David Mendez
Computer Organization
12/04/21

Cache Simulation

To explore the effects of cache size, block size, associativity, and replacement policies on both the hit rate of a cache and its speed, a simulation of different cache types was made in a Python program.

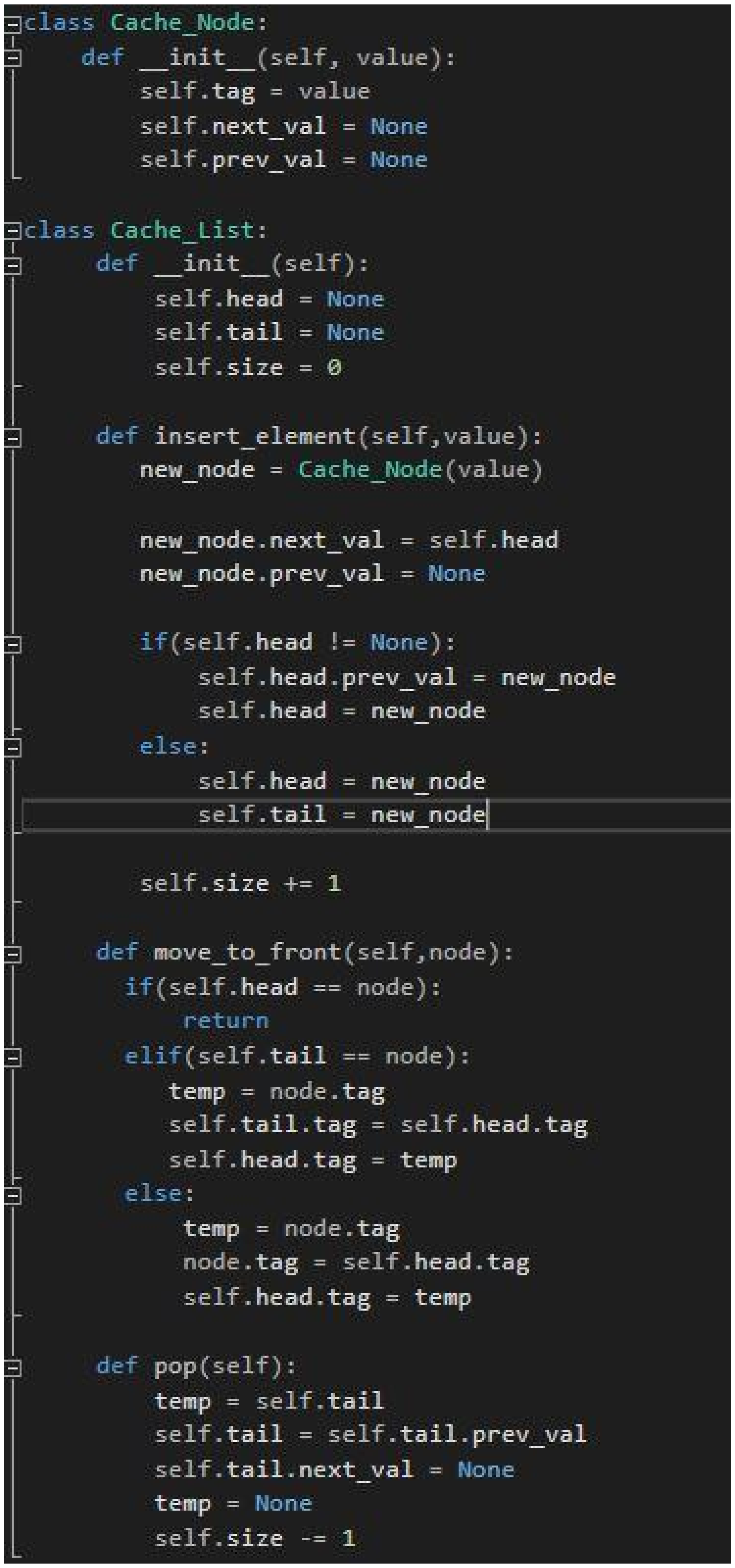
1) Fully Associative

a) FIFO

To model a fully associative cache, a doubly linked list was used. The general structure of the linked list is shown in Figure 1. The list inserts elements from the front of the list, removes elements from the back of the list, and can move any element to the front of the list. This was chosen due to the O(1) insertion and removal time complexity from the head and tail.

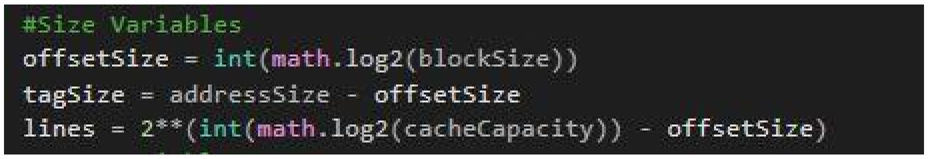
The parameters of the function were the memory calls from the trace file, Log_2(main memory size), cache size in bytes, and block size in bytes. When the function runs, the offset size, tag size, and line count has to be determined. The offset size is Log_2(blockSize), the tag size is Log_2(main memory size) - offset size. The number of lines is $2^{Log_2(cacheSize) - offsetSize}$. This is done and is shown in Figure 2.

Since a fully associative cache is one set of N blocks, the linked list acts as the set and each node holds a block. At each memory call, the 32 bit memory address is converted into a binary number. The bits from the range 0 to the tag size are the tag. The linked list is looped through until a matching tag is found. This is considered a hit. If no matching tag is found then this is considered a miss. If the linked list has less elements than the line count then the “cache” has space and so the tag is simply inserted into the linked list. If the linked list size matches the line number then the “cache” is full and the first in first out (FIFO) replacement policy has to be followed. The linked list acts as a queue since insertion occurs at the head and the first element placed in the “cache” is at the end of the linked list. Therefore, the last element of the list is removed and the new tag is placed into the list. This is shown in Figure 3.



