Report

October 19, 2022

1 University of Central Florida

1.1 Department of Computer Science

1.2 CDA 5106: Spring 2020

1.3 Machine Problem 1: Cache Design, Memory Hierarchy Design by

1.4 « Sahar Sheikholeslami »

1.4.1 Honor Pledge: "I have neither given nor received unauthorized aid on this test or assignment."

1.4.2 Student's electronic signature: _____Sahar Sheikholeslami_____

(sign by typing your name)

[1]:
```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import math
```

2 1) GRAPH #1 (total number of simulations: 55)

2.0.1 For this experiment:

2.0.2 o L1 cache: SIZE is varied, ASSOC is varied, BLOCKSIZE = 32.

2.0.3 o L2 cache: None.

2.0.4 o Replacement policy: LRU

2.0.5 o Inclusion property: non-inclusive

2.0.6 Plot L1 miss rate on the y-axis versus log2(SIZE) on the x-axis, for eleven different cache sizes:

2.0.7 SIZE = 1KB, 2KB, ..., 1MB, in powers-of-two. (That is, log2(SIZE) = 10, 11, ..., 20.) The graph should contain five separate curves (i.e., lines connecting points), one for each of the following associativities: direct-mapped, 2-way set-associative, 4-way set-associative, 8-way setassociative, and fully- associative. All points for direct-mapped caches should be connected with a line, all points for 2- way set-associative caches should be connected with a line, etc.

[2]:
```
BockSize = 32
size1 = []
log1 = []
fullyAssoc1=[]
for i in range (11):
    sizei = (2**i)*1024
    size1.append(sizei)
    logi= math.log(sizei, 2)
    log1.append(logi)
    fullyassoci=sizei/BockSize
    fullyAssoc1.append(fullyassoci)


    print("size of cache ", i+1, " is " , sizei, " and the log is: ", logi)


print(size1)
print(log1)
print( fullyAssoc1)
```

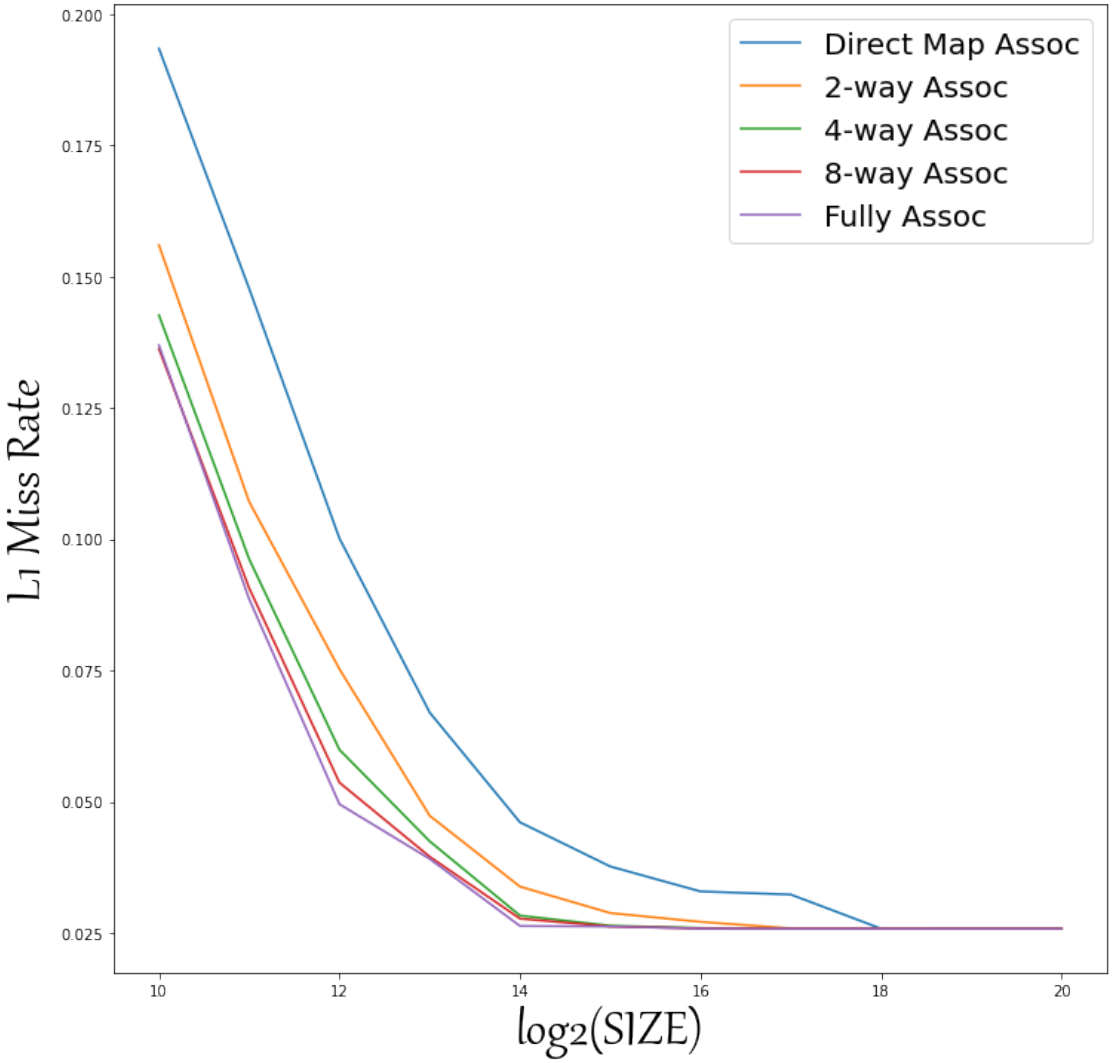
```
size of cache  1  is  1024  and the log is:  10.0
size of cache  2  is  2048  and the log is:  11.0
size of cache  3  is  4096  and the log is:  12.0
size of cache  4  is  8192  and the log is:  13.0
size of cache  5  is  16384  and the log is:  14.0
size of cache  6  is  32768  and the log is:  15.0
size of cache  7  is  65536  and the log is:  16.0
size of cache  8  is  131072  and the log is:  17.0
```

```
size of cache  9  is  262144  and the log is:  18.0
size of cache  10  is  524288  and the log is:  19.0
size of cache  11  is  1048576  and the log is:  20.0
[1024, 2048, 4096, 8192, 16384, 32768, 65536, 131072, 262144, 524288, 1048576]
[10.0, 11.0, 12.0, 13.0, 14.0, 15.0, 16.0, 17.0, 18.0, 19.0, 20.0]
[32.0, 64.0, 128.0, 256.0, 512.0, 1024.0, 2048.0, 4096.0, 8192.0, 16384.0,
32768.0]
```

