ENG 463-L1

Lab #1 Cache Simulator

Brian Worts and Chris Jenson

02/23/21



**Table of Contents**

Introduction………………………………………………………………………………………. 2

Problem Statement…...…………………………………………………………………………... 2

Procedure/Results…...……...…...……………………………………………………………….. 2

Discussion………………………………………………………………………………………. 9

Conclusion ................................................................................................................................... 9

Appendix A (Python Code).......................................................................................................... 10

**Intro:**

Caches are an essential part of modern computer systems and are best described as a hardware or software component that stores data so that future requests for that data can be served faster since accessing a cache is faster than accessing memory. One can judge a cache's performance on the miss/hit ratios where a hit means that the desired data already exists in the cache. Caches can have various configurations and policies to best serve different specifications, and this lab looks at these various policies and configurations to determine how they affect the performance of a cache.

**Problem Statement:**

Using the provided trace files, study, via simulation, the performance of memory management policies and memory configurations for cache memory systems.

**Procedure/Results:**

The first step of the process was to complete a flow diagram for the simulator, shown in *Figure 1.*

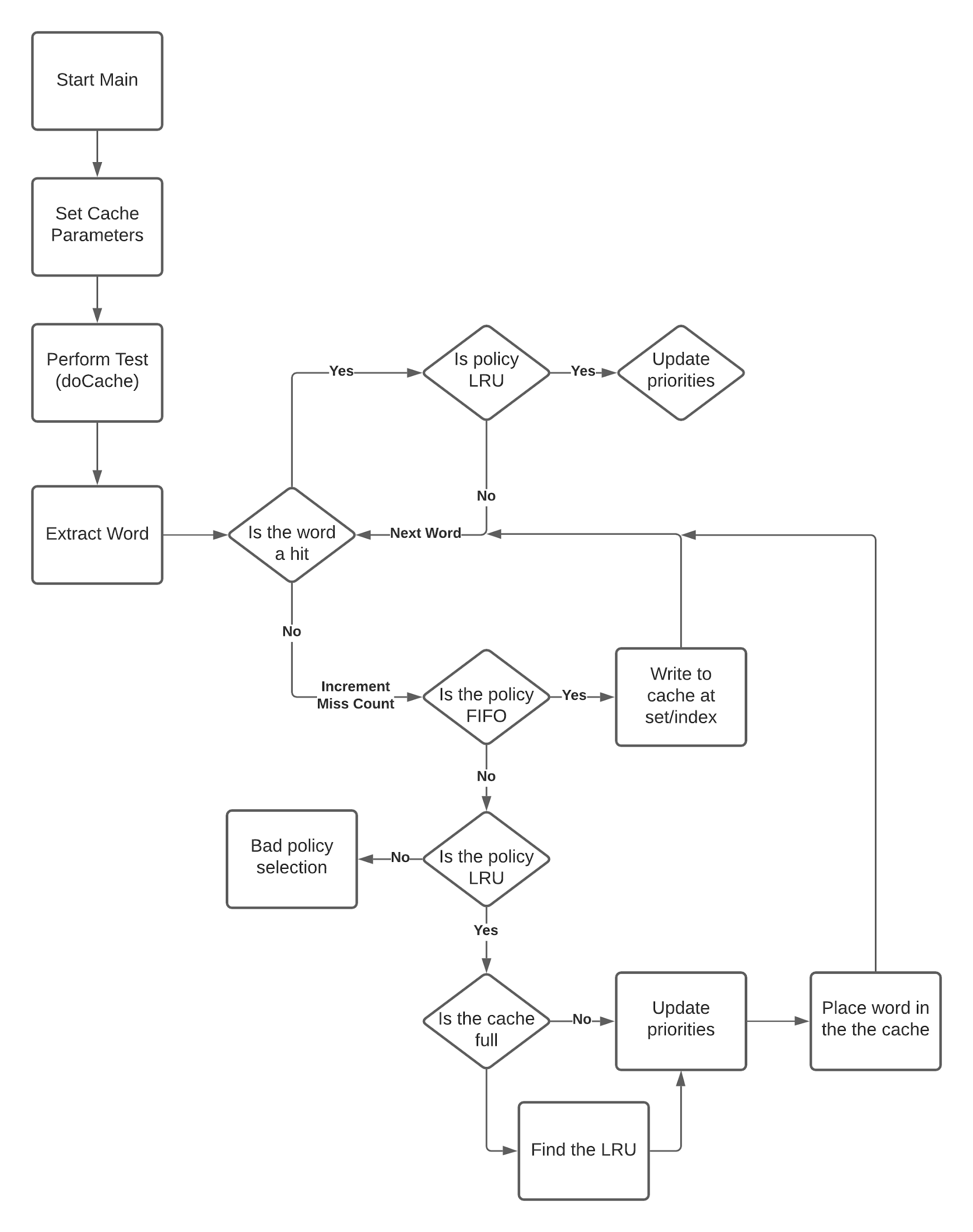
**

Figure 1: Flow Diagram

Once the simulation was complete, its results were obtained, shown in *Table 1* for TRACE1.dat and in *Table 2* for TRACE2.dat*.*

Table 1: Trace 1 Results

|  |  |  |  |
| --- | --- | --- | --- |
| **Replacement Policy** | **KN** | **K** | **Miss Rate** |
| FIFO | 64 | 2 | 3.3466 |
| FIFO | 64 | 4 | 3.8269 |
| FIFO | 64 | 8 | 4.5004 |
| FIFO | 64 | 16 | 5.6788 |
| FIFO | 256 | 2 | 1.9686 |
| FIFO | 256 | 4 | 2.0968 |
| FIFO | 256 | 8 | 2.3607 |
| FIFO | 256 | 16 | 3.0298 |
| LRU | 64 | 2 | 3.2656 |
| LRU | 64 | 4 | 3.7403 |
| LRU | 64 | 8 | 4.4589 |
| LRU | 64 | 16 | 5.6855 |
| LRU | 256 | 2 | 1.8887 |
| LRU | 256 | 4 | 2.0422 |
| LRU | 256 | 8 | 2.2960 |
| LRU | 256 | 16 | 2.9661 |

Table 2: Trace 2 Results

|  |  |  |  |
| --- | --- | --- | --- |
| **Replacement Policy** | **KN** | **K** | **Miss Rate** |
| FIFO | 64 | 2 | 8.2384 |
| FIFO | 64 | 4 | 8.3402 |
| FIFO | 64 | 8 | 8.3639 |
| FIFO | 64 | 16 | 8.6026 |
| FIFO | 256 | 2 | 5.1833 |
| FIFO | 256 | 4 | 5.5055 |
| FIFO | 256 | 8 | 5.7328 |
| FIFO | 256 | 16 | 5.9212 |
| LRU | 64 | 2 | 8.1729 |
| LRU | 64 | 4 | 8.3021 |
| LRU | 64 | 8 | 8.3734 |
| LRU | 64 | 16 | 8.5752 |
| LRU | 256 | 2 | 5.0005 |
| LRU | 256 | 4 | 5.4295 |
| LRU | 256 | 8 | 5.6816 |
| LRU | 256 | 16 | 5.9204 |

Lastly, the simulation results were analyzed with graphs, shown in *Figures 2-7.*