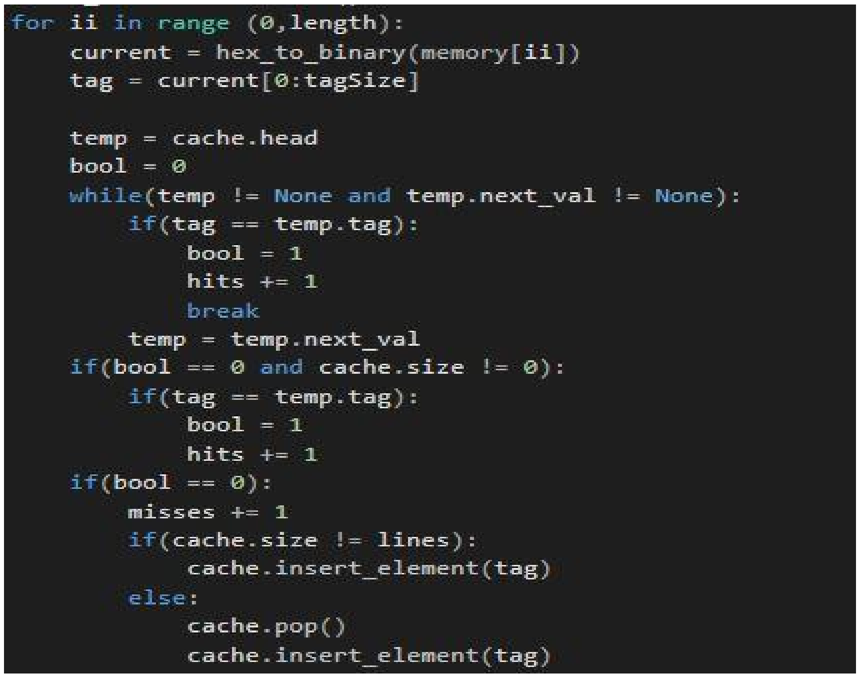
```
class Cache_Node:
    def __init__(self, value):
        self.tag = value
        self.next_val = None
        self.prev_val = None

class Cache_List:
     def __init__(self):
         self.head = None
         self.tail = None
         self.size = 0

     def insert_element(self,value):
        new_node = Cache_Node(value)

        new_node.next_val = self.head
        new_node.prev_val = None

        if(self.head != None):
            self.head.prev_val = new_node
            self.head = new_node
        else:
            self.head = new_node
            self.tail = new_node

        self.size += 1

     def move_to_front(self,node):
       if(self.head == node):
           return
       elif(self.tail == node):
          temp = node.tag
          self.tail.tag = self.head.tag
          self.head.tag = temp
       else:
           temp = node.tag
           node.tag = self.head.tag
           self.head.tag = temp

     def pop(self):
         temp = self.tail
         self.tail = self.tail.prev_val
         self.tail.next_val = None
         temp = None
         self.size -= 1
```

Figure 1 (Linked List Implementation)



```
#Size Variables
offsetSize = int(math.log2(blockSize))
tagSize = addressSize - offsetSize
lines = 2**(int(math.log2(cacheCapacity)) - offsetSize)
```

Figure 2 (Initializing offset size, tag size, and line number)

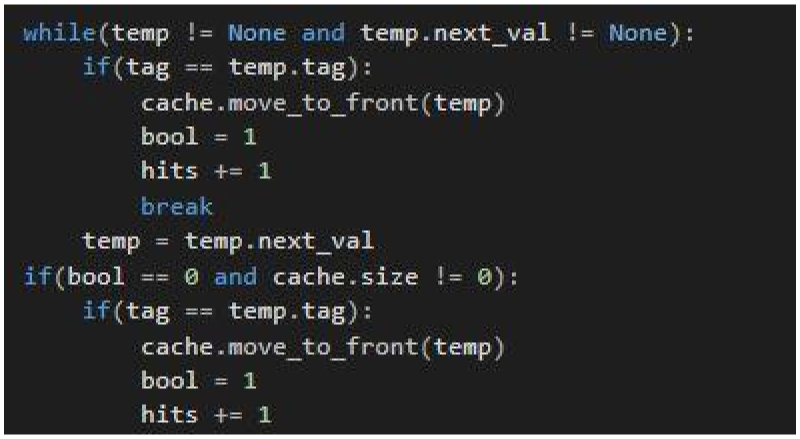


```
for ii in range (0,length):
    current = hex_to_binary(memory[ii])
    tag = current[0:tagSize]

    temp = cache.head
    bool = 0
    while(temp != None and temp.next_val != None):
        if(tag == temp.tag):
            bool = 1
            hits += 1
            break
        temp = temp.next_val
    if(bool == 0 and cache.size != 0):
        if(tag == temp.tag):
            bool = 1
            hits += 1
    if(bool == 0):
        misses += 1
        if(cache.size != lines):
            cache.insert_element(tag)
        else:
            cache.pop()
            cache.insert_element(tag)
```

Figure 3 (Fully Associative FIFO Implementation)

b) LRU

For the least recently used (LRU) replacement policy, the implementation is very similar. The only difference is that when there is a hit, that element is moved to the front of the list. This means the element at the end of the list is the element which was least recently used and this is what is removed. This ensures frequently used elements are kept in the “cache”. This is shown in Figure 4.



```python
while(temp != None and temp.next_val != None):
    if(tag == temp.tag):
        cache.move_to_front(temp)
        bool = 1
        hits += 1
        break
    temp = temp.next_val
if(bool == 0 and cache.size != 0):
    if(tag == temp.tag):
        cache.move_to_front(temp)
        bool = 1
        hits += 1
```

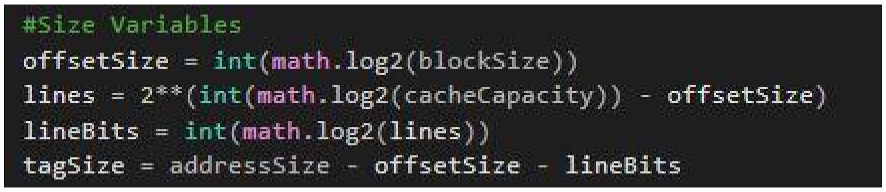
Figure 4 (Fully Associative LRU Implementation)

2) Directly Mapped

For the directly mapped case, the data structure used is changed. Since there are N sets of one block, each set can be treated as a key and the block as a value. Therefore a dictionary was used (the Python equivalent of an unordered map) which has O(1) lookup, insertion, and deletion time.