2.0.8 L1 Miss Arrays for the 11 sizes and different Assoc

[3]:
```
L1MissRates_Assoc1 =[0.193460,0.147740,0.100170,0.067000,0.046090,0.037680,0.
↪032920, 0.032330, 0.025840, 0.025840, 0.025840]
L1MissRates_Assoc2 =[0.156030,0.107140, 0.075280, 0.047340, 0.033840, 0.028810,␣
↪0.027130, 0.025900, 0.025840, 0.025820, 0.025820]
L1MissRates_Assoc4 =[0.142700, 0.096220, 0.059920, 0.042470, 0.028320, 0.
↪026400, 0.025950, 0.025820, 0.025820, 0.025820, 0.025820]
L1MissRates_Assoc8 =[0.136270,0.090690 , 0.053650, 0.039540, 0.027740, 0.
↪026250,0.025890,0.025820, 0.025820, 0.025820, 0.025820]
L1MissRates_FullyAssoc = [0.136960, 0.088600, 0.049540, 0.039120, 0.026340, 0.
↪026240, 0.025830,0.025820, 0.025820, 0.025820, 0.025820]
```

[4]:
```
x = log1
y = [L1MissRates_Assoc1, L1MissRates_Assoc2, L1MissRates_Assoc4,␣
↪L1MissRates_Assoc8, L1MissRates_FullyAssoc ]
Label = ["Direct Map Assoc", "2-way Assoc", "4-way Assoc", "8-way Assoc",␣
↪"Fully Assoc"]

for i in range(len(y)):
    plt.rcParams['figure.figsize'] = [12, 12]
    #naming x axis
    plt.xlabel('log2(SIZE)', fontname="Gabriola", fontsize=40)
    # naming the y axis
    plt.ylabel('L1 Miss Rate', fontname="Gabriola", fontsize=40)
    plt.plot(x, y[i], label = Label[i], linestyle="-")

plt.legend(fontsize=20)
plt.show()
```


Direct Map Assoc
2-way Assoc
4-way Assoc
8-way Assoc
Fully Assoc
L1 Miss Rate
0.200
0.175
0.150
0.125
0.100
0.075
0.050
0.025
10
12
14
16
18
20
log2(SIZE)

2.1 1. Discuss trends in the graph. For a given associativity, how does increasing cache size affect miss rate? For a given cache size, what is the effect of increasing associativity?

