Cache Performance Analysis Through Simulation

By Steven Miller
Department of Electrical and Computer Engineering
University of Florida
4/13/2023

Table of Contents:

Introduction

This paper seeks to examine the efficiency of all three cache designs: direct mapped, fully associative, and set associative by simulating them in a program, and measuring their “hit rates”, which is how often we find data that we want in the cache.

A **direct mapped cache** works by directly assigning each memory address to a block of the cache. A downside to the direct mapped cache is that because every memory addresses assignment in the cache is essentially predetermined, there is often a low hit rate associated with this type of cache design.

A **fully associative cache** works by placing a memory address in any block in the cache that is empty. This downside to this type of cache design is that you must iterate through the cache until you find an empty block, which can be computationally expensive.

A **set associative cache** works by splitting the cache blocks into “sets”. Each memory address is given a set number. Once its set number is found, the set is iterated until an empty block is found. A downside to this is that you must iterate through each set. Which can be computationally expensive.

Description of Simulation

The memory addresses for this simulation are 32 bits in length.

When the simulation is started, the user is asked which cache type they would like to simulate, how big the cache should be in terms of bytes, and they are asked how big a block should be in bytes. Both entries must be powers of 2. They are asked if LRU or FIFO should be used as a replacement strategy. Finally, if they select a set associative design, they are asked how many blocks should be in a set.

The different cache designs are implemented as follows:

Direct mapped:

When the cache needs to be populated, the memory address has its offset width (in bits), and its block number width (in bits) subtracted to produce its tag. The tag is then placed into the cache index corresponding to the block number.

Fully associative:

When the cache needs to be populated, the memory address has its offset width(in bits), and its block number width (in bits) subtracted to produce its tag. The cache is then iterated to find an empty block. Once an empty block is found, the tag number of the memory address is placed in that block.

Set associative:

When the cache needs to be populated, the memory address has its offset width(in bits), its set number width(in bits) subtracted to produce its tag. The set is then found and iterated to find an empty spot to place the tag.

Description of Simulation

When either fully associative or set associative designs are full. A replacement strategy must be implemented. To implement this replacement strategy, every block in the cache will have a “counter” variable that will be compared with a global “counter” variable. Depending on what the user selected, one of the following replacement strategies will take place:

LRU(Least Recently Used):

The global counter variable will be incremented every time a hit occurs, or when new data is added or swapped in the cache. The block where the hit or data placement occurred will then have its own counter variable value overwritten by the global counter variable.

In the event that that a cache is full, the entire cache is iterated to find the block with the lowest counter value, once it finds the block, the blocks tag is replaced, and the counter variable is overwritten by the global counter variable.

FIFO(First In First Out):

The global counter variable will be incremented when new data is added or swapped in the cache. The block where the data placement occurred will then have its own counter variable value overwritten by the global counter variable.

In the event that that a cache is full, the entire cache is iterated to find the block with the lowest counter value, once it finds the block, the blocks tag is replaced, and the counter variable is overwritten by the global counter variable.

Description of Tests

Two sets of tests were ran:

1.Hit rate of all cache types using LRU.

2.Hit rate of all cache types using FIFO.

The two sets were then split into subsets of tests:

1. **Hit rate of all cache types using LRU.**

    1a.Hit rate of direct mapped using LRU.

    1b.hit rate of fully associative.

    1c.Hit rate of set associative.

        1ca.Hit rate of 2 way set associative.

        1cb.Hit rate of 4 way set associative.

        1cc.Hit rate of 8 way set associative.

2. **Hit rate of all cache types using FIFO.**

    2a.Hit rate of direct mapped using LRU.

    2b.hit rate of fully associative.

    2c.Hit rate of set associative.

        2ca.Hit rate of 2 way set associative.

        2cb.Hit rate of 4 way set associative.

        2cc.Hit rate of 8 way set associative.

Description of Tests

1.Hit rate of all cache types using LRU.