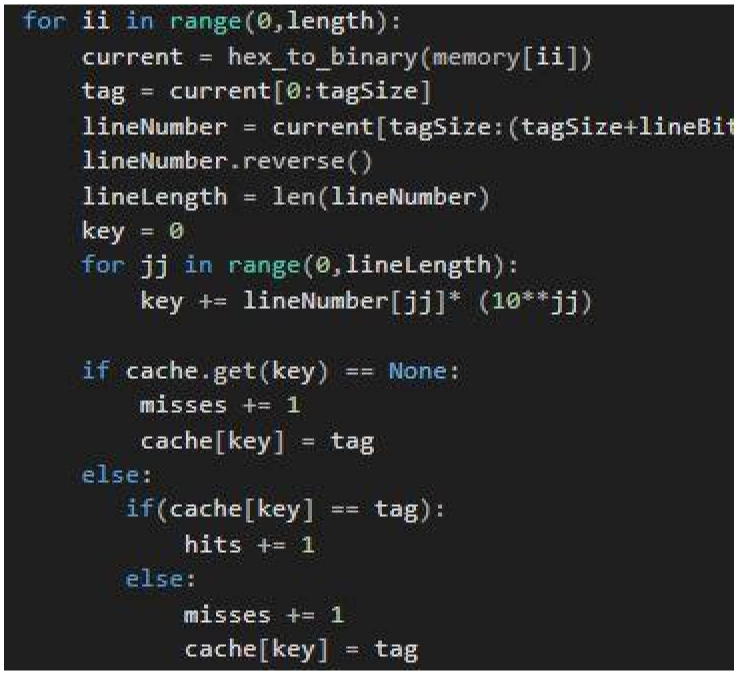
The same parameters were used as for the fully associative function. The offset size and line count are determined in the same way as the fully associative case. In this case we also have to determine the bits in the line. This is accomplished by Log_2(lineCount). Then to find the tag size, it is Log_2(addressSize) - offsetSize - bitsInLine. This is shown in Figure 5.

The implementation of the direct mapped cache is that when a 32-bit memory address is read, it is converted to binary. The tag is collected from the bit 0 to the tag size bit. From the tag size bit to the (tag size bit + bits in line) bit, are the line bits. The line bits act as the key to the map. If there is no value at this key then there is space in the “cache” for the tag so the tag is inserted into the map at this key. If the tag at that key matches the new tag then this is a hit. If the tag at that key does not match the new key then this is a miss. The old tag is removed and replaced with the new tag. This is shown in Figure 6.



```python
#Size Variables
offsetSize = int(math.log2(blockSize))
lines = 2**(int(math.log2(cacheCapacity)) - offsetSize)
lineBits = int(math.log2(lines))
tagSize = addressSize - offsetSize - lineBits
```

Figure 5 (Bits per Line and Tag Size Calculation for Directly Mapped)



```
for ii in range(0,length):
    current = hex_to_binary(memory[ii])
    tag = current[0:tagSize]
    lineNumber = current[tagSize:(tagSize+lineBit
    lineNumber.reverse()
    lineLength = len(lineNumber)
    key = 0
    for jj in range(0,lineLength):
        key += lineNumber[jj]* (10**jj)

    if cache.get(key) == None:
        misses += 1
        cache[key] = tag
    else:
       if(cache[key] == tag):
           hits += 1
       else:
           misses += 1
           cache[key] = tag
```

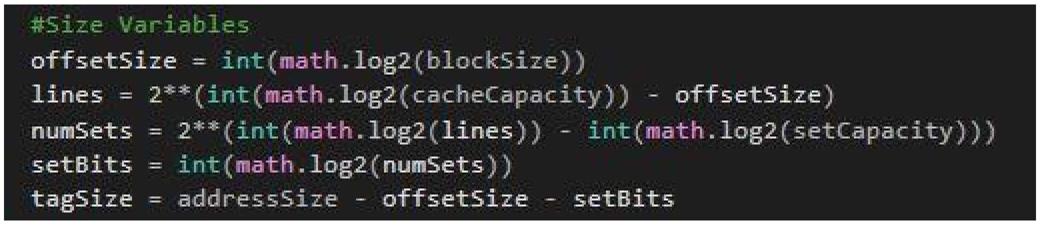
Figure 6 (Direct Mapped Implementation)

3) **Set Associative**

The implementation of set associative is a mix between fully associative and directly mapped. Set associative is N sets with M blocks. This represents an M associative set.