2.1.1 As it can be seen in the graph each line represents a specific associativity tracing it to different cache sizes. Given that the block size, replacement and inclusion palicies are constant in all of them, this allows us to concentrate on effects of increased cache size with repspect to each associativity. As it can be seen increasing cache size, decreases the miss rate so they have inverse relationship with one another. This fundamentally makes sense, the more cache we have the more data we can store in it. Hence, the less the likely hood of miss rates. However adding cache size at some point does not help the miss rates as much as it can be seen in the figure aat some point miss rates do not change much by adding more cache and they turn into almost constant lines. which in turn is dependent on the number of Compulsory Misses

2.1.2 for a given cache size we would consider a vertical line through the graph. As it can be seen the higher the number of Associativity the lower the miss rates. So number of Associativity also has inverse correlation with cache size. That said the benefits of adding associativity at some point start having diminishing effects. It can be seen 2-way, 4-way, 8-way and full Associativity do almost merge at higher cache sizes and the 4-way, 8-way and fully Associate caches do follow each other very narrowly even at smaller cache sizes. So at some point adding more Associativity does not necessarily yield in much better miss rates

2.2 2. Estimate the compulsory miss rate from the graph.

2.2.1 All the graphs converge to 0.025820 so that would be the estimated compulsory miss rate. becuase that is the time the cache will minimally need to fetch the data originally needed from memory

2.3 3. For each associativity, estimate the conflict miss rate from the graph.

2.3.1 conflict miss = missrate - cumpulsory miss rate

[5]:
```
cumPulsoryMissRate = 0.025820
for i in range(len(y)):
    ConflictMissRate = y[i][0]-cumPulsoryMissRate
    print (Label[i], "Miss Rate = " , ConflictMissRate)
```

```
Direct Map Assoc Miss Rate =  0.16763999999999998
2-way Assoc Miss Rate =  0.13021
4-way Assoc Miss Rate =  0.11688
8-way Assoc Miss Rate =  0.11045
Fully Assoc Miss Rate =  0.11114
```

3 GRAPH #2 (no additional simulations with respect to GRAPH #1)

3.1 Same as GRAPH #1, but the y-axis should be AAT instead of L1 miss rate. Discussion to include in your report:

3.2 (Note: The value of Access time of 1KB 8-way Cache with block size 32 is not available in Cacti Table, you can ignore this point in your graph.)

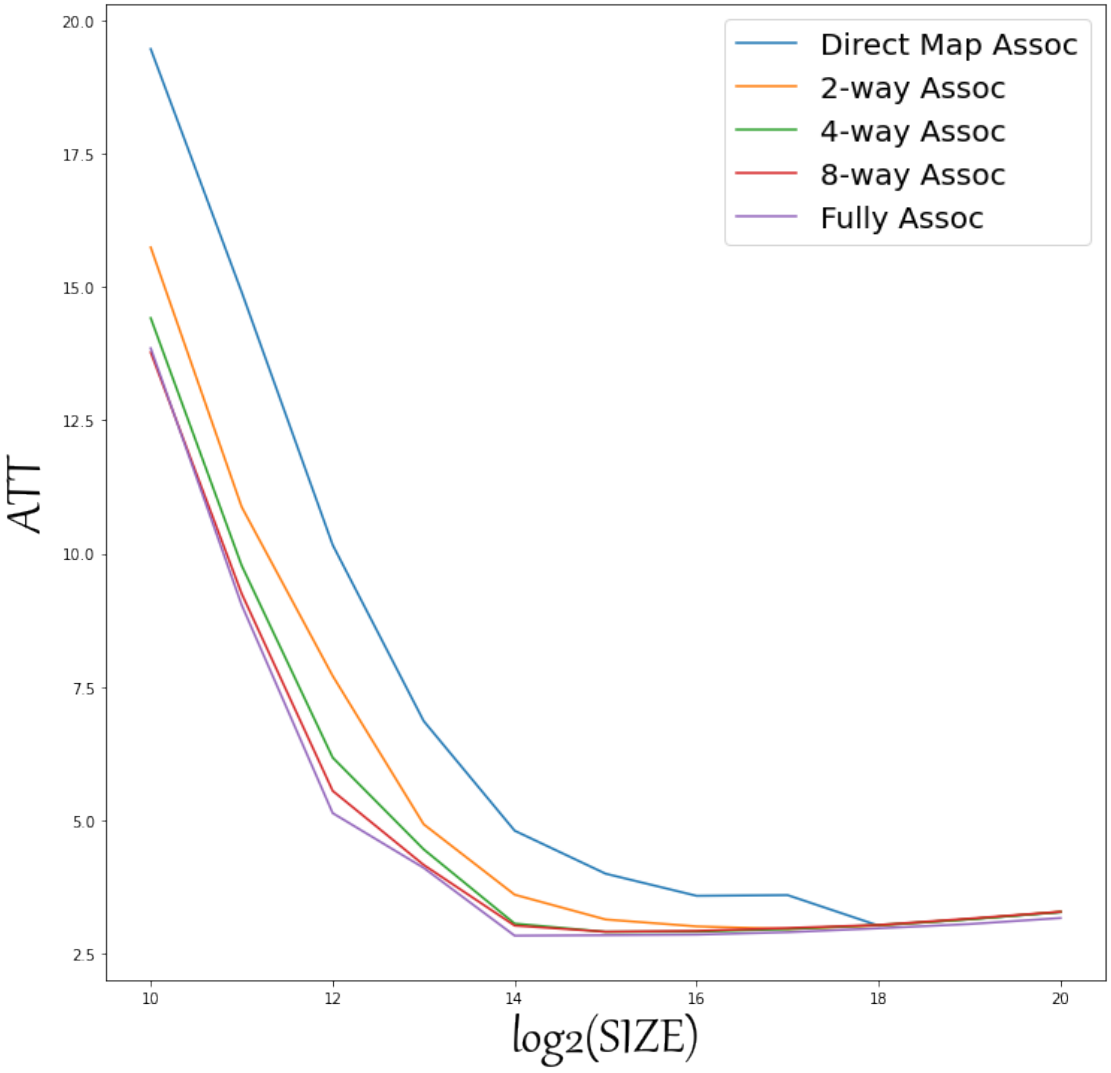
3.3 1. For a memory hierarchy with only an L1 cache and BLOCKSIZE = 32, which configuration yields the best (i.e., lowest) AAT?