- **1a.Hit rate of direct mapped**
  - **1aa.Hit rate of cache size 512**
  - **1ab.Hit rate of cache size 1024**
  - **1ac.Hit rate of cache size 2048**
  - **1ad.Hit rate of cache size 4096**
  - **1ae.Hit rate of cache size 8192**
  - **1af.Hit rate of cache size 16384**
- **1b. Hit rate of fully associative**
  - **1ba.Hit rate of cache size 512**
  - **1bb.Hit rate of cache size 1024**
  - **1bc.Hit rate of cache size 2048**
  - **1bd.Hit rate of cache size 4096**
  - **1be.Hit rate of cache size 8192**
  - **1bf.Hit rate of cache size 16384**

Description of Tests

1c.Hit rate of set associative.

- **1ca.Hit rate of 2 way associative**
  - **1caa.Hit rate of cache size 512**
  - **1cab.Hit rate of cache size 1024**
  - **1cac.Hit rate of cache size 2048**
  - **1cad.Hit rate of cache size 4096**
  - **1cae.Hit rate of cache size 8192**
  - **1caf.Hit rate of cache size 16384**
- **1cb. Hit rate of 4 way associative**
  - **1cba.Hit rate of cache size 512**
  - **1cbb.Hit rate of cache size 1024**
  - **1cbc.Hit rate of cache size 2048**
  - **1cbd.Hit rate of cache size 4096**
  - **1cbe.Hit rate of cache size 8192**
  - **1cbf.Hit rate of cache size 16384**
- **1cc.Hit rate of 8 way associative**
  - **1cca.Hit rate of cache size 512**
  - **1ccb.Hit rate of cache size 1024**
  - **1ccc.Hit rate of cache size 2048**
  - **1ccd.Hit rate of cache size 4096**
  - **1cce.Hit rate of cache size 8192**
  - **1ccf.Hit rate of cache size 16384**

Description of Tests

2.Hit rate of all cache types using LRU.

- **2a.Hit rate of direct mapped**
  - **2aa.Hit rate of cache size 512**
  - **2ab.Hit rate of cache size 1024**
  - **2ac.Hit rate of cache size 2048**
  - **2ad.Hit rate of cache size 4096**
  - **2ae.Hit rate of cache size 8192**
  - **2af.Hit rate of cache size 16384**
- **2b. Hit rate of fully associative**
  - **2ba.Hit rate of cache size 512**
  - **2bb.Hit rate of cache size 1024**
  - **2bc.Hit rate of cache size 2048**
  - **2bd.Hit rate of cache size 4096**
  - **2be.Hit rate of cache size 8192**
  - **2bf.Hit rate of cache size 16384**

Description of Tests

2c.Hit rate of set associative.

2ca.Hit rate of 2 way associative

2caa.Hit rate of cache size 512

2cab.Hit rate of cache size 1024

2cac.Hit rate of cache size 2048

2cad.Hit rate of cache size 4096

2cae.Hit rate of cache size 8192

2caf.Hit rate of cache size 16384

2cb. Hit rate of 4 way associative

2cba.Hit rate of cache size 512

2cbb.Hit rate of cache size 1024

2cbc.Hit rate of cache size 2048

2cbd.Hit rate of cache size 4096

2cbe.Hit rate of cache size 8192

2cbf.Hit rate of cache size 16384

2cc.Hit rate of 8 way associative

2cca.Hit rate of cache size 512

2ccb.Hit rate of cache size 1024

2ccc.Hit rate of cache size 2048

2ccd.Hit rate of cache size 4096

2cce.Hit rate of cache size 8192

2ccf.Hit rate of cache size 16384

For all these tests, the size of a block is always 64 bytes. The reason for this is that a block size of 32 bytes results in noticeably higher hit rates. 64 bytes were chosen to simulate a "worst case" scenario. If the block size were any higher, the hit rate would be unrealistically low.