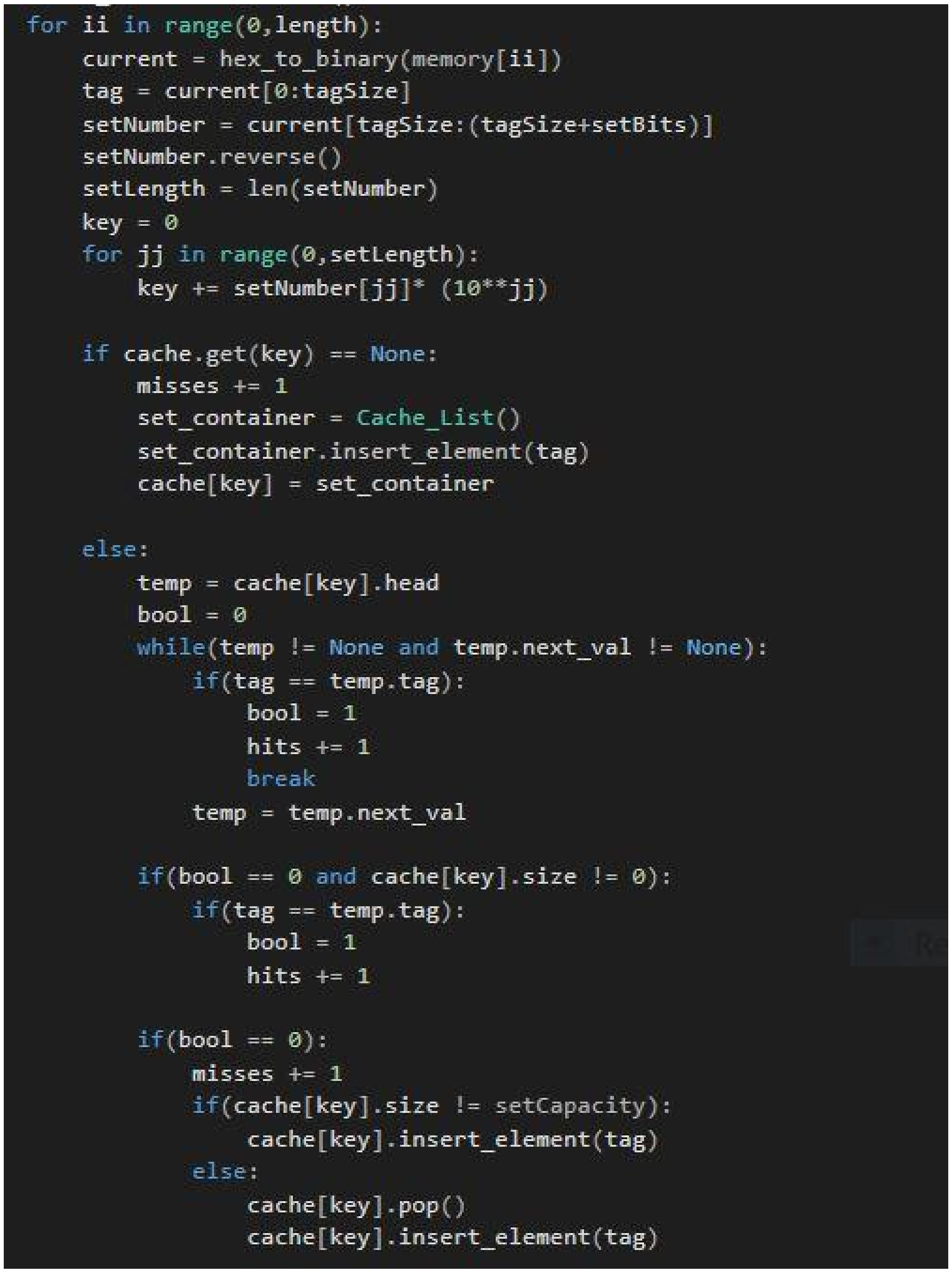
Some information must be found before the simulation runs. This includes the number of bits in the set and the tag size. The number of sets is $2^{Log_2(line\ count) - Log_2(set\ associativity)}$. The bits in the set are $Log_2(setCount)$. Finally, the tagSize = addressSize - offsetSize - bitsInSet. This is shown in Figure 7.

The sets are modeled by a dictionary. The set bits act as the key and map to a linked list of blocks. Each set maintains its own replacement policy, either FIFO or LRU, which have the same implementation as the fully associative FIFO and LRU replacement policies. When a 32-bit memory address is read, it is converted to binary. The tag is collected from the bit 0 to the tag size bit. From the tag size bit to the (tag size bit + bits in set) bit, are the set bits. The set bits act as a key to a set of size M. The set is then searched for the tag. If the tag is found it is a hit. If it is not found then it is a miss. If the set has less than M elements, then the tag is inserted into the set. If the set already has M elements then a replacement policy replaces one of the tags already in the set with the new tag. These are shown in Figure 8 and 9.