3.3.1 AAT = 1 + 1 _ ty

[6]:
```
y = [L1MissRates_Assoc1, L1MissRates_Assoc2, L1MissRates_Assoc4,␣
 ↪L1MissRates_Assoc8, L1MissRates_FullyAssoc ]
MissPenalty = 100

## from excell
AccessTime1KB = [0.114797, 0.140329,0.14682, 0.14682,  0.155484]
AccessTime2KB = [0.12909, 0.161691, 0.154496, 0.180686, 0.176515]
AccessTime4KB = [0.147005, 0.181131, 0.185685, 0.189065, 0.182948]
AccessTime8KB = [0.16383, 0.194195, 0.211173, 0.212911, 0.198581]
AccessTime16KB = [0.198417,0.223917,0.233936, 0.254354, 0.205608]
AccessTime32KB = [0.233353, 0.262446, 0.27125, 0.288511, 0.22474]
AccessTime64KB = [0.294627, 0.300727, 0.319481, 0.341213, 0.276281]
AccessTime128KB = [0.3668, 0.374603, 0.38028, 0.401236, 0.322486]
AccessTime256KB = [0.443812, 0.445929, 0.457685, 0.458925, 0.396009]
AccessTime512KB = [0.563451,0.567744,0.564418,0.578177,0.475728]
AccessTime1024KB = [0.69938,0.706046,0.699607,0.705819,0.588474]

AccessTimes = [AccessTime1KB ,
                AccessTime2KB ,
                AccessTime4KB ,
                AccessTime8KB ,
                AccessTime16KB,
                AccessTime32KB,
                AccessTime64KB,
                AccessTime128KB,
                AccessTime256KB,
                AccessTime512KB,
                AccessTime1024KB,
              ]
ATT = []

for i in range (5):
    A =[]
    for j in range(11):
```

```
        x = AccessTimes[j][i] + y[i][j]*MissPenalty
        A.append(x)
    ATT.append(A)
```

[7]:
```
x = log1
y = ATT
Label = ["Direct Map Assoc", "2-way Assoc", "4-way Assoc", "8-way Assoc",␣
 ↪"Fully Assoc"]

for i in range(len(y)):
    plt.rcParams['figure.figsize'] = [12, 12]
    plt.xlabel('log2(SIZE)', fontname="Gabriola", fontsize=40)
    # naming the y axis
    plt.ylabel('ATT', fontname="Gabriola", fontsize=40)
    plt.plot(x, y[i], label = Label[i], linestyle="-")

plt.legend(fontsize=20)
plt.show()
```


Direct Map Assoc
2-way Assoc
4-way Assoc
8-way Assoc
Fully Assoc
ATT
log2(SIZE)
20.0
17.5
15.0
12.5
10.0
7.5
5.0
2.5
10
12
14
16
18
20

3.3.2 Fully Assocuative yilding the best at cache size log(size)= 14 of of 16384 bytes. As it can be seen the larger the size of the cache, the less ATT. At some point adding cache size starts having diminishing effects due to the fact that we are keeping unnecessary information in the cache. Sort of polutting the cache.