Results of Tests

The following results were obtained from performing the 1st set of tests (LRU):

	Hit Rate				
Size	***Direct**	**Fully**	**Set(size 2)**	**Set(size 4)**	**Set(size 8)**
512	**0.785403**	**0.927607**	**0.819589**	**0.845463**	**0.850272**
1024	**0.831627**	**0.980232**	**0.891239**	**0.920655**	**0.925086**
2048	**0.889568**	**0.997584**	**0.921712**	**0.952496**	**0.962198**
4096	**0.948028**	**0.999209**	**0.967748**	**0.972844**	**0.980998**
8192	**0.963735**	**0.99954**	**0.98381**	**0.987651**	**0.988252**
16384	**0.981923**	**0.999666**	**0.989115**	**0.990599**	**0.990733**

***Note that a direct mapped cache will have the same hit rate regardless of replacement method. This is because a memory address will always map to the same block in a direct mapped cache, regardless of whether it's full or not. Even if it must replace data.**

10/15

Results of Tests

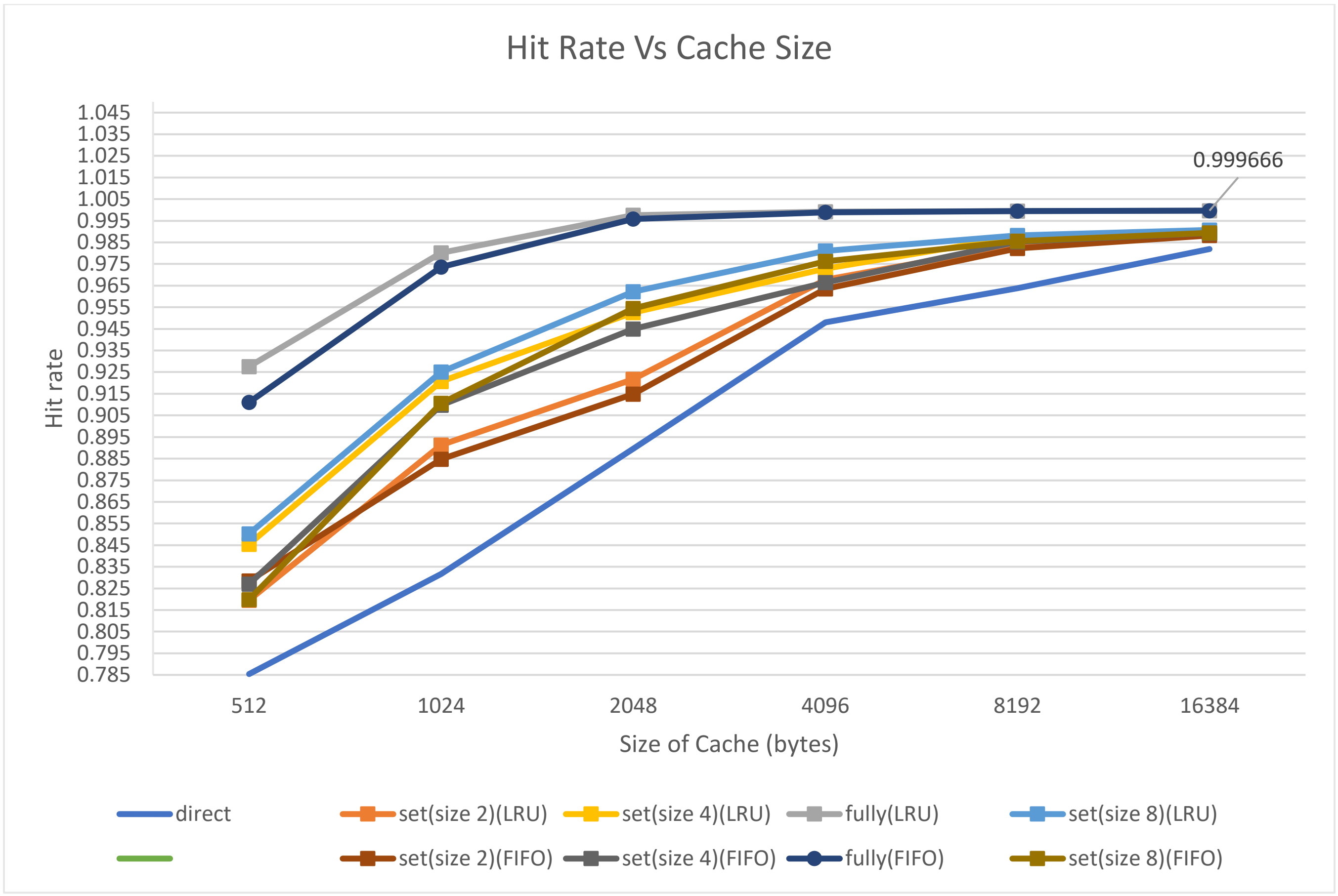
The following results were obtained from performing the 2nd set of tests(FIFO):