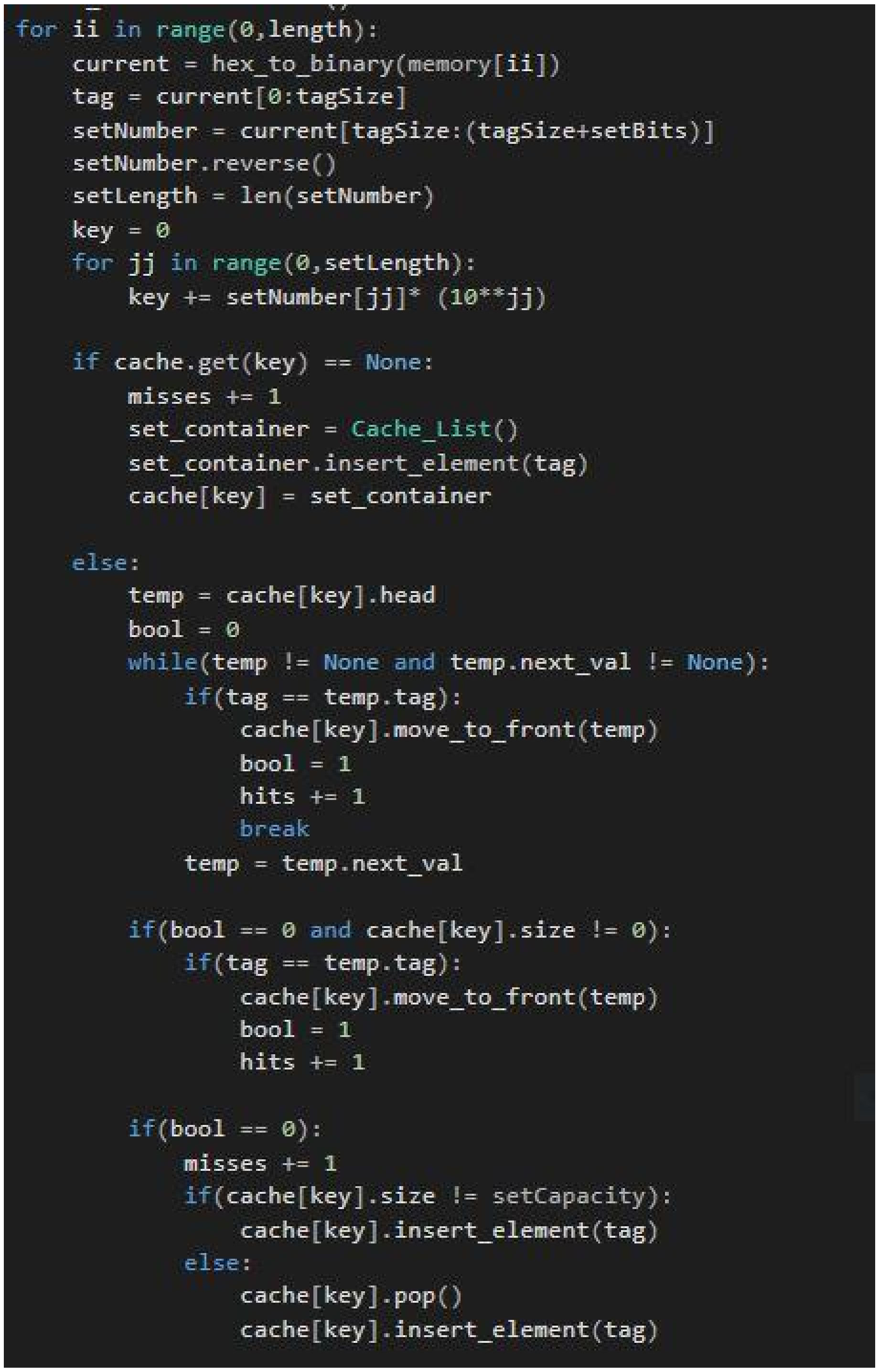
```
#Size Variables
offsetSize = int(math.log2(blockSize))
lines = 2**(int(math.log2(cacheCapacity)) - offsetSize)
numSets = 2**(int(math.log2(lines)) - int(math.log2(setCapacity)))
setBits = int(math.log2(numSets))
tagSize = addressSize - offsetSize - setBits
```

Figure 7 (Set Associative Variables)



```
for ii in range(0,length):
    current = hex_to_binary(memory[ii])
    tag = current[0:tagSize]
    setNumber = current[tagSize:(tagSize+setBits)]
    setNumber.reverse()
    setLength = len(setNumber)
    key = 0
    for jj in range(0,setLength):
        key += setNumber[jj]* (10**jj)

    if cache.get(key) == None:
        misses += 1
        set_container = Cache_List()
        set_container.insert_element(tag)
        cache[key] = set_container

    else:
        temp = cache[key].head
        bool = 0
        while(temp != None and temp.next_val != None):
            if(tag == temp.tag):
                bool = 1
                hits += 1
                break
            temp = temp.next_val

        if(bool == 0 and cache[key].size != 0):
            if(tag == temp.tag):
                bool = 1
                hits += 1

        if(bool == 0):
            misses += 1
            if(cache[key].size != setCapacity):
                cache[key].insert_element(tag)
            else:
                cache[key].pop()
                cache[key].insert_element(tag)
```

Figure 8 (FIFO Set Associative)



```
for ii in range(0,length):
    current = hex_to_binary(memory[ii])
    tag = current[0:tagSize]
    setNumber = current[tagSize:(tagSize+setBits)]
    setNumber.reverse()
    setLength = len(setNumber)
    key = 0
    for jj in range(0,setLength):
        key += setNumber[jj]* (10**jj)

    if cache.get(key) == None:
        misses += 1
        set_container = Cache_List()
        set_container.insert_element(tag)
        cache[key] = set_container

    else:
        temp = cache[key].head
        bool = 0
        while(temp != None and temp.next_val != None):
            if(tag == temp.tag):
                cache[key].move_to_front(temp)
                bool = 1
                hits += 1
                break
            temp = temp.next_val

        if(bool == 0 and cache[key].size != 0):
            if(tag == temp.tag):
                cache[key].move_to_front(temp)
                bool = 1
                hits += 1

        if(bool == 0):
            misses += 1
            if(cache[key].size != setCapacity):
                cache[key].insert_element(tag)
            else:
                cache[key].pop()
                cache[key].insert_element(tag)
```

Figure 9 (Set Associative LRU)

4) Tests

The plotting of graphs was done using matplotlib. This makes it easy to have multiple plots on one figure. Therefore to test the difference between cache designs and replacement policies, every cache design and every replacement policy got a plot. This meant that every test would test the difference between cache designs and replacement policies.

For control purposes, all the following tests were conducted on the gcc.trace file using the HiPerGator supercomputer through VPN access. Using the same file and machine meant that all data was directly comparable. Since the most common set-associative caches in the real world are 2-way set associative and 4-way set associative, these were chosen as the set-associative caches to test.

The first test was to test the change in hit rate with increasing cache size. Block size was kept constant and cache size was increased by powers of 2. Cache size was chosen as a test parameter since theoretically as the cache grows it can hold more memory which would increase the hit rate.

The second test was to test the change in time with increasing cache size. Block size was kept constant and cache size was increased by powers of 2. Theoretically, fully associative should take the longest amount of time since all of the blocks are stored in one set and the entire set must be checked which is O(N) time complexity. To test this hypothesis, this test was chosen.