4 8.2. Replacement policy study

4.1 GRAPH #3 (total number of simulations: 27)

4.2 For this experiment:

4.3 o L1 cache: SIZE is varied, ASSOC = 4, BLOCKSIZE = 32.

4.4 o L2 cache: None.

4.5 o Replacement policy: varied

4.6 o Inclusion property: non-inclusive

4.7 Plot AAT on the y-axis versus log2(SIZE) on the x-axis, for nine different cache sizes: SIZE = 1KB, 2KB, ... , 256KB, in powers-of-two. (That is, log2(SIZE) = 10, 11, ..., 18.) The graph should contain three separate curves (i.e., lines connecting points), one for each of the following replacement policies: LRU, Pseudo-LRU, Optimal. All points for LRU replacement policy should be connected with a line, all points for Pseudo-LRU replacement policy should be connected with a line, etc.

4.8 Discussion to include in your report:

4.9 1. Discuss trends in the graph. Which replacement policy yields the best (i.e., lowest) AAT?

[8]:
```
BlockSize = 32
size2 = []
Logs2=[]
for i in range (9):
    sizei = (2**i)*1024
    size2.append(sizei)
    log2 = math.log(sizei, 2)
    Logs2.append(log2)
    print("cashe ", i+1, "size of cashe in byes: ", sizei, " and the log is",␣
 ↪log2 )

print(Logs2)
print(size2)
```

```
cashe  1 size of cashe in byes:  1024  and the log is 10.0
cashe  2 size of cashe in byes:  2048  and the log is 11.0
cashe  3 size of cashe in byes:  4096  and the log is 12.0
cashe  4 size of cashe in byes:  8192  and the log is 13.0
cashe  5 size of cashe in byes:  16384  and the log is 14.0
cashe  6 size of cashe in byes:  32768  and the log is 15.0
```

```
cashe  7 size of cashe in byes:  65536  and the log is 16.0
cashe  8 size of cashe in byes:  131072  and the log is 17.0
cashe  9 size of cashe in byes:  262144  and the log is 18.0
[10.0, 11.0, 12.0, 13.0, 14.0, 15.0, 16.0, 17.0, 18.0]
[1024, 2048, 4096, 8192, 16384, 32768, 65536, 131072, 262144]
```

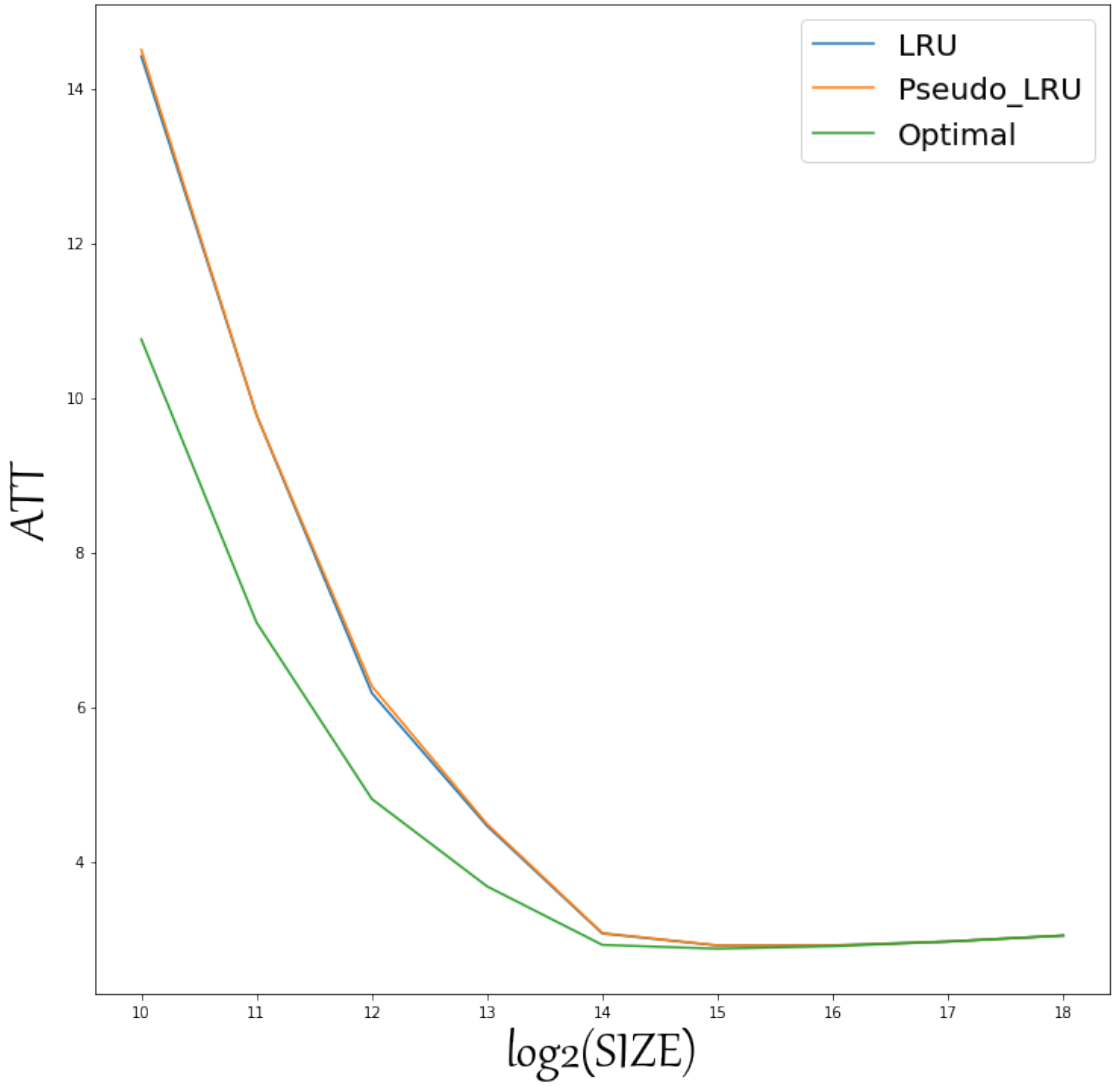
[9]:
```
L1MissRates_LRU =  [0.142700, 0.096220, 0.059920, 0.042470, 0.028320, 0.026400,␣
↪0.025950, 0.025820, 0.025820]
L1MissRates_Pseudo_LRU = [0.143550, 0.096190, 0.060820, 0.042710, 0.028390, 0.
↪026410, 0.025910, 0.025820, 0.025820]
L1MissRates_Optimal = [0.106090, 0.069360, 0.046220, 0.034650, 0.026850, 0.
↪025980, 0.025820, 0.025820, 0.025820]
```

