Mini Project III

EE 275 Advanced Computer Architecture

Cache Simulator

Submitted By

Bhavin Patel

SJSU-ID – 015954770

MSEE – Fall 2022

San Jose State University

Department of Electrical Engineering

1. Introduction

A CPU hardware cache is a smaller memory, located closer to the processor, that stores recently referenced data or instructions so that they can be quickly retrieved if needed again. By reducing costly reads and writes that access the slower main memory, caching has an enormous impact on the performance of a CPU. Virtually all modern processors employ some form of caching. Cache partitions led to the birth of multi-level cache hierarchies, where processor cores would have their own small, private cache (L1) that sat above a larger shared cache (L2), with some processors including a third cache level (L3) and occasionally a fourth (L4).

This project simulates a one level processor cache(L1) for a series of memory reads. It simulates all three fundamental caching schemes: direct-mapped, *n*-way set associative, and fully associative. It also simulates different cache sizes, block sizes and replacement policies. The simulator will output several statistics (e.g. hit rate, total running time of the program, etc).

The project helps understand how the different cache design parameters such as cache-size, block-size, replacement policy, and cache associativity affect the cache performance. The final statistics helps pick the best configuration for a cache design. The cache design also requires considering the hardware resource requirements, however it does not affect the cache hit rate, and therefore is outside the scope of this project.

2. Cache Design Parameters

2.1 Cache Associativity

How and where a cache stores and retrieves data is based on the way that cache is organized. This is called the logical organization of the cache. Determining what gets stored is controlled by management heuristics built into the cache, but it is also heavily influenced by the logical organization. Thus, how a cache is laid out plays a huge role in its performance.
There are three main ways to organize a cache:

- Direct-mapped
- Fully associative
- Set associative

Direct-Mapped

In a direct-mapped cache, the cache entries are organized into a number of sets. The set number from the address is used to index each set of entries. Once the set is identified, the cache tags are compared. If they are a match, this is a cache hit and the specified data is output.

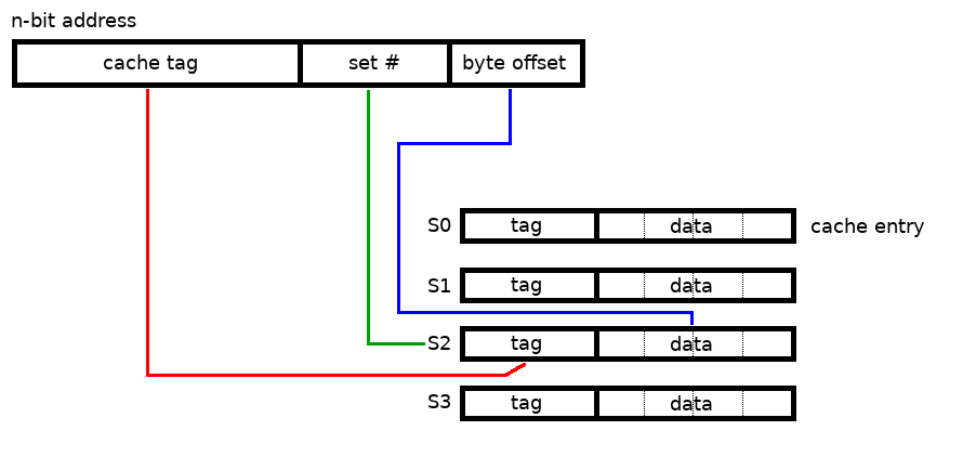

Fig 1. A direct-mapped cache

Fully Associative

A fully associative cache is the opposite of a direct-mapped cache. Instead of multiple sets containing a single entry, fully associative caches have multiple cache entries all contained in a single set. Thus, the set number no longer provides any information and is not used. Instead, when a memory address is handled by the cache, every cache entry is checked for a matching tag. If found, the byte offset is used to output the correct data within the cache block.

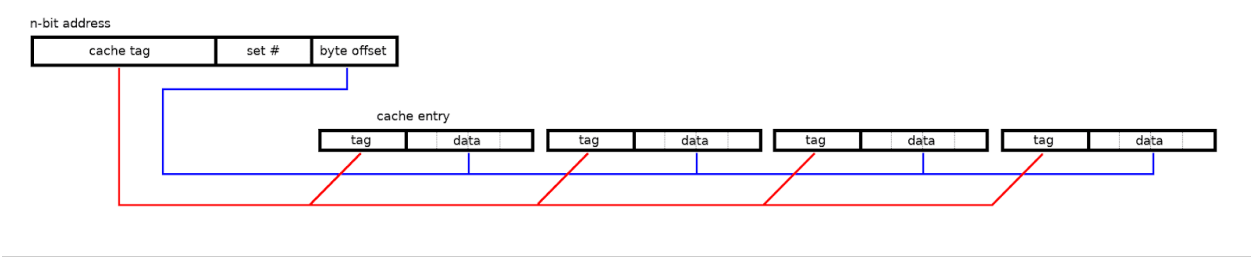

Fig 2. Fully Associative cache

Set Associative

A set associative cache offers the best of both worlds. It consists of multiple sets with multiple cache entries per set. How does it work? First, the set number allows the cache to jump to the appropriate set of entries. Next, each set of entries is searched for a matching tag. If found, the byte offset is used to output the requested data. This approach allows the cache to offer an optimized balance of power consumption and contention rate.

Fig 3. Set Associative Cache

2.2 Cache Blocks

When a CPU needs to access an item in main memory, it uses an address to locate that item. A CPU hardware cache typically works transparently, meaning without the programmer having to acknowledge the cache in any way. Thus, the address used to access memory is first handled by the cache. This address is used to identify whether or not a data item is located in the cache.

Caches are organized into groups of data called cache blocks. Each address is partitioned into a number of bit fields so that the correct cache block can be identified. These fields are the cache tag, set number, and byte offset. Figure 1 shows an address split into fields that the cache can interpret.

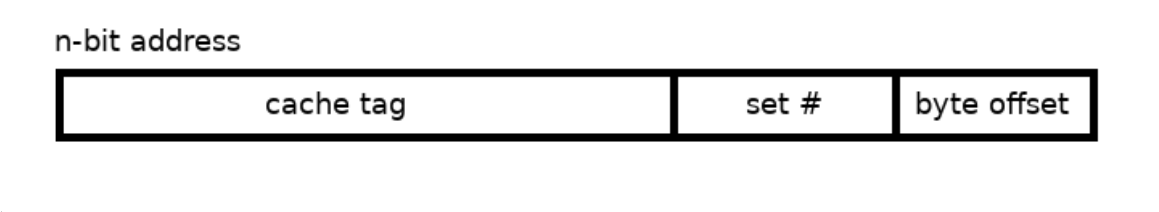

Fig 4. Addressing a Cache Block

When a CPU cache is given an address, it breaks this address into the necessary fields and begins checking its cache entries. A cache entry consists of a cache tag (here labeled tag) and a cache block (labeled data).