The third test was to keep cache size constant and check the hit rate with block size increasing by powers of 2. Theoretically, the bigger the block size the more memory addresses in a block and this exploits locality since memory addresses near the memory address just used are more likely to be used. However, if the block size gets too large there will be far too few blocks in a cache which will decrease the hit rate.

The following test *will* change the file. Another file will be tested to ensure that the results were not a fluke of the trace file chosen for the previous tests.

The fourth test will check the hit rates on both the swim trace file and gcc trace file with a constant cache size of 1024 bytes and a constant block size of 8 bytes.

The fifth test will test the difference with increasing set-associativity. Since there is less computation time involved since the cache and block size are held constant now, this is more feasible to do (run time for the simulation can be quite long when multiple runs of different cache/block sizes are run).

5) Results:

a) Test 1

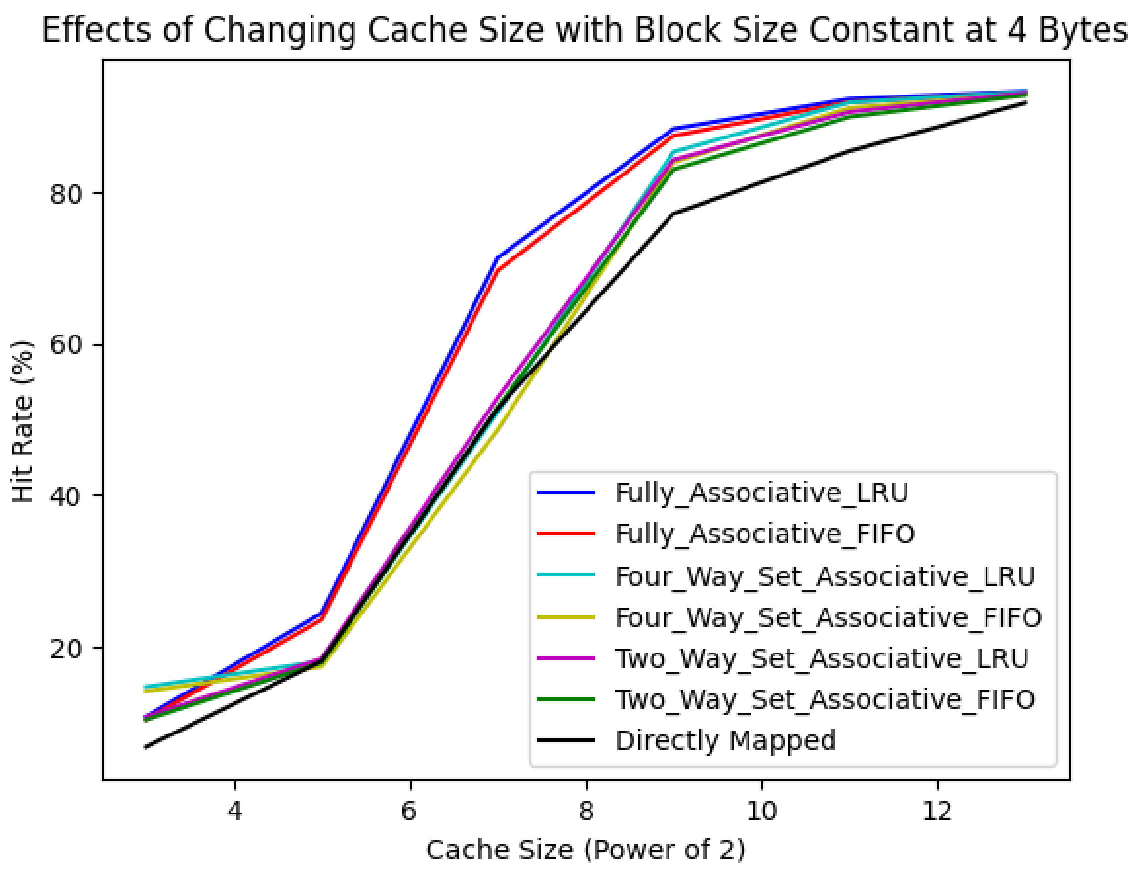


Figure 10

Figure 10 shows the results of the first test. The general behavior was that directly mapped had the lowest hit rate, the set associative caches were in the middle, and the fully associative caches had the highest hit rate. There was a small deviation from the trend at a cache size of 2^6 to 2^7 bytes where the four way set associative cache with a FIFO replacement strategy actually had the lowest hit rate. This is thought to be because the sets per cache was very low at this cache size and the poor performance of a FIFO replacement policy.

The LRU replacement algorithm outperformed the FIFO replacement algorithm in every cache design. In each case, the difference in hit rates between the two was not very large but it was noticeable.

In every cache design, an increasing cache size came with an increased hit rate. At a cache size of 2^{16} bytes every cache design had a hit rate of above 90%. Of course, the main memory size was only 2^{32} bytes so this cache size is quite large for the size of main memory. In a real cache, this would likely not be the case and hit rates would likely not be able to get so high.