[10]:
```
MissPenalty = 100
AccessTime1KB4 = [0.114797, 0.140329, 0.14682, 0.14682 ,0.155484]
AccessTime2KB4 = [0.12909, 0.161691, 0.154496, 0.180686, 0.176515]
AccessTime4KB4 = [0.147005, 0.181131, 0.185685, 0.189065, 0.182948]
AccessTime8KB4 = [0.16383, 0.194195, 0.211173, 0.212911, 0.198581]
AccessTime16KB4 = [0.198417,0.223917,0.233936, 0.254354, 0.205608]
AccessTime32KB4 = [0.233353, 0.262446, 0.27125, 0.288511, 0.22474]
AccessTime64KB4 = [0.294627, 0.300727, 0.319481, 0.341213, 0.276281]
AccessTime128KB4 = [0.3668, 0.374603, 0.38028, 0.401236, 0.322486]
AccessTime256KB4 = [0.443812, 0.445929, 0.457685, 0.458925, 0.396009]

AccessTimes2 = [AccessTime1KB ,
                AccessTime2KB ,
                AccessTime4KB ,
                AccessTime8KB ,
                AccessTime16KB,
                AccessTime32KB,
                AccessTime64KB,
                AccessTime128KB,
                AccessTime256KB,

              ]

y2 = [L1MissRates_LRU, L1MissRates_Pseudo_LRU,L1MissRates_Optimal]

ATT2 = []

for i in range (3):
    A2 =[]
    for j in range(9):
        x = AccessTimes2[j][2] + y2[i][j]*MissPenalty
        A2.append(x)
    ATT2.append(A2)
```

```
print (ATT2)
```

```
[[14.41682, 9.776496, 6.177685, 4.4581729999999995, 3.065936, 2.91125,
2.9144810000000003, 2.96228, 3.039685], [14.50182, 9.773496, 6.267685,
4.4821729999999995, 3.072936, 2.9122500000000002, 2.910481, 2.96228, 3.039685],
[10.75582, 7.090496000000001, 4.807685, 3.676173, 2.918936, 2.86925, 2.901481,
2.96228, 3.039685]]
```

[11]:
```
x = Logs2
y = ATT2
Label = ["LRU", "Pseudo_LRU", "Optimal"]

for i in range(len(y)):
    plt.rcParams['figure.figsize'] = [12, 12]
    plt.xlabel('log2(SIZE)', fontname="Gabriola", fontsize=40)
    # naming the y axis
    plt.ylabel('ATT', fontname="Gabriola", fontsize=40)
    plt.plot(x, y[i], label = Label[i], linestyle="-")

plt.legend(fontsize=20)
plt.show()
```