	Hit Rate				
Size	***Direct**	**Fully**	**Set(size 2)**	**Set(size 4)**	**Set(size 8)**
512	**0.785403**	**0.911052**	**0.828462**	**0.827045**	**0.81988**
1024	**0.831627**	**0.973668**	**0.884747**	**0.909718**	**0.910623**
2048	**0.889568**	**0.995839**	**0.914864**	**0.944891**	**0.954396**
4096	**0.948028**	**0.998933**	**0.96348**	**0.966413**	**0.976185**
8192	**0.963735**	**0.999471**	**0.982222**	**0.985305**	**0.985524**
16384	**0.981923**	**0.999666**	**0.988187**	**0.989296**	**0.989399**

***Note that a direct mapped cache will have the same hit rate regardless of replacement method. This is because a memory address will always map to the same block in a direct mapped cache, regardless of whether it's full or not. Even if it must replace data.**

11/15

Results of Tests



Conclusions

From the data we obtained in the previous pages, we can come to the following 3 conclusions:

1. **LRU replacement algorithm will always be more efficient than a FIFO replacement algorithm.**

   The reason for this is that LRU considers both the items that are accessed and the order that they are accessed in.
   For example: Address 0x1234 and 0x1235 are inserted into a 2-block cache of either set associative or fully associative. Where each address takes up 1 block.

Block number	Address	Counter
0	0x1234	1
1	0x1235	2

   Total hits:0
   Total misses:2

   If we were to perform a read command on 0x1234, the cache would become:
   For LRU:

Block number	Address	Counter
0	0x1234	3
1	0x1235	2

   Total hits:1
   Total misses:2
   For FIFO:

Block number	Address	Counter
0	0x1234	1
1	0x1235	2

   Total hits:1
   Total misses:2

Conclusions

We now insert address 0x1236:

For LRU:

Block number	Address	Counter
0	0x1234	3
1	0x1236	4

Total hits:1
Total misses:3

FIFO:

Block number	Address	Counter
0	0x1236	3
1	0x1235	2

Total hits:1
Total misses:3

We now perform a read on address 0x1234 again:

For LRU:

Block number	Address	Counter
0	0x1234	5
1	0x1236	4

Total hits:2
Total misses:3

FIFO:

Block number	Address	Counter
0	0x1236	3
1	0x1234	4

Total hits:1
Total misses:4

If we calculate the hit ratio:
LRU: .67
FIFO: .25

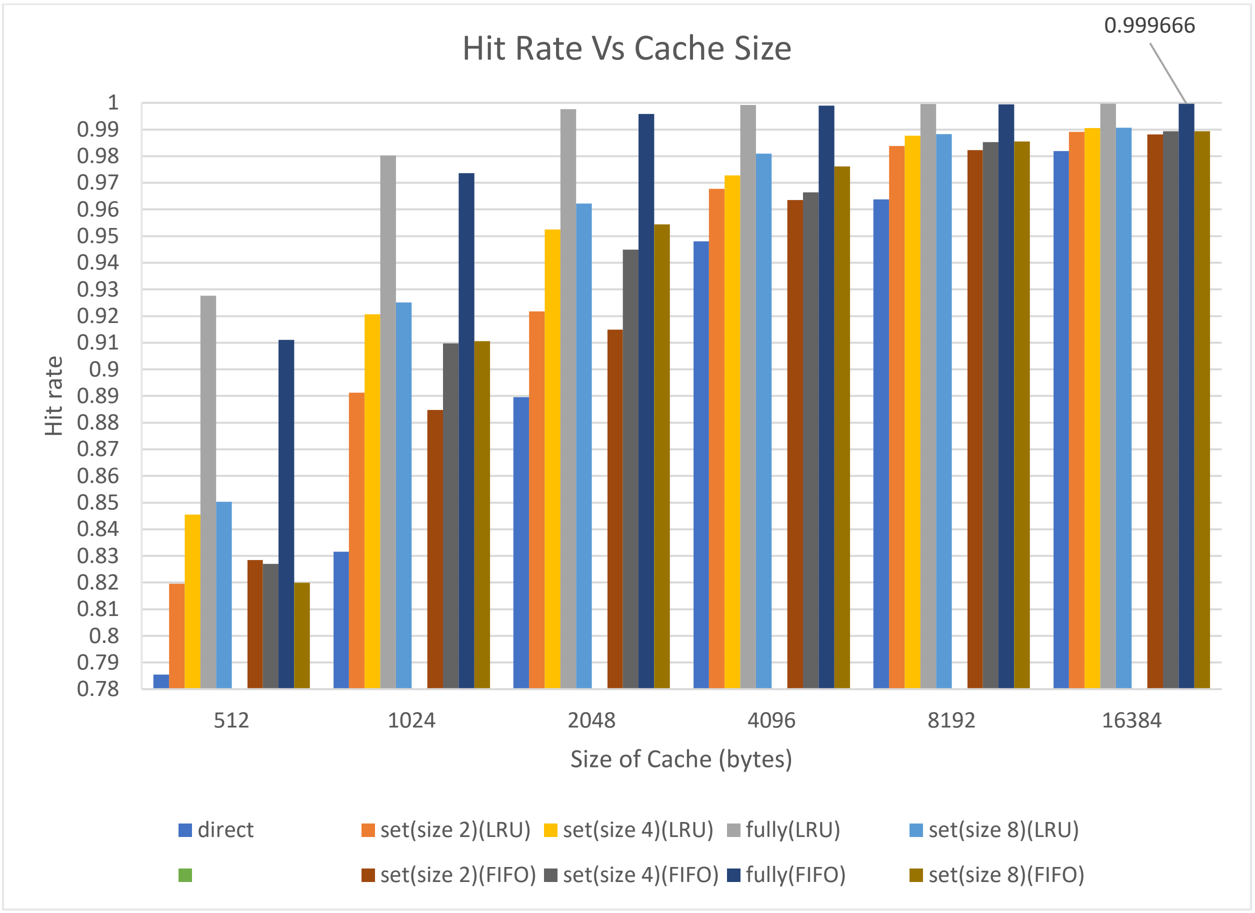
Conclusions

2. As cache size increases, the hit rate for all cache types increases. Independent of the replacement algorithm.

The reason for this is that as the cache size increases, more blocks become available to use by the addresses. Thus, lowering the amount of times we need to perform replacement.

3. Fully associative will always be the most efficient cache.

Lets refer to the following bar chart:



As we can see, fully associative always has the highest hit rate out of all the cache designs. This is because fully associative will only implement replacement when the cache is full.

Summary

In conclusion:

1.LRU replacement will always be more efficient than a FIFO replacement. Regardless of the cache type used. This is because LRU takes into the account items most recently accessed as well as the order that items came in.

2.As cache size increases, the hit rate increases, this is because there is more room on the cache for addresses to be placed.

3.Fully associative will always be the most efficient cache type. As it will only implement replacement when the cache is completely full.