b) Test 2

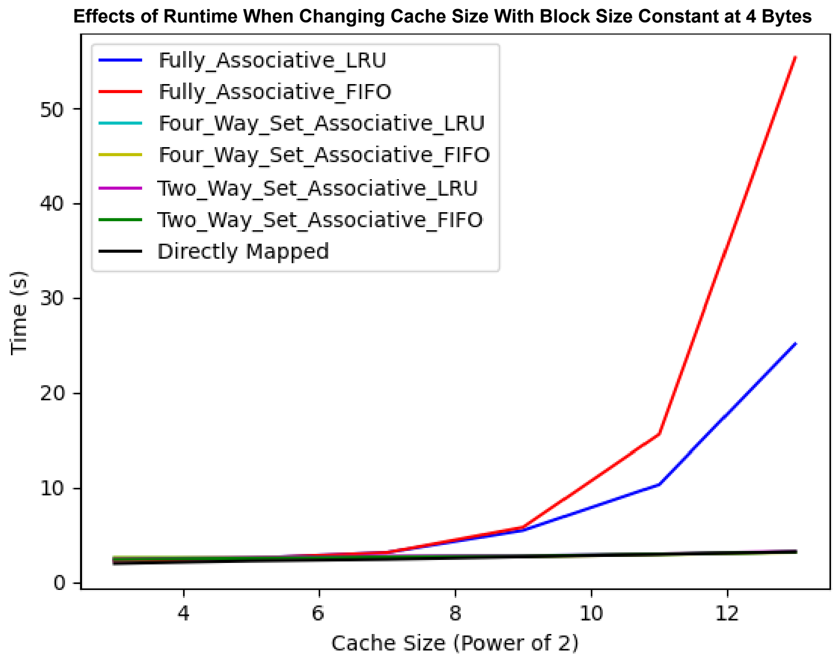


Figure 11

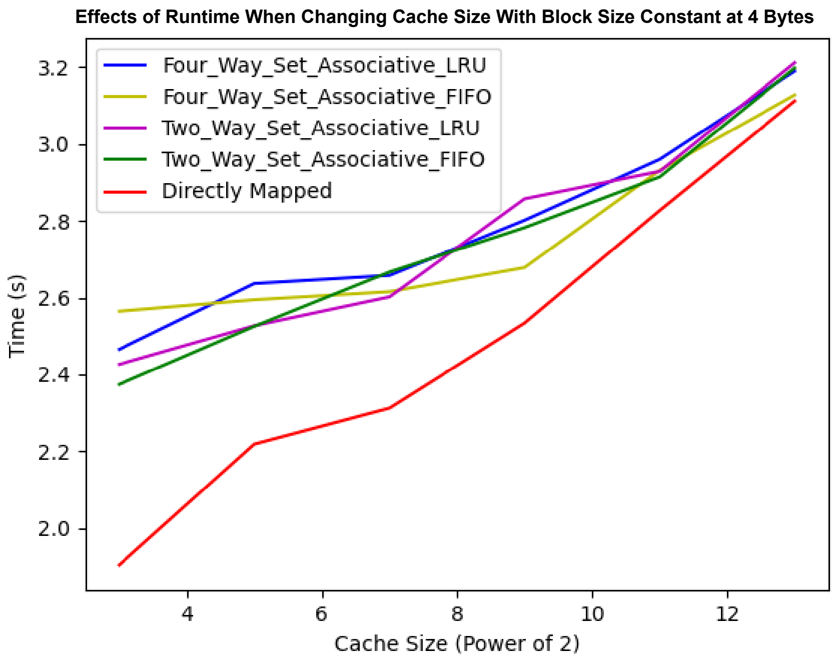


Figure 12

Figure 11 showed a huge disparity in runtime between fully associative and the other cache designs. The LRU full associative design had a longer runtime than the FIFO full associative design. Figure 12 zooms in on the other cache designs. In all cases, directly mapped had the lowest runtime. There does not seem to be a correlation between the 2-way set associative and 4-way set associative in terms of runtime as they criss-cross multiple times for both replacement policies.

c) Test 3

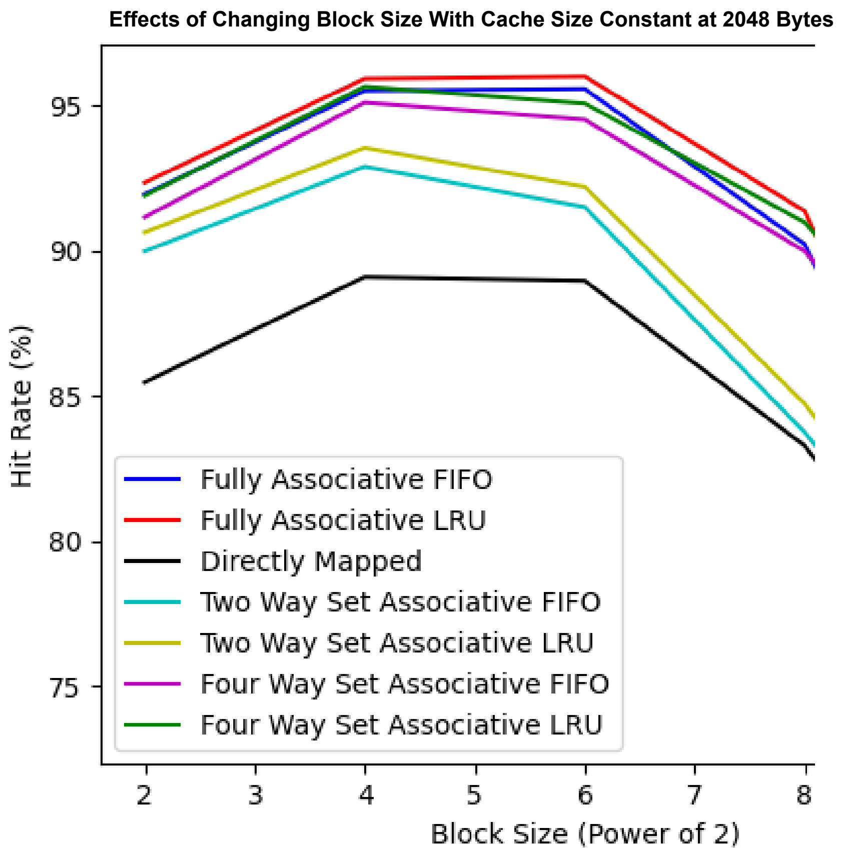


Figure 13

Figure 13 shows the hit rates for each cache design with increasing block size. In all cases, the fully associative cache design has a higher hit rate than the other cache designs. Next is the four way set associative designs. Then the two way set associative designs. The lowest hit rate was always the directly mapped design. For each design, the highest hit rate was the one that utilized the LRU replacement policy over the FIFO replacement policy.