LRU
Pseudo_LRU
Optimal
ATT
14
12
10
8
6
4
10
11
12
13
14
15
16
17
18
log2(SIZE)

4.9.1 As it can be seen the larger the size of the cache, the less ATT . Among the policies, optimal policy gives us the best ATT. Which is understandeable becuase in theory that is the best yeild in terms of cache performance. Optimal Policy, Cache size of log (size) = 14 = 16384 Bytes. As it can be seen the larger the size of the cache, the less ATT. At some point adding cache size starts having diminishing effects due to the fact that we are keeping unnecessary information in the cache. Sort of polutting the cache.

4.10 8.3. Inclusion property study

4.10.1 GRAPH #4 (total number of simulations: 12)

4.10.2 For this experiment:

4.10.3 o L1 cache: SIZE = 1KB, ASSOC = 4, BLOCKSIZE = 32.

4.10.4 o L2 cache: SIZE = 2KB – 64KB, ASSOC = 8, BLOCKSIZE = 32.

4.10.5 o Replacement policy: LRU

4.10.6 o Inclusion property: varied

4.10.7 Plot AAT on the y-axis versus log2(L2 SIZE) on the x-axis, for six different L2 cache sizes: L2 SIZE = 2KB, 4KB, ... , 64KB, in powers-of-two. (That is, log2(L2 SIZE) = 11, 12, ..., 16.) The graph should contain two separate curves (i.e., lines connecting points), one for each of the following inclusion properties: non-inclusive and inclusive. All points for non-inclusive cache should be connected with a line, all points for inclusive cache should be connected with a line.

4.10.8 Discussion to include in your report:

4.10.9 1. Discuss trends in the graph. Which inclusion property yields a better (i.e., lower) AAT?

[12]:
```
BlockSize = 32
size3 = []
Logs3=[]
for i in range (6):
    sizei3 = (2**(i+1)*1024)
    size3.append(sizei3)
    log3 = math.log(sizei3, 2)
    Logs3.append(log3)
    print("cashe ", i+1, "size of cashe in byes: ", sizei3, " and the log is",␣
 ↪log3 )

print(Logs3)
```

```
cashe  1 size of cashe in byes:  2048  and the log is 11.0
cashe  2 size of cashe in byes:  4096  and the log is 12.0
cashe  3 size of cashe in byes:  8192  and the log is 13.0
cashe  4 size of cashe in byes:  16384  and the log is 14.0
cashe  5 size of cashe in byes:  32768  and the log is 15.0
```

```
cashe  6 size of cashe in byes:  65536  and the log is 16.0
[11.0, 12.0, 13.0, 14.0, 15.0, 16.0]
```