- The cache tag is an identifier that signals which cache block is being referred to.
- The cache block is the actual data stored at that tag and represents a block of items from main memory. To get to individual words within that block, an offset is used.

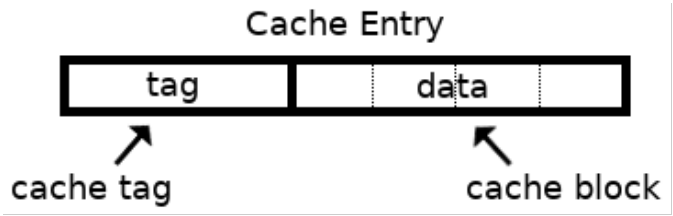

Fig 5. Cache Entry

2.3 Cache Size

It could be left up to the programmer or compiler to determine what data should be placed in the cache memories, but this would be complicated since different processors have different numbers of caches and different cache sizes. It would also be hard to determine how much of the cache memory to allocate to each program when several programs are running on the same processor. Instead the allocation of space in the cache is managed by the processor. When the processor accesses a part of memory that is not already in the cache it loads a chunk of the memory around the accessed address into the cache, hoping that it will soon be used again.

The chunks of memory handled by the cache are called cache lines. The size of these chunks is called the cache line size. Common cache line sizes are 32, 64 and 128 bytes.

A cache can only hold a limited number of lines, determined by the cache size. For example, a 64 kilobyte cache with 64-byte lines has 1024 cache lines.

2.4 Replacement Policies

If all the cache lines in the cache are in use when the processor accesses a new line, one of the lines currently in the cache must be evicted to make room for the new line. The policy for selecting which line to evict is called the *replacement policy*.

The most common replacement policy in modern processors is **LRU**, for *least recently used*. This replacement policy simply evicts the cache line that was least recently used, assuming that the more recently used cache lines are more likely to soon be used again.

The simplest cache replacement algorithm in modern processors is **FIFO**. In this algorithm, the system keeps track of all cache lines of the set, the oldest line is replaced when a cache line needs to be replaced for loading a new line.

Another replacement policy is **random** replacement, meaning that a random cache line is selected for eviction.

3. Implementation

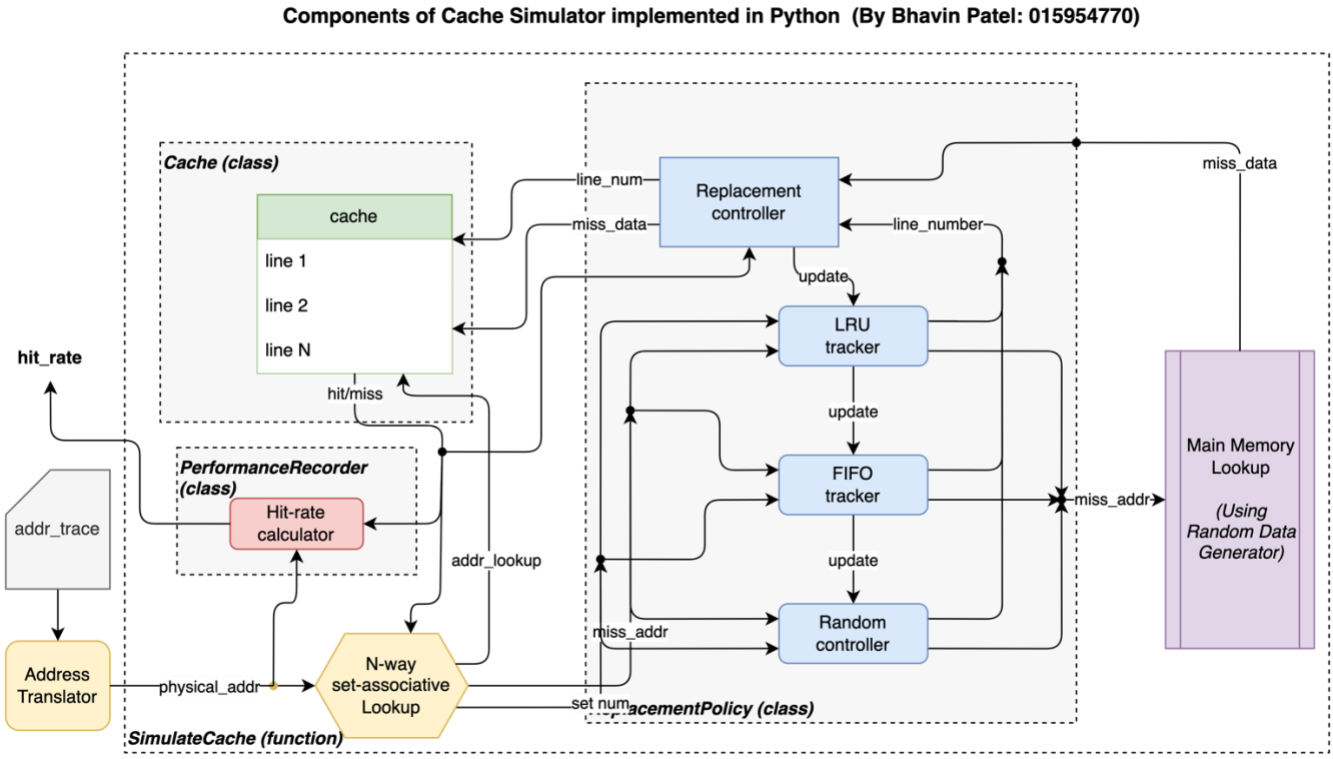

Fig 6. Cache Implementation in Python

The Cache implementation in Python contains below components.

1. **Address Translator:**

    - The set of addresses given in the addr_trace.txt are relative to the first address. The first address is the absolute address and the remaining lines are offset to this absolute address. The address translator converts the relative address in the given addr_trace.txt to their absolute addresses. After the conversion, the absolute addresses are forwarded to the ***SimulateCache*** function.

2. **Cache:**

    - The *Cache* class implements the cache using three levels of classes.

        1. Cache (top)
        2. CacheSet (set)

3. CacheLine (line)

Cache contains multiple instances of CacheSet depending on the value of block size and cache size passed as a parameter. Each CacheSet contains multiple instances of CacheLine, and similarly each CacheLine contains multiple blocks.