The trend is as was expected. Initially, the increasing block size takes advantage of locality. That is that the next memory address that will be used will be in a nearby memory address from the one just accessed. As we increase the block size, we include more of those nearby memory addresses. However, at a block size of 2^6 the block size became too big and the hit rate began to suffer. This is due to a lower number of blocks.

d) Tests 4

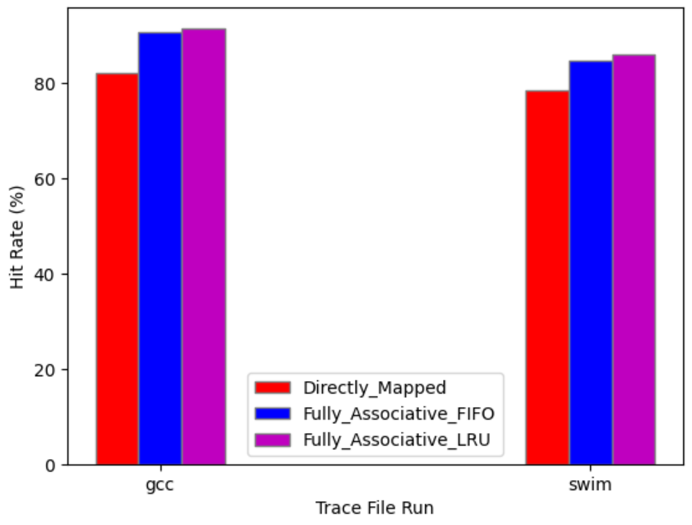


Figure 14

In Figure 14 we see the continued trend that a directly mapped cache has a far lower hit rate than a fully associative cache. Looking at Figure 15 and Figure 16, we see that directly mapped is visually less than all the set associative caches. This shows that even across different files, the same patterns hold.

e) Test 5

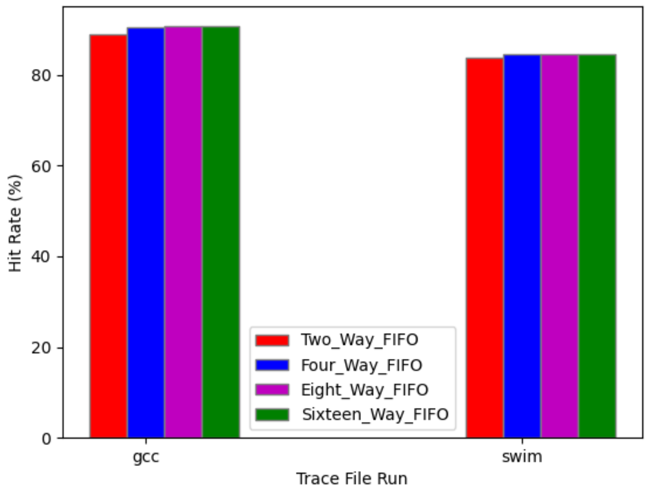


Figure 15

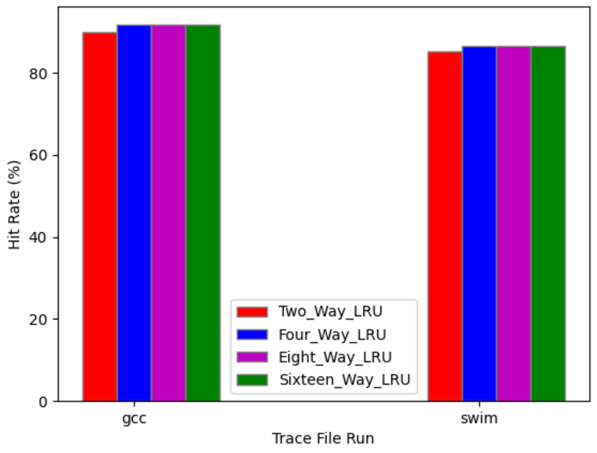


Figure 16

Test 6 shows that there is not much of a difference of increasing set associativity past 4-way associative. This would explain why most caches are either 2-way set associative or 4-way set associative, after this there is not much gained from increasing the associativity at this scale of memory. Perhaps when getting into really high memory addresses this would matter since some smartphones and laptops user set-associative caches with higher associativity than 4.

5) Conclusion

In conclusion, the behavior of the cache simulation closely matched the theoretical predictions of what the results would be. In all cases, directly mapped had the lowest hit rate, with set associative caches in the middle, and fully associative caches had the highest hit rate. However, the time was the lowest for the directly mapped cache, slightly higher for the set associative caches, and much higher for the fully associative caches. This explains why set associative caches are so common in industry, they have the benefits of both direct mapped and fully associative cache designs.

Increasing cache size increased the hit rate but with that came increased runtime. This means real cache design must find a middle ground between a good hit rate and a fast enough runtime. Also, increasing block size increased the hit rate due to the principle of locality. However, if the block size is too large then the hit rate begins to suffer as the number of blocks in a cache decreases. Therefore a middle ground must be found for the block size as well. Large enough to take advantage of locality but small enough that there are not too few blocks.

In terms of replacement policy, the least recently used policy had a higher hit rate for every cache design than the first in first out policy. LRU also had about equal runtime to FIFO. Therefore, there is no clear advantage to using FIFO and LRU should be used instead. Indeed, LRU is much more commonly used in industry compared to FIFO for this reason.

Increasing the associativity of a set brings diminishing returns in the hit rate achieved after about a 4-way associative set size. Increasing the associativity brings the behavior closer to a fully associative cache design so this means the run time increases. If there is not much of a difference in hit rate between 4-way and a higher associative cache, for this size of a main memory, then there is no point in increasing the associativity. This would explain why 2-way set associative and 4-way set associative are such common cache designs in industry.