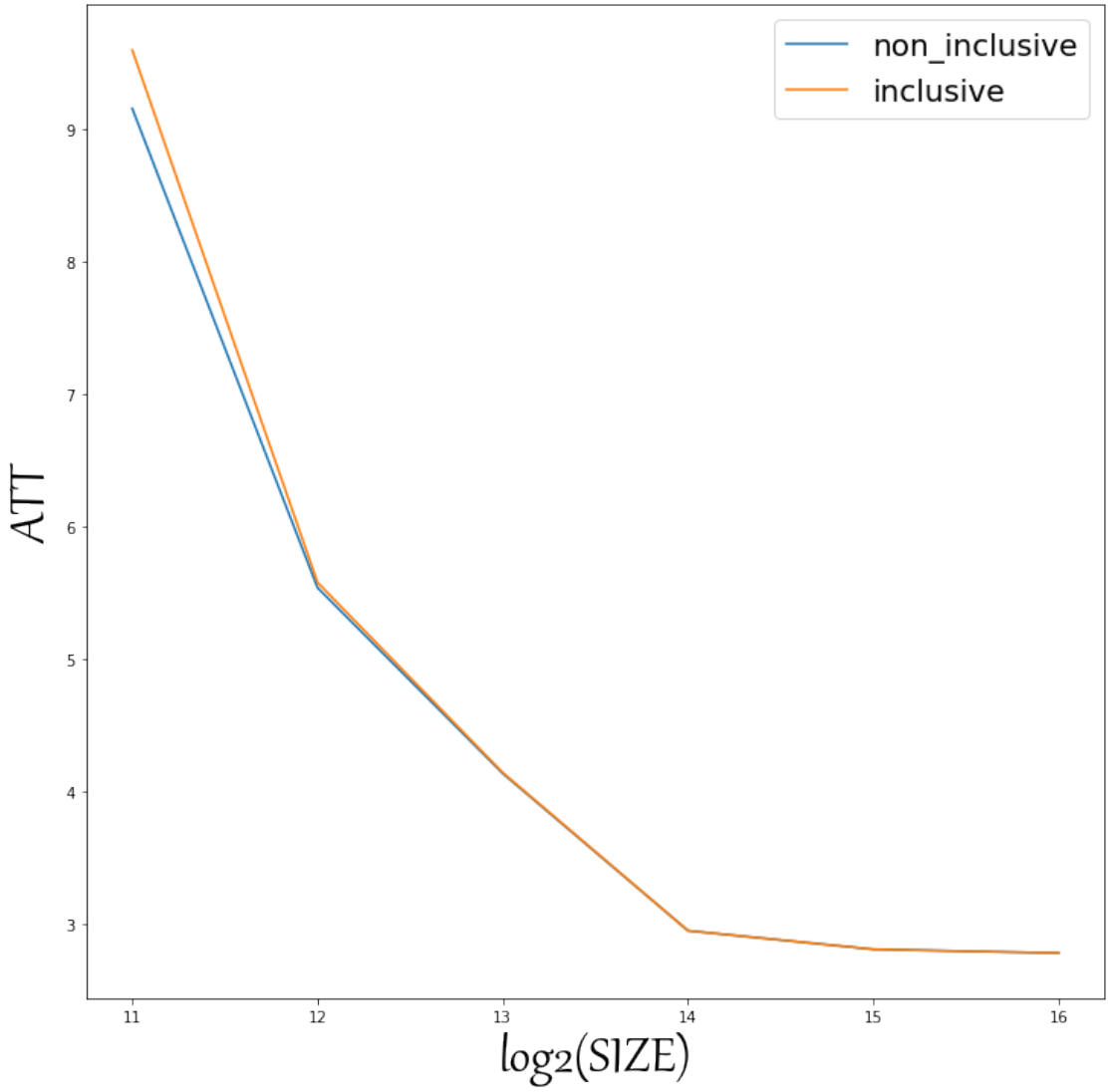
[13]:
```
L1MissRates_non_inclusive = [0.142700, 0.142700, 0.142700, 0.142700, 0.142700,␣
↪0.142700]
L2MissRates_non_inclusive = [0.629713, 0.376174 , 0.277715, 0.194114, 0.183952,␣
↪0.181430]

L1MissRates_inclusive = [0.145230, 0.143090, 0.142770, 0.142700, 0.142700, 0.
↪142700]
L2MissRates_inclusive = [0.649108, 0.377944, 0.278000, 0.194114, 0.183952, 0.
↪181430]
```

5 ATT = 1 + 1 (2 + 2 _)

[18]:
```
AccessTimes3 = [
                AccessTime2KB ,
                AccessTime4KB ,
                AccessTime8KB ,
                AccessTime16KB,
                AccessTime32KB,
                AccessTime64KB,
              ]

AccessTime1KB

y3 = [L2MissRates_non_inclusive, L2MissRates_inclusive]
z3= [L1MissRates_non_inclusive, L1MissRates_inclusive ]


ATT3 = []
for i in range (2):
    A3 =[]
    for j in range(6):
        x = AccessTime1KB[2]+ (z3[i][j] * (AccessTimes3[j][3]+␣
↪(y3[i][j]*MissPenalty)))
        A3.append(x)
    ATT3.append(A3)

print(ATT3)
```

```
[[9.1586084022, 5.5418025555, 4.140195449699999, 2.9531230957999997,
2.8129855596999995, 2.7845171951], [9.600056511779998, 5.58187400685,
4.146223303470001, 2.9531230957999997, 2.8129855596999995, 2.7845171951]]
```

[19]:
```
x = Logs3
y = ATT3
Label = ["non_inclusive", "inclusive"]

for i in range(len(y)):
    plt.rcParams['figure.figsize'] = [12, 12]
    plt.xlabel('log2(SIZE)', fontname="Gabriola", fontsize=40)
    # naming the y axis
    plt.ylabel('ATT', fontname="Gabriola", fontsize=40)
    plt.plot(x, y[i], label = Label[i], linestyle="-")


plt.legend(fontsize=20)
# plt.legend()
plt.show()
```



[20]:
```
ATT3[0] == ATT3[1]
```

[20]:
```
False
```

5.1 lines seem very close but they are not the same as it can be seen above with comparing the np.arrays. Non inclusive has slightly better ATT. The larger the cache gets the ATT decreases. At some point again adding cache size just has diminishing effects as it can be seen ATT after log(size) = 14 starts flattening to almost a line and after log(size) = 15 its almost horizantal line.