- The *Cache* class supports two primary functions.

i. Lookup (function)

Lookup function allows reading the cache line from the given physical address. The ***Cache*** identifies the set number from the *LookupObj* and passes it to the detected ***CacheSet***.

The ***CacheSet*** then matches the tag from *LookupObj* to the tags inside each of the ***CacheLine*** it contains. If a tag match is found, the *LookupObj* is forwarded to that ***CacheLine***. At the same time, it prompts the ***ReplacementPolicy*** and ***PerformanceRecorder*** to update accordingly.

The ***CacheLine*** then uses the block offset from *LookupObj* to find the given block and copies its data to the *LookupObj* field.

ii. Load (function)

Load function allows loading a line of data from main memory to a cache set. The ***Cache*** identifies the set number from the *LoadObj* and passes it to the detected ***CacheSet***.

The ***CacheSet*** then asks the ***ReplacementPolicy*** to return a line number which is to be replaced as per the LRU/FIFO/Random replacement policy. This line number helps locate the ***CacheLine*** that needs to be modified

The ***CacheLine*** then receives the new line of data and it overwrites the existing line with this given new data.

3. SimulateCache:

- The *SimulateCache* function takes the cache size, block size, replacement policy, and the associativity as input and acts as the top level controller block to simulate the address trace.
- It reads the translated addresses one-by-one and provides it to ***Cache*** block for *lookup.* If the lookup is successful, the function prompts the ***Performance Recorder*** to update the hitCount. If the lookup is unsuccessful, it asks to update the missCount.

4. ReplacementPolicy

- The *ReplacementPolicy* class implements the LRU/FIFO/Random replacement policies. It keeps track of which line entered the first in a set for tracking FIFO policy. It keeps track of how many reads are performed on each line of set to keep track of LRU replacement policy. When ***CacheSet*** asks the ***ReplacementPolicy*** to return a line to be replaced, it returns the first line entered(for FIFO) or the least line used(for LRU). In case of Random policy, it selects one of the line at random and returns it.

5. PerformanceRecorder:

- The *PerformanceRecorder* keeps track of address hits and misses reported from the ***lookup*** function. At the end it returns the hit ratio that is determined by below equation.

$$Cache\ Hit\ Ratio\ = \frac{Total\ Number\ of\ Cache\ hits}{Total\ Number\ of\ Cache\ Hits\ +\ Number\ of\ Cache\ Misses}$$

4. Simulation Results

4.1 Simulation Plots

4.1.1 Scenario 1 : Variable Cache Size

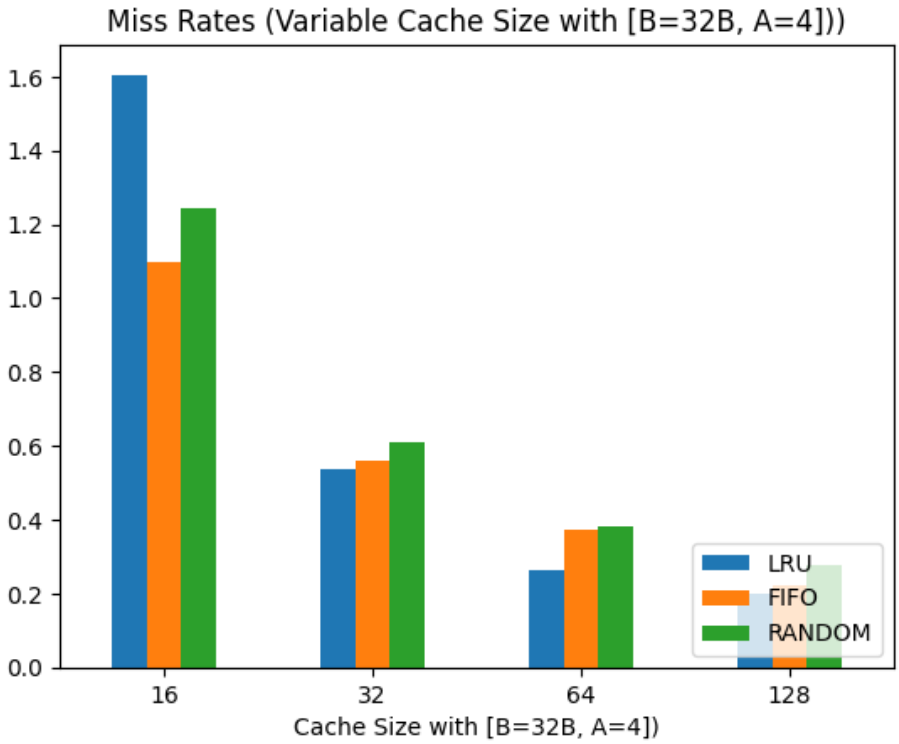

Fig. 7 Miss Rates Plot for Variable Cache Size (Scenario-1)

Cache Size	LRU	FIFO	Random
16	1.605067	1.098467	1.257733
32	0.536667	0.561667	0.607267
64	0.265000	0.374800	0.387933
128	0.197467	0.222000	0.272400

For varying cache size, the miss rate decreased with increasing cache size. This occurs because the larger cache size will allow more number of lines to be accommodate, which will **reduce the capacity misses** and therefore improve the overall performance.

Performance Rank:

1. LRU
2. FIFO
3. Random

4.1.2 Scenario 2 : Variable Block Size

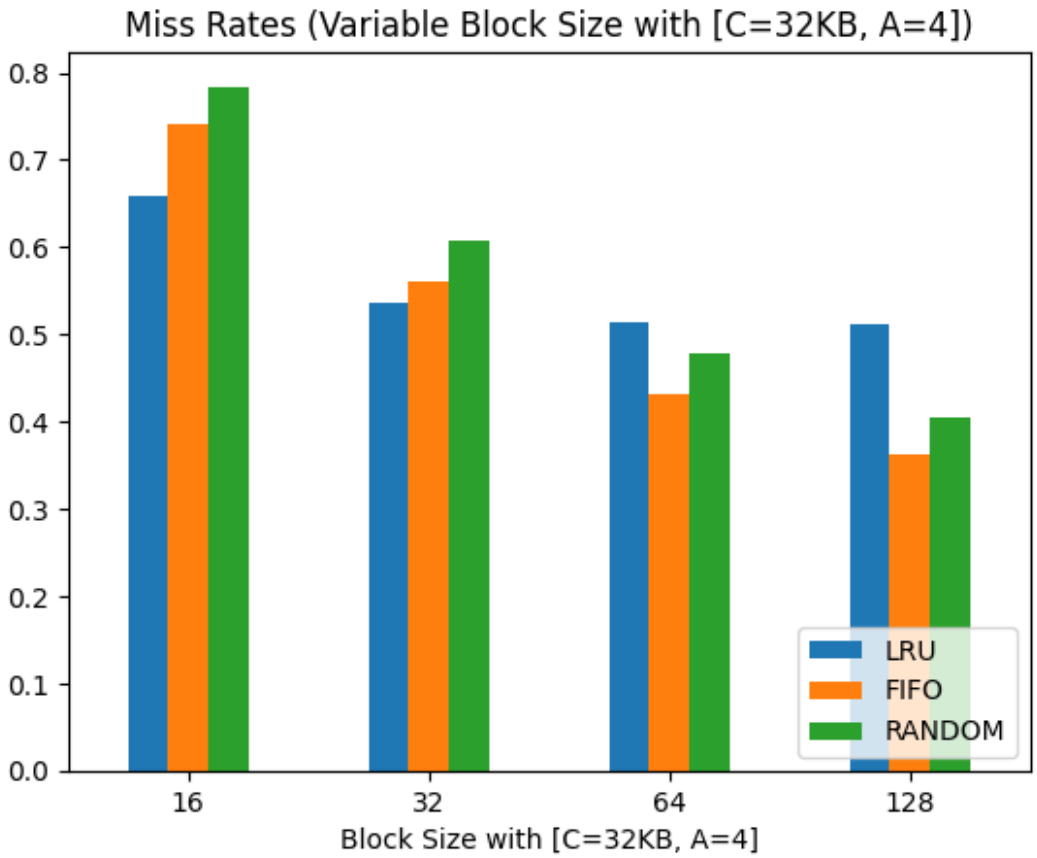

Fig. 8 Miss Rates for Variable Block Size (Scenario-2)

Block Size	LRU	FIFO	Random
16	0.659133	0.740800	0.774467
32	0.536667	0.561667	0.604667
64	0.514000	0.431333	0.473133
128	0.511933	0.362400	0.410067

For varying block size, the miss rate also decreased with increasing block size. This occurs because the block cache size will allow more number of contagious blocks to be accommodate, which will **increase the spatial locality**. Therefore it will increase the hit rate for contagious memory reads such as array iterations.

Performance Rank:

1/2. LRU (for lower block sizes)

1/2. FIFO (for higher block sizes)

3. Random

4.1.3 Scenario 3 : Variable Associativity

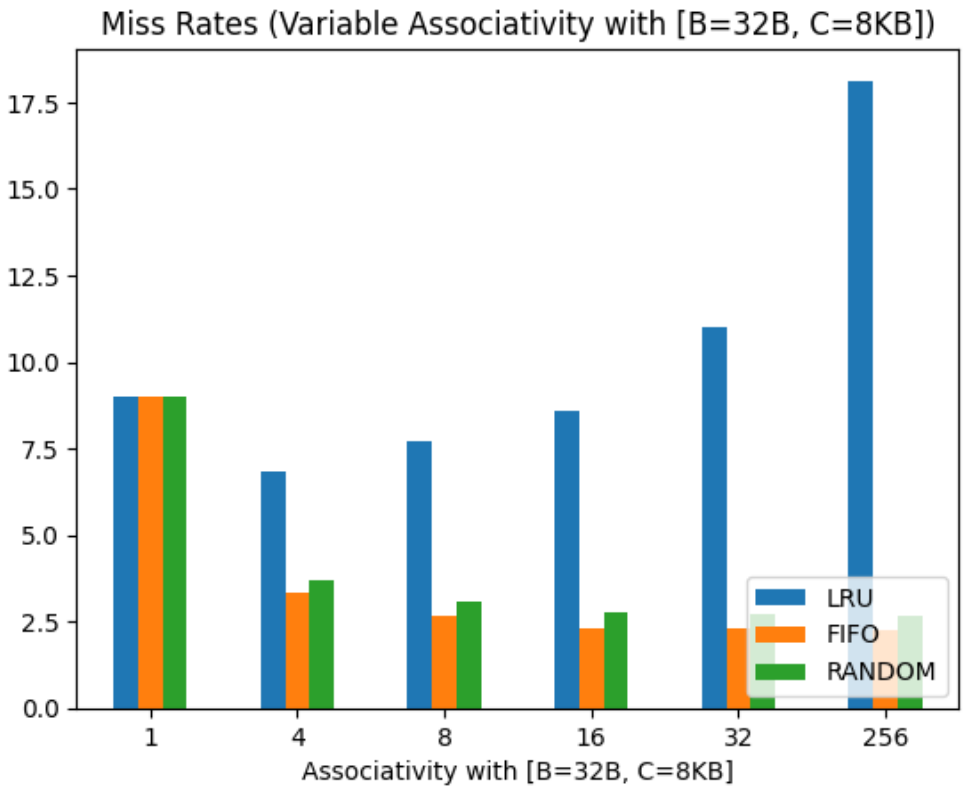

Fig. 9 Miss Rates for Variable Associativity (Scenario-3)

Associativity	LRU	FIFO	Random
1	8.983200	8.983200	8.983200
4	6.819400	3.341733	3.718533
8	7.700000	2.649400	3.087667
16	8.579200	2.323867	2.748200
32	11.008933	2.291000	2.736133
256	18.127000	2.236867	2.744133

For varying associativity size, the miss rate also decreased with increasing associativity. This occurs because the higher associativity will **reduce the conflict misses**.

However, in case of LRU, the miss rate reduced only till the associativity of 4. After that, increased associativity provided a negative performance. This could be the reason why the **associativity of 4 and 8 are more common implementation for LRU in the modern processors.**

Performance Rank:

1. FIFO
2. LRU
3. Random

4.2 Simulation Log

Scenario-1 : fixed(block-size=32-Bytes, Associativity:4) Varying(Cache-size: 16KB to 128KB)

```
---------------

Cache Size: 16K Block Size: 32 Associativity: 4 Policy: LRU
TAG-bits: 20 SET-bits: 7 OFFSET-bits: 5
Total: 1500000 Hit: 1475924 Miss: 24076 hitRate(%)= 98.395

Cache Size: 16K Block Size: 32 Associativity: 4 Policy: FIFO
TAG-bits: 20 SET-bits: 7 OFFSET-bits: 5
Total: 1500000 Hit: 1483523 Miss: 16477 hitRate(%)= 98.902

Cache Size: 16K Block Size: 32 Associativity: 4 Policy: RANDOM
TAG-bits: 20 SET-bits: 7 OFFSET-bits: 5
Total: 1500000 Hit: 1481134 Miss: 18866 hitRate(%)= 98.742
---------------

Cache Size: 32K Block Size: 32 Associativity: 4 Policy: LRU
TAG-bits: 19 SET-bits: 8 OFFSET-bits: 5
Total: 1500000 Hit: 1491950 Miss: 8050 hitRate(%)= 99.463

Cache Size: 32K Block Size: 32 Associativity: 4 Policy: FIFO
TAG-bits: 19 SET-bits: 8 OFFSET-bits: 5
Total: 1500000 Hit: 1491575 Miss: 8425 hitRate(%)= 99.438

Cache Size: 32K Block Size: 32 Associativity: 4 Policy: RANDOM
TAG-bits: 19 SET-bits: 8 OFFSET-bits: 5
Total: 1500000 Hit: 1490891 Miss: 9109 hitRate(%)= 99.393
---------------

Cache Size: 64K Block Size: 32 Associativity: 4 Policy: LRU
TAG-bits: 18 SET-bits: 9 OFFSET-bits: 5
Total: 1500000 Hit: 1496025 Miss: 3975 hitRate(%)= 99.735

Cache Size: 64K Block Size: 32 Associativity: 4 Policy: FIFO
TAG-bits: 18 SET-bits: 9 OFFSET-bits: 5
Total: 1500000 Hit: 1494378 Miss: 5622 hitRate(%)= 99.625

Cache Size: 64K Block Size: 32 Associativity: 4 Policy: RANDOM
TAG-bits: 18 SET-bits: 9 OFFSET-bits: 5
Total: 1500000 Hit: 1494181 Miss: 5819 hitRate(%)= 99.612
---------------

Cache Size: 128K Block Size: 32 Associativity: 4 Policy: LRU
TAG-bits: 17 SET-bits: 10 OFFSET-bits: 5
Total: 1500000 Hit: 1497038 Miss: 2962 hitRate(%)= 99.803

Cache Size: 128K Block Size: 32 Associativity: 4 Policy: FIFO
TAG-bits: 17 SET-bits: 10 OFFSET-bits: 5
Total: 1500000 Hit: 1496670 Miss: 3330 hitRate(%)= 99.778

Cache Size: 128K Block Size: 32 Associativity: 4 Policy: RANDOM
TAG-bits: 17 SET-bits: 10 OFFSET-bits: 5
Total: 1500000 Hit: 1495914 Miss: 4086 hitRate(%)= 99.728
```

Scenario-2 : fixed(cache-size=32KB, Associativity:4) Varying(Block-size: 16-Bytes to 128-Bytes)

```
---------------

Cache Size: 32K Block Size: 16 Associativity: 4 Policy: LRU
TAG-bits: 19 SET-bits: 9 OFFSET-bits: 4
Total: 1500000 Hit: 1490113 Miss: 9887 hitRate(%)= 99.341

Cache Size: 32K Block Size: 16 Associativity: 4 Policy: FIFO
TAG-bits: 19 SET-bits: 9 OFFSET-bits: 4
Total: 1500000 Hit: 1488888 Miss: 11112 hitRate(%)= 99.259

Cache Size: 32K Block Size: 16 Associativity: 4 Policy: RANDOM
TAG-bits: 19 SET-bits: 9 OFFSET-bits: 4
```

```
Total: 1500000 Hit: 1488383 Miss: 11617 hitRate(%)= 99.226
---------------

Cache Size: 32K Block Size: 32 Associativity: 4 Policy: LRU
TAG-bits: 19 SET-bits: 8 OFFSET-bits: 5
Total: 1500000 Hit: 1491950 Miss: 8050 hitRate(%)= 99.463

Cache Size: 32K Block Size: 32 Associativity: 4 Policy: FIFO
TAG-bits: 19 SET-bits: 8 OFFSET-bits: 5
Total: 1500000 Hit: 1491575 Miss: 8425 hitRate(%)= 99.438

Cache Size: 32K Block Size: 32 Associativity: 4 Policy: RANDOM
TAG-bits: 19 SET-bits: 8 OFFSET-bits: 5
Total: 1500000 Hit: 1490930 Miss: 9070 hitRate(%)= 99.395
---------------

Cache Size: 32K Block Size: 64 Associativity: 4 Policy: LRU
TAG-bits: 19 SET-bits: 7 OFFSET-bits: 6
Total: 1500000 Hit: 1492290 Miss: 7710 hitRate(%)= 99.486

Cache Size: 32K Block Size: 64 Associativity: 4 Policy: FIFO
TAG-bits: 19 SET-bits: 7 OFFSET-bits: 6
Total: 1500000 Hit: 1493530 Miss: 6470 hitRate(%)= 99.569

Cache Size: 32K Block Size: 64 Associativity: 4 Policy: RANDOM
TAG-bits: 19 SET-bits: 7 OFFSET-bits: 6
Total: 1500000 Hit: 1492903 Miss: 7097 hitRate(%)= 99.527
---------------

Cache Size: 32K Block Size: 128 Associativity: 4 Policy: LRU
TAG-bits: 19 SET-bits: 6 OFFSET-bits: 7
Total: 1500000 Hit: 1492321 Miss: 7679 hitRate(%)= 99.488

Cache Size: 32K Block Size: 128 Associativity: 4 Policy: FIFO
TAG-bits: 19 SET-bits: 6 OFFSET-bits: 7
Total: 1500000 Hit: 1494564 Miss: 5436 hitRate(%)= 99.638

Cache Size: 32K Block Size: 128 Associativity: 4 Policy: RANDOM
TAG-bits: 19 SET-bits: 6 OFFSET-bits: 7
Total: 1500000 Hit: 1493849 Miss: 6151 hitRate(%)= 99.590
```