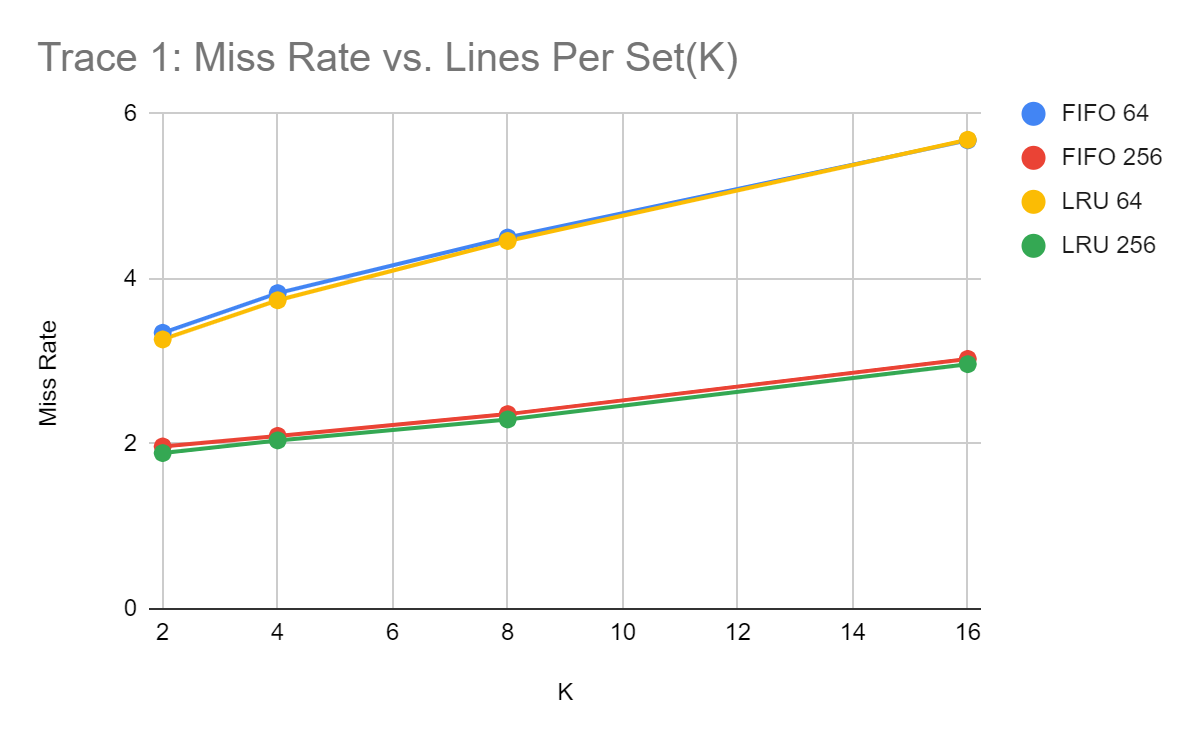
**

Figure 2: Miss Rate vs K for Trace 1

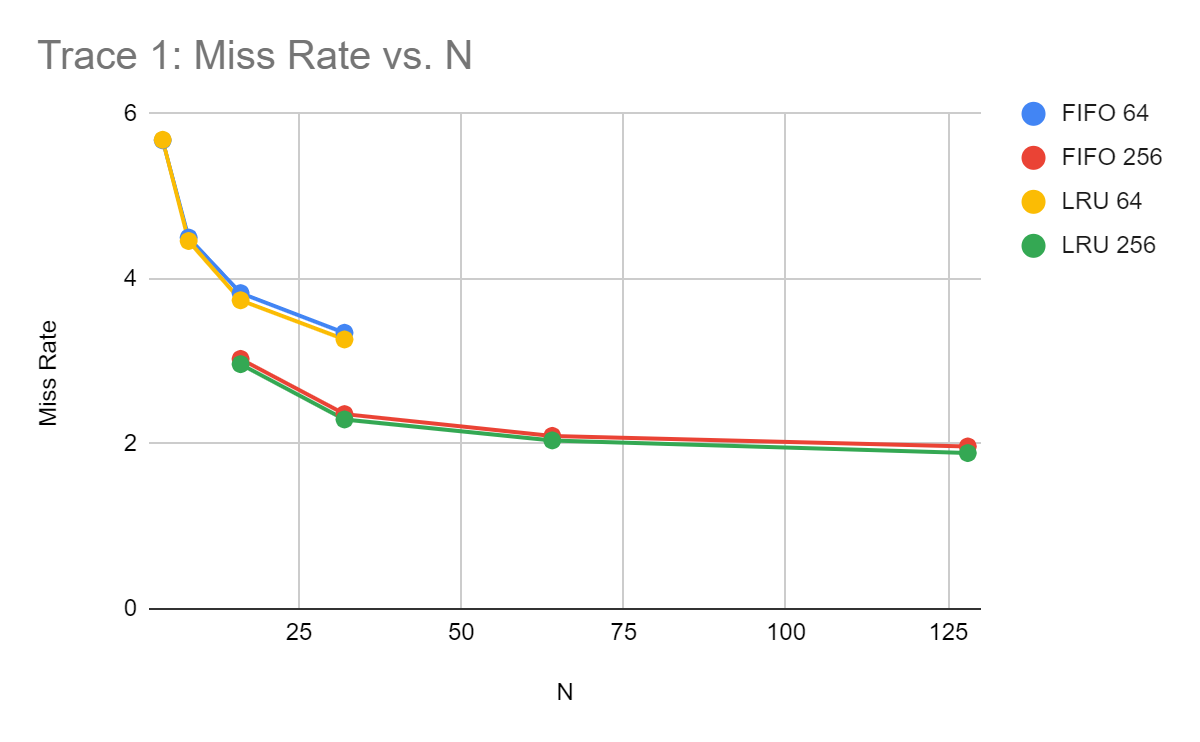
**

Figure 3: Miss Rate vs N for Trace 1

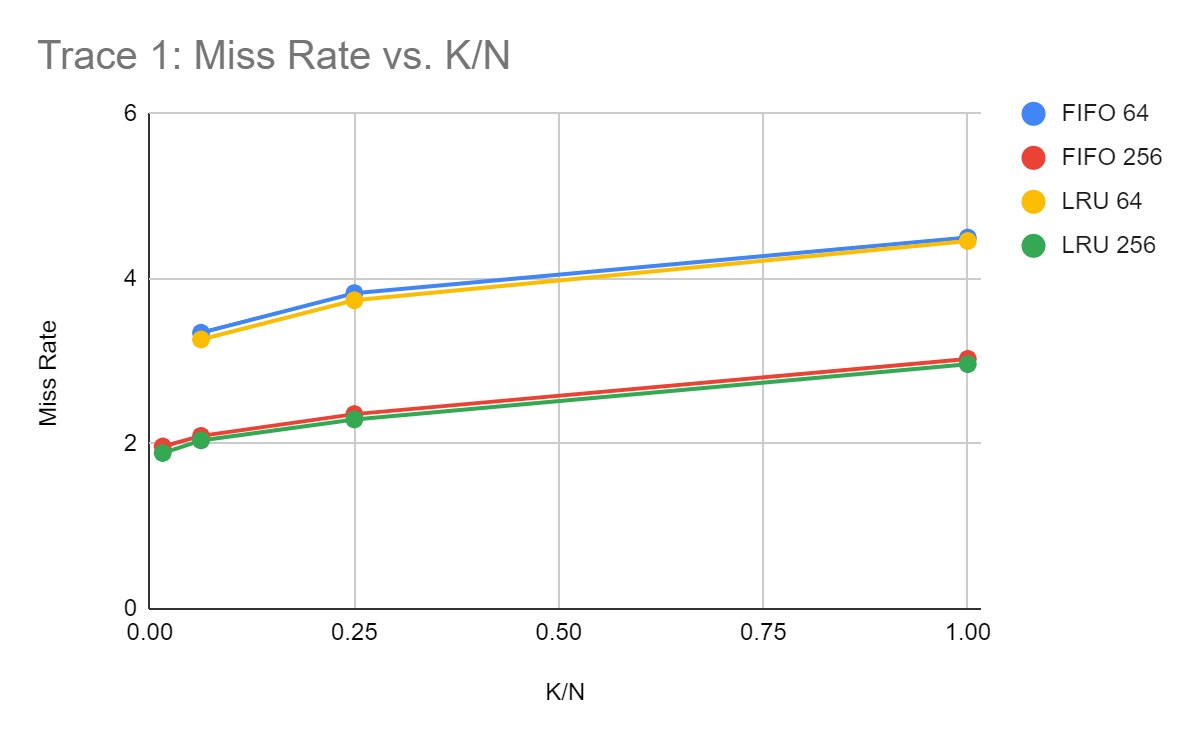
**

Figure 4: Miss Rate vs K/N for Trace 1

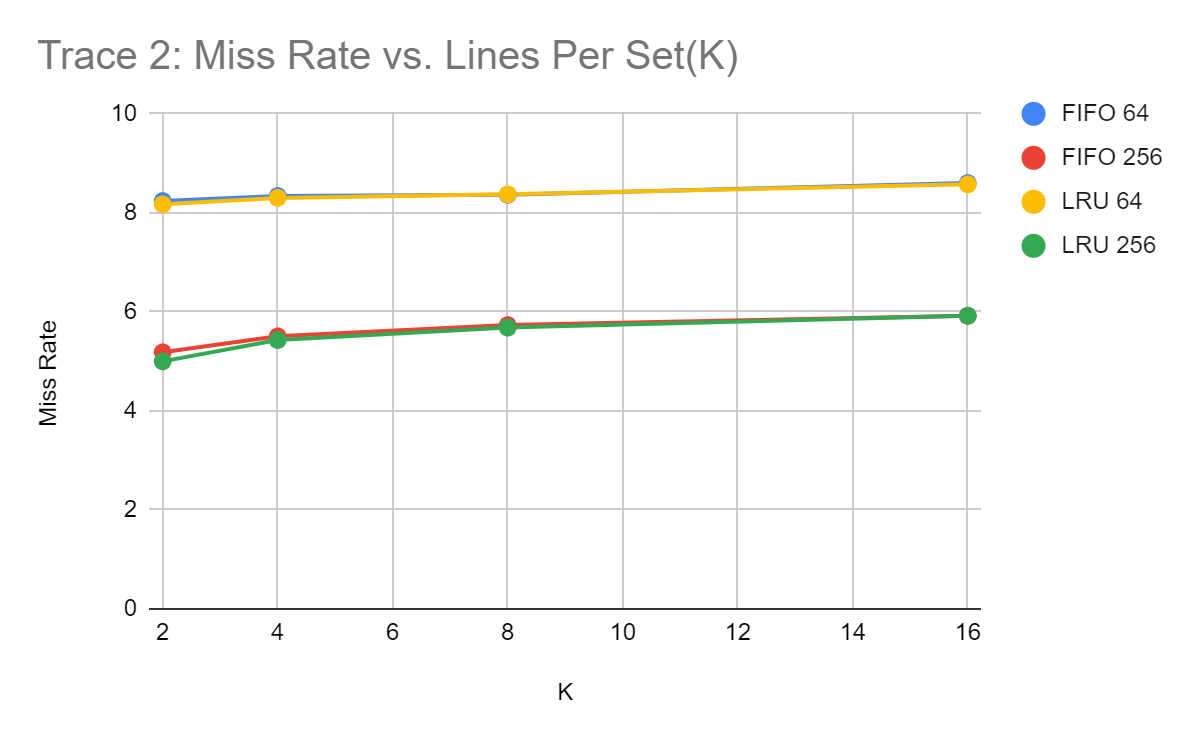
**

Figure 5: Miss Rate vs K for Trace 2

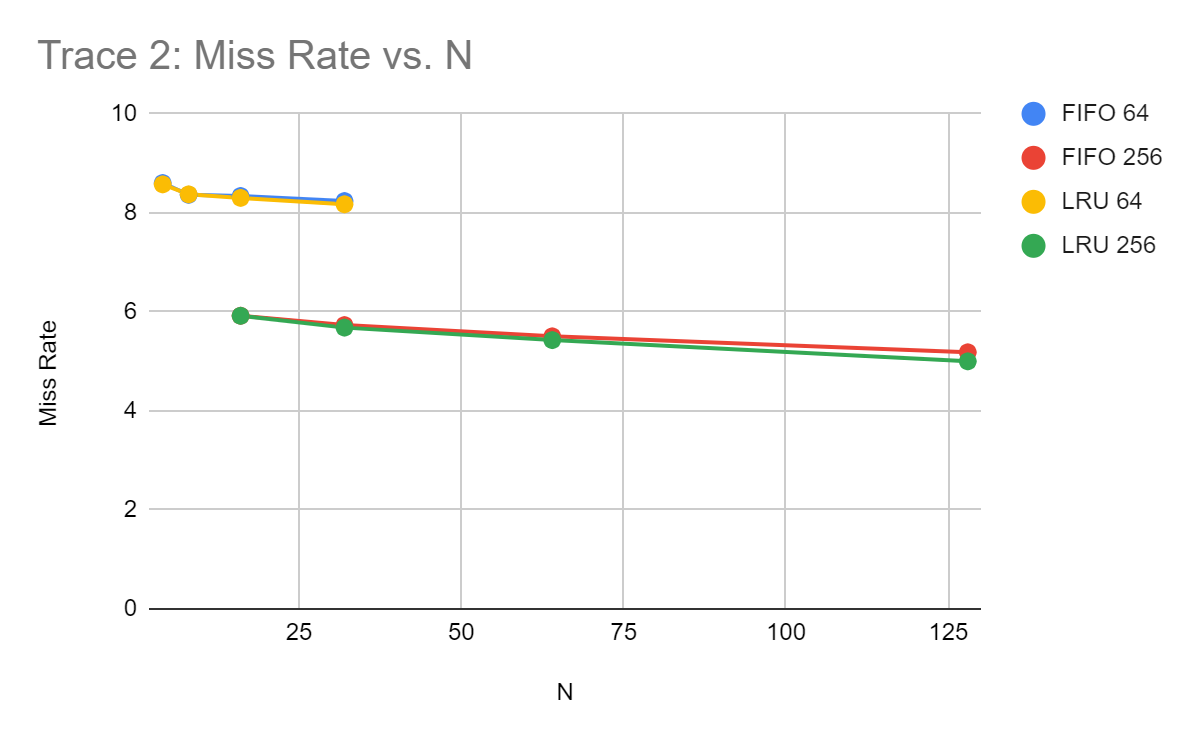
**

Figure 6: Miss Rate vs N for Trace 2

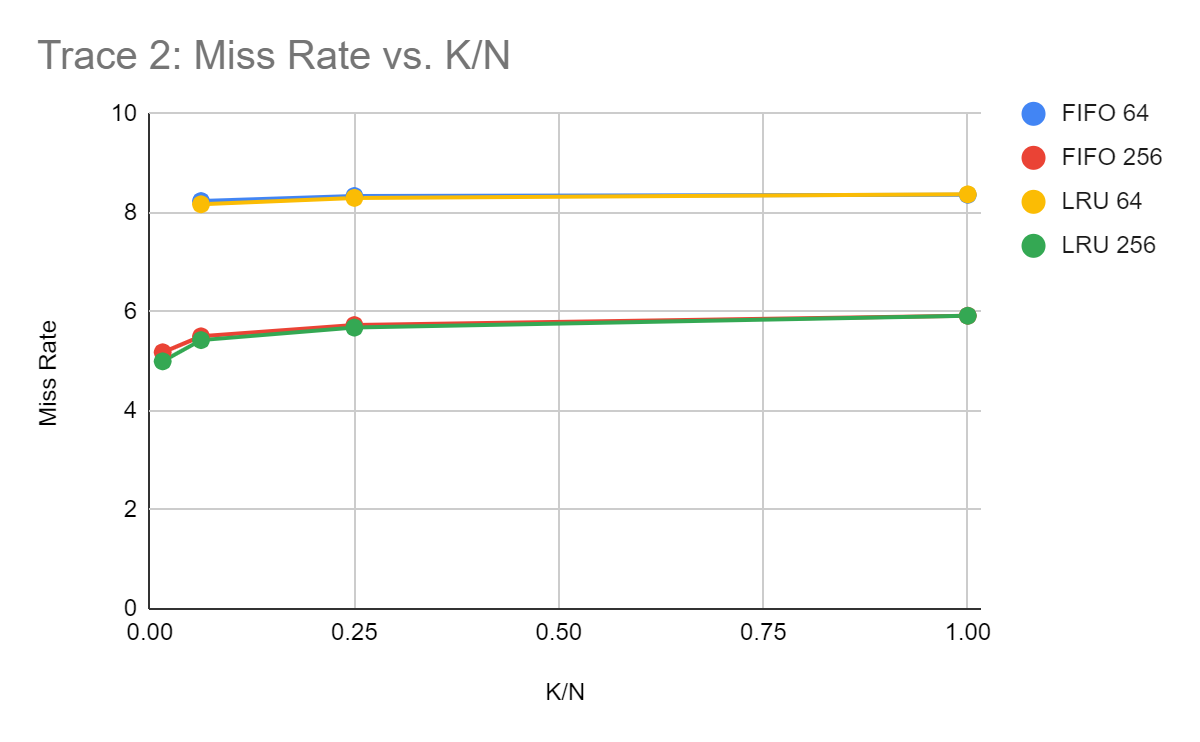
**

Figure 7: Miss Rate vs K/N for Trace 2

**Discussion:**

From the simulation results shown in *Table 1 & 2*, it can be seen that the miss rate for the LRU replacement policy is lower than that of FIFO. This is because LRU keeps track of recently used items so frequent inputs will have more hits because they are not being replaced as often. The FIFO method has better runtime because hits do not have to modify the cache. The graphs show us that, as KN increases, the miss rate decreases in a diminishing fashion. This is because large sets take advantage of spatial locality at the cost of performance. One of the early hurdles the group had to overcome was how the code would determine the replacement priority for LRU. Working it out on paper showed how the lower priorities could be left the same while higher ones would have to be decremented. Once that was figured out, it was possible to implement the cache in the LRU style.

**Conclusion:**

Simulating the cache systems yielded results that matched our expectations. Tests with larger sets gave lower miss rates. The LRU replacement policy, overall, had lower miss rates than the FIFO systems which was expected and desired. These results were consistent with varying K and N parameters. These findings, together, indicate that the cache was successfully implemented. This lab furthered the groups understanding of how cache systems work and their respective qualities.

