all three cache designs, the hit rate for all of them increases, and gets better. Although with small cache size, hit rate for some of the designs will be as low as 0.05, the hit rate vs cache size graph for all three designs appears to be exponential growth.