Scenario-3 : fixed(block-size=32-Bytes, cache-size:8KB) Varying(Associativity: 1, 2,.. to fully-associative)

```
---------------

Cache Size: 8K Block Size: 32 Associativity: 1 Policy: LRU
TAG-bits: 19 SET-bits: 8 OFFSET-bits: 5
Total: 1500000 Hit: 1365252 Miss: 134748 hitRate(%)= 91.017

Cache Size: 8K Block Size: 32 Associativity: 1 Policy: FIFO
TAG-bits: 19 SET-bits: 8 OFFSET-bits: 5
Total: 1500000 Hit: 1365252 Miss: 134748 hitRate(%)= 91.017

Cache Size: 8K Block Size: 32 Associativity: 1 Policy: RANDOM
TAG-bits: 19 SET-bits: 8 OFFSET-bits: 5
Total: 1500000 Hit: 1365252 Miss: 134748 hitRate(%)= 91.017
---------------

Cache Size: 8K Block Size: 32 Associativity: 4 Policy: LRU
TAG-bits: 21 SET-bits: 6 OFFSET-bits: 5
Total: 1500000 Hit: 1397709 Miss: 102291 hitRate(%)= 93.181

Cache Size: 8K Block Size: 32 Associativity: 4 Policy: FIFO
TAG-bits: 21 SET-bits: 6 OFFSET-bits: 5
Total: 1500000 Hit: 1449874 Miss: 50126 hitRate(%)= 96.658

Cache Size: 8K Block Size: 32 Associativity: 4 Policy: RANDOM
TAG-bits: 21 SET-bits: 6 OFFSET-bits: 5
Total: 1500000 Hit: 1444222 Miss: 55778 hitRate(%)= 96.281
---------------
```

```
Cache Size: 8K Block Size: 32 Associativity: 8 Policy: LRU
TAG-bits: 22 SET-bits: 5 OFFSET-bits: 5
Total: 1500000 Hit: 1384500 Miss: 115500 hitRate(%)= 92.300

Cache Size: 8K Block Size: 32 Associativity: 8 Policy: FIFO
TAG-bits: 22 SET-bits: 5 OFFSET-bits: 5
Total: 1500000 Hit: 1460259 Miss: 39741 hitRate(%)= 97.351

Cache Size: 8K Block Size: 32 Associativity: 8 Policy: RANDOM
TAG-bits: 22 SET-bits: 5 OFFSET-bits: 5
Total: 1500000 Hit: 1453685 Miss: 46315 hitRate(%)= 96.912
---------------

Cache Size: 8K Block Size: 32 Associativity: 16 Policy: LRU
TAG-bits: 23 SET-bits: 4 OFFSET-bits: 5
Total: 1500000 Hit: 1371312 Miss: 128688 hitRate(%)= 91.421

Cache Size: 8K Block Size: 32 Associativity: 16 Policy: FIFO
TAG-bits: 23 SET-bits: 4 OFFSET-bits: 5
Total: 1500000 Hit: 1465142 Miss: 34858 hitRate(%)= 97.676

Cache Size: 8K Block Size: 32 Associativity: 16 Policy: RANDOM
TAG-bits: 23 SET-bits: 4 OFFSET-bits: 5
Total: 1500000 Hit: 1458777 Miss: 41223 hitRate(%)= 97.252
---------------

Cache Size: 8K Block Size: 32 Associativity: 32 Policy: LRU
TAG-bits: 24 SET-bits: 3 OFFSET-bits: 5
Total: 1500000 Hit: 1334866 Miss: 165134 hitRate(%)= 88.991

Cache Size: 8K Block Size: 32 Associativity: 32 Policy: FIFO
TAG-bits: 24 SET-bits: 3 OFFSET-bits: 5
Total: 1500000 Hit: 1465635 Miss: 34365 hitRate(%)= 97.709

Cache Size: 8K Block Size: 32 Associativity: 32 Policy: RANDOM
TAG-bits: 24 SET-bits: 3 OFFSET-bits: 5
Total: 1500000 Hit: 1458958 Miss: 41042 hitRate(%)= 97.264
---------------

Cache Size: 8K Block Size: 32 Associativity: 256 Policy: LRU
TAG-bits: 27 SET-bits: 0 OFFSET-bits: 5
Total: 1500000 Hit: 1228095 Miss: 271905 hitRate(%)= 81.873

Cache Size: 8K Block Size: 32 Associativity: 256 Policy: FIFO
TAG-bits: 27 SET-bits: 0 OFFSET-bits: 5
Total: 1500000 Hit: 1466447 Miss: 33553 hitRate(%)= 97.763

Cache Size: 8K Block Size: 32 Associativity: 256 Policy: RANDOM
TAG-bits: 27 SET-bits: 0 OFFSET-bits: 5
Total: 1500000 Hit: 1458838 Miss: 41162 hitRate(%)= 97.256
```

4.3 Performance Ranking Table

Rank	Cache Size	Block Size		Associativity	
		< 64 Bytes	>= 64 Bytes	< 8	>= 8
1	LRU	LRU	FIFO	LRU	FIFO
2	FIFO	FIFO	Random	FIFO	Random
3	Random	Random	LRU	Random	LRU
Best	**LRU**	**LRU**	**FIFO**	**LRU**	**FIFO**

5. Conclusion

The Cache Simulator was successfully implemented in Python with variable cache size, block size and associativity for different cache replacement policy. The hit rate for each of the configuration was measured for a quantitative analysis.

The project gave a good understanding of how the cache design parameters including the replacement policies affect the cache performance and why it is critical to choose the optimal parameters. It was observed that the LRU policy gave the best performance in most cases. However, the cache design parameters need to be properly selected because the higher associativity and higher block size results in a lower performance for LRU compared to other policies. The miss rate reduced till the associativity reached 8 and then increased thereafter. It could be the reason why associativity of 4 and 8 are most common implementation for modern processors. FIFO policy was the second best, however, unlike LRU it did not result in a negative performance with increased block size or associativity. The Random policy provided the highest miss rate in most cases.

Increase in cache size improved performance in all cases. However, it comes with additional latency which is not the scope of this project. The Increase in block size gave better performance for all policies except LRU. The same can be said for increased associativity, where FIFO and Random produced better results whereas LRU provided optimal miss rate only between associativity of 4 to 8.