**Appendix A - Source Code**

#Brian Worts and Chris Jenson

#ELC 463, Computer Engineering Lab II

#Lab 1, Cache Simulator

import math

#def doCache(K, sets, repo):

def doCache(K, KN, repo, dataSet, iteration):

numSets = int(KN/K)

cache = [ [ None for i in range(K) ] for j in range(numSets) ]

setHitIndex = 0

setToWrite = 0

misscount = 0

LRUSetReplacePointer = 0

LRUReplacementIndexList = []

refCount = 0

#For every word

for entry in range(0, len(dataSet)-3, 3):

word = [] # List containing all references

refCount += 1

reference = str(bin((dataSet[entry+2]<<16) | (dataSet[entry+1]<<8) | (dataSet[entry])))

reference = reference[2:].zfill(24)

word.append(reference)

#Seperate the reference into tag and index, can ignore offset

indexSize = (int)(math.log(K, 2))

tag = reference[0:(21-indexSize)] # slice out the offset

index = reference[(21-indexSize):(21)] # slice out the index

##############################

## LOOK FOR HIT IN CACHE

hit = False

for set in cache:

if (set[int(index, 2)] == tag):

hit = True

setHitIndex = cache.index(set)

##############################

#HIT

if (hit):

#If FIFO, do nothing

#If LRU, do stuff

if (repo == 'LRU'):

# get the replacement index of the hit

temp = LRUReplacementIndexList[setHitIndex]

# for all priorities greater than the hit, subtract them by 1

for index in range(len(LRUReplacementIndexList)):

if(LRUReplacementIndexList[index] > temp):

LRUReplacementIndexList[index] -= 1

# give the MRU the lowest replacement priority

LRUReplacementIndexList[setHitIndex] = len(LRUReplacementIndexList)-1

##############################

#MISS

else:

#increment misscount

misscount += 1

if(repo == 'FIFO'):

cache[setToWrite][int(index, 2)] = tag #Replace in settowrite at index

setToWrite += 1

if(setToWrite == numSets):

setToWrite = 0

elif (repo == 'LRU'):

# miss and LRUReplacementIndexList not full:

if(len(LRUReplacementIndexList) < numSets):

#update the priority list

LRUReplacementIndexList.append(LRUSetReplacePointer)

#add the word to the cache

cache[LRUSetReplacePointer][int(index, 2)] = tag #Replace in settowrite at index

LRUSetReplacePointer += 1

# cache is full and we have a miss

else:

#find the least recently uses (0) and replace word there

writeLocation = LRUReplacementIndexList.index(0)

cache[writeLocation][int(index, 2)] = tag

# decrement all priorities by 1

for index in range(len(LRUReplacementIndexList)):

LRUReplacementIndexList[index] -= 1

# make the priority of the MRU the highest value

LRUReplacementIndexList[writeLocation] = len(LRUReplacementIndexList)-1

else:

print('BAD RE-PO')

#Calculate the missrate and print the test results

missRate = (misscount/(refCount\*10))\*100

formatted\_rate = "{:.4f}".format(missRate)

print(str(iteration) + ': Replacement Policy = ' + repo + '; KN = ' + str(K\*numSets) + '; K = ' + str(K) + '; sets(N) = ' + str(numSets) + '; MISS RATE = ' + str(formatted\_rate) + "%" + '; MISS COUNT = ' + str(misscount))

# File Read In

f1 = open('TRACE1.DAT', 'rb') # Open the trace file

data1 = f1.read() # Read the file into date

f1.close()

f2 = open('TRACE2.DAT', 'rb') # Open the trace file

data2 = f2.read() # Read the file into date

f2.close()

# Parameters

L = 8 # Number of bytes per line of cache memory

K = [2, 4, 8, 16] # Number of lines per set

KN = [64, 256] # Number of sets

Repo = ['FIFO','LRU'] # Replacement Policy

sets = [4, 8, 16, 32] # Number of sets in the cache

way = 1 # Set as a 1-way associative cache

apl = 8 # Set as 8 contiguous addresses per line

memRef = 60000 # Number of 3-byte memory references

refs = 600000 # Total number of references

iteration = 1

#Cache tests on TRACE1

for z in Repo:

for y in KN:

for x in K:

doCache(x, y, z, data1, iteration)

iteration += 1

#Cache tests on TRACE2

for z in Repo:

for y in KN:

for x in K:

doCache(x, y, z, data2, iteration)

iteration += 1