APPENDIX

Python Code

```
#!/usr/bin/env python
# coding: utf-8

##########################################
# # EE275 Mini Project 3 (Fall 2022)

# #### Project: Cache Simulator

# #### Created By: Bhavin Patel (015954770)
##########################################

import math
import random
from enum import Enum
import matplotlib.pyplot as plt
import pandas as pd

trc_limit=1.0 # % of trace to process [ between 0.0 to 1.0], reduce it for shorter simulation


##########################################
# #### Reading Address Trace
##########################################

lines = ()
with open("addr_trace.txt", "r") as f:
    lines = f.read().splitlines();

addr_trace = [int(i) for i in lines]
#print (addr_trace)


##########################################
# #### Translating Trace
##########################################
prevAddr = 0
for idx in range(len(addr_trace)):
    addr_trace[idx] += prevAddr
    prevAddr = addr_trace[idx]
#print (addr_trace)


##########################################
# ### Main Memory Implementation
```

```
45  ##########################################
46
47  # Main memory reads are supported simply by returning a line of random bytes
48  def getRandDataLine(b_size):
49      line = []
50      for i in range(b_size):
51          line.append(random.randint(0,pow(2,8))) ## every location holds 8-bit of
    data
52      return line
53
54
55  ##########################################
56  # ### Performance Recorder
57  ##########################################
58
59  ## Recorder tracks all the hits/misses
60  class Recorder:
61
62      def __init__(self):
63          self.history = []
64          self.hitCount = 0;
65          self.missCount = 0;
66          self.hitRate = 0.0;
67          self.total = 0;
68
69      # called when after a read is processed
70      def record(self, addr, hit):
71          self.history.append({'addr': addr, 'hit': hit});
72          if(hit):
73              self.hitCount += 1
74          else:
75              self.missCount += 1
76          self.hitRate = (self.hitCount / (self.hitCount + self.missCount)) * 100.0
77          self.total = self.hitCount + self.missCount
78
79      # show total hits and misses
80      def showRecord(self):
81          print("Total:", self.total,
82                "Hit:", self.hitCount,
83                "Miss:", self.missCount,
84                "hitRate(%)=", '{0:.3f}'.format(self.hitRate))
85
86      # show hits and misses for each address reads in trace
87      def showHistory(self):
88          for item in self.history:
89              print('{0:      >15}'.format(item['addr']),      '      ',      '{0:
    >5}'.format(item['hit']))
90
91
92
93
94  ##########################################
95  # ### Replacement Policy Manager
```

```
96  ##########################################
97
98  # Enum for replacement policies
99  class Policy(Enum):
100     LRU = 1
101     FIFO = 2
102     RANDOM = 3
103
104 # ReplacementPolicy: Keeps track of all the reads for LRU, FIFO and Random
    policies
105 class ReplacementPolicy:
106
107     def __init__(self, associativity, policy):
108         self.associativity = associativity
109         self.policy = policy
110         self.entryQueue = []
111         self.usedQueue = []
112         for i in range(associativity):
113             self.usedQueue.append(0)
114
115     # called when a line is loaded from MM to Cache
116     def reportLoad(self, lineNum):
117
118         # for FIFO : update the FIFO queue
119         self.entryQueue.append(lineNum)
120         if(len(self.entryQueue) > self.associativity):
121             self.entryQueue.pop(0)
122
123         #for LRU : initialize the 'used count' for the loaded line
124         self.usedQueue[lineNum] = 1
125
126     # called when a line in cache needs to be replaced
127     # returns the line number to replace
128     def getLoadableIdx(self):
129
130         # for FIFO: pop from the front of FIFO queue
131         if(self.policy == Policy.FIFO):
132             if(len(self.entryQueue) == 0):
133                 return 0
134             elif(len(self.entryQueue) < self.associativity):
135                 return len(self.entryQueue)
136             else:
137                 return self.entryQueue[0]
138
139         # for LRU: get the line that has the lowest 'used count'
140         elif(self.policy == Policy.LRU):
141             tmp = min(self.usedQueue)
142             return self.usedQueue.index(tmp)
143
144         # for RANDOM: just return the random line
145         else:
146             return random.randint(0,self.associativity-1)
147
```

```
148     # called when a line is read
149     # LRU updates the line count accordingly
150     def reportLookup(self, lineNum):
151         self.usedQueue[lineNum] += 1
152 
153 
154 ##########################################
155 # ## Cache Implementation
156 ##########################################
157 
158 # #### 1. Cache Structure
159 
160 ######## a single cache Line ########
161 class CacheLine:
162 
163     def __init__(self, blockSize, associativity):
164         self.cBlocks = []
165         self.tag = 0;
166         self.associativity = associativity
167         for i in range(blockSize):
168             self.cBlocks.append(0)
169 
170     # show the line
171     def show(self, setNum, lineNum):
172         frmt = "{:>5}"*len(self.cBlocks)
        print("set=",    '{0:    >5}'.format(setNum),    "    |    line=",    '{0:
173 >5}'.format(lineNum    +    setNum    *    self.associativity),    "    |    ",
    frmt.format(*self.cBlocks))
174 
175     # find given block in the cache line
176     def lookup(self, lookupObj):
177         lookupObj.data = self.cBlocks[lookupObj.offset]
178 
179     # load the cache line with new data
180     def load(self, loadObj):
181         self.tag = loadObj.tag
182         self.cBlocks = loadObj.dataList
183 
184 ######## a single set of Cache ########
185 class CacheSet:
186 
187     def __init__(self, linesInSet, blockSize, associativity, policy):
188         # create multiple cache lines
189         self.rPolicy = ReplacementPolicy(associativity, policy)
190         self.cLines = []
191         self.associativity = associativity
192         for i in range(linesInSet):
193             self.cLines.append(CacheLine(blockSize, associativity))
194 
195     # show the set
196     def show(self, setNum):
197         for lineNum in range(len(self.cLines)):
198             self.cLines[lineNum].show(setNum, lineNum)
```

```
199
200     # find the given line in this set
201     def lookup(self, lookupObj):
202         lookupObj.hit = False;
203         lookupObj.data = 0;
204         for i in range(len(self.cLines)):
205             if(self.cLines[i].tag == lookupObj.tag): #hit
206                 lookupObj.hit = True
207                 lookupObj.line = i
208                 self.cLines[i].lookup(lookupObj)
209                 self.rPolicy.reportLookup(i)
210
211     # load one of the lines in set with given data
212     def load(self, loadObj):
213         loadIdx = self.rPolicy.getLoadableIdx()
214         self.cLines[loadIdx].load(loadObj)
215         self.rPolicy.reportLoad(loadIdx)
216
217
218 ######## Main Cache ########
219 class Cache:
220
221     def __init__(self, cacheSize, blockSize, associativity, policy):
222         self.recorder=Recorder();
223         # create multiple cache sets
224         numOfLines = cacheSize // blockSize
225         numOfSets = numOfLines // associativity
226         linesInSet = numOfLines // numOfSets
227         self.cSets = []
228         for i in range(numOfSets):
229             self.cSets.append(CacheSet(linesInSet,  blockSize,  associativity,
policy))
230
231     # show the cache contents
232     def show(self):
233         for setNum in range(len(self.cSets)):
234             print("---------------------------")
235             self.cSets[setNum].show(setNum)
236
237     # find the given physical address in cache
238     def lookup(self, lookupObj):
239         self.cSets[lookupObj.set].lookup(lookupObj)
240         self.recorder.record(lookupObj.addr, lookupObj.hit)
241
242     # load the given data in cache line
243     def load(self, loadObj):
244         self.cSets[loadObj.set].load(loadObj)
245
246
247
248 # #### 2. Cache Lookup and Load information
249
```

```
250 ### Lookup Object: this object holds all the information about an instance of
    cache read operation
251 class LookUpObj:
252     def __init__(self,  addr, n_tag, n_set, n_offset):
253         self.addr = addr
254         self.tag = int(addr) >> (n_set + n_offset) & ((1 << n_tag) - 1)
255         self.set = int(addr) >> (n_offset) & ((1 << n_set) - 1)
256         self.offset = int(addr) & ((1 << n_offset) - 1)
257         self.data = 0
258         self.line = -1
259         self.hit = False
260 
261     def show(self):
262         print("addr=", '{0: >10}'.format(self.addr),
263               "TAG: ", '{0: >5}'.format(self.tag),
264               "SET:", '{0: >5}'.format(self.set),
265               "OFFSET:", '{0: >5}'.format(self.offset),
266               " | data=", '{0: >5}'.format(self.data),
267               "line=", '{0: >5}'.format(self.line),
268               "hit:", '{0: >5}'.format(self.hit))
269 
270 ### Load Object: this object holds all the information about an instance of data
    to be loaded in a cache
271 class LoadObj:
272     def __init__(self,  addr, dataList, n_tag, n_set, n_offset):
273         self.addr = addr
274         self.tag = int(addr) >> (n_set + n_offset) & ((1 << n_tag) - 1)
275         self.set = int(addr) >> (n_offset) & ((1 << n_set) - 1)
276         self.offset = int(addr) & ((1 << n_offset) - 1)
277         self.dataList = dataList
278 
279     def show(self):
280         print("addr=", '{0: >10}'.format(self.addr),
281               "TAG: ", '{0: >5}'.format(self.tag),
282               "SET:", '{0: >5}'.format(self.set),
283               "OFFSET:", '{0: >5}'.format(self.offset),
284               " | data=", *self.dataList)
285 
286 
287 ##########################################
288 # ## Parameterized Cache Simulation Function
289 ##########################################
290 
291 ### Main function to simulate the cache for given cache parameters:
292 #   1. cache size
293 #   2. block size.
294 #   3. associativity.
295 #   4. replacement policy
296 
297 def simulateCache(cSize, bSize, assoc, pol):
298 
299     ############## set-up the cache parameters ###########
300     CACHE_SIZE=cSize * 1024 #bytes
```

```
301     BLOCK_SIZE=bSize #bytes
302     ASSOCIATIVITY=assoc
303     POLICY=pol
304     print ("")
305     print ("Cache Size: {0}K".format(cSize), "Block Size:", bSize, "Associativity:", assoc, "Policy:", POLICY.name)
306
307     ## number of address bits, offset bits, set bits, tag bits
308     ADDR_SIZE=32
309     N_OFFSET=int(math.log2(BLOCK_SIZE))
310     N_SET=int(math.log2(((CACHE_SIZE//BLOCK_SIZE)//ASSOCIATIVITY)))
311     N_TAG=ADDR_SIZE - N_OFFSET - N_SET
312     print ("TAG-bits:", N_TAG, "SET-bits:", N_SET, "OFFSET-bits:", N_OFFSET, )
313
314     ############### create cache ##############
315     cache = Cache(CACHE_SIZE, BLOCK_SIZE, ASSOCIATIVITY, POLICY)
316     #cache.show()
317
318     ############### look for each addresses given in trace ##############
319     for idx in range(len(addr_trace)):
320         lookupObj = LookUpObj(addr_trace[idx], N_TAG, N_SET, N_OFFSET)
321         #lookupObj.show()
322         cache.lookup(lookupObj)
323         #lookupObj.show()
324
325         # load data if you get a miss
326         if(lookupObj.hit == False):
327             loadDataLine = getRandDataLine(BLOCK_SIZE)
328             loadObj = LoadObj(lookupObj.addr, loadDataLine, N_TAG, N_SET, N_OFFSET)
329             #print("loading data line(to set {0}): {1}".format(loadObj.set, loadDataLine))
330             cache.load(loadObj)
331
332         # limited to first N address for testing
333         if(idx >= (len(addr_trace) * trc_limit)):
334             break
335
336     # show cache again to see that the data loaded
337     #cache.show()
338
339     # show statistics
340     cache.recorder.showRecord()
341     #cache.recorder.showHistory()
342
343     return cache.recorder.hitRate
344
345
346
347 # ## Simulate Cache for Different Combinations of: cache-size, block-size, associativity
348
349 ##########################################
```

```
# ######## Simulate cache for different combinations of each parameters
##########################################

cache_size_KB = [16, 32, 64, 128]
block_size_B = [16, 32, 64, 128]
associativity_N=[1,4,8, 16, 32]
policies=[Policy.LRU, Policy.FIFO, Policy.RANDOM]

## function to plot a chart
def plotHits(ht, label, colName):
    df = pd.DataFrame(ht)
    df.set_index(colName, inplace=True)
    display(df)
    df.plot.bar(rot=0)
    plt.title("Miss Rates ({0})".format(label))
    plt.legend(loc='upper right')
    plt.show()

############  Scenario 1  ############
print("Scenario-1 : fixed(block-size=32-Bytes, Associativity:4) Varying(Cache-size: 16KB to 128KB)")
allHits = []
label = 'Scenario-1: Variable Cache Size with [Block=32B, Assoc=4])'
colName = 'Cache Size'
for cs in cache_size_KB:
    print ("---------------")
    hits = {}
    for pol in policies:
        hits[pol.name] = 100.0 - simulateCache(cs, 32, 4, pol)
    hits[colName] = cs
    allHits.append(hits)
plotHits(allHits, label, colName)

############  Scenario 2  ############
print ("Scenario-2 : fixed(cache-size=32KB, Associativity:4) Varying(Block-size: 16-Bytes to 128-Bytes)")
allHits = []
label = 'Scenario-2: Variable Block Size with [Cache=32KB, Assoc=4]'
colName = 'Block Size'
for bs in block_size_B:
    print ("---------------")
    hits = {}
    for pol in policies:
        hits[pol.name] = 100.0 - simulateCache(32, bs, 4, pol)
    hits[colName] = bs
    allHits.append(hits)
plotHits(allHits, label, colName)

############  Scenario 3  ############
print("Scenario-3 : fixed(block-size=32-Bytes, cache-size:8KB) Varying(Associativity: 1, 2,.. to fully-associative)")
allHits = []
label = 'Scenario-3: Variable Associativity with [Block=32B, Cache=8KB]'
```

```
colName = 'Associativity'
associativity_N.append((8 * 1024) // 32) #append full-assoc.: 8KB(cache-size)/32B[block-size]
for assoc in associativity_N:
    print ("----------------")
    hits = {}
    for pol in policies:
        hits[pol.name] = 100.0 - simulateCache(8, 32, assoc, pol)
    hits[colName] = assoc
    allHits.append(hits)
plotHits(allHits, label, colName)
```