Project - 1

Cache Identification and Evaluation

COEN 313: Advanced Computer Architect

Submitted By

Group 3: Ryan Cooper, Addison Fattor, Maxen Chung & Nithin Vasudevan

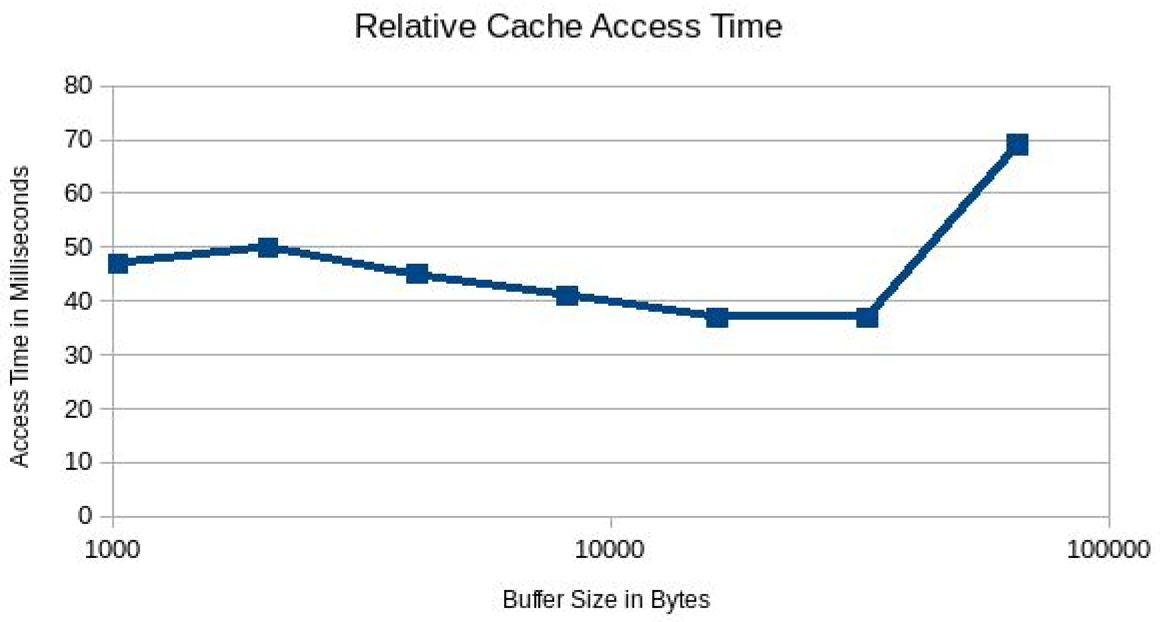
Instructor's Name	**: Dr. Amr Zaky**
Date of Submission	**: Nov 2nd, 2017**

CPU Stats:
http://www.cpu-world.com/CPUs/Core_i7/Intel-Core%20i7-4578U%20Mobile%20processor.html

Part 1:

Cache Size:

Our method of measuring the cache size was to time how long it took to access elements in a buffer sequentially. The timing method we used was to count clock cycles during execution. Our first buffer started at a size of 1024 bytes. We doubled the size of the buffer after every iteration until the time it took to go through the whole buffer was substantially increased (by 30 ms) with respect to the size of the buffer. This meant that we would time how long it took to access all of the data in the buffer and would divide that time by the relative increase in size from the original buffer size. When we noticed that a buffer took longer than our threshold of 30 ms, we determined that that buffer size was two times the size of the cache. Below is a graph of the amount of time it took to access all the data in the buffer relative to the buffer size.



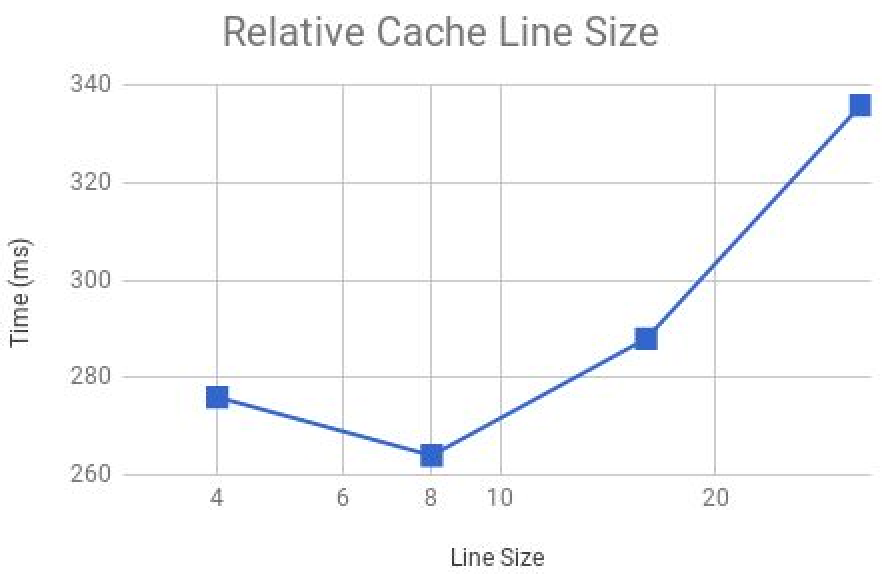
The large increase in relative access time between buffer sizes of 32768 and 65536 bytes made it clear that the cache size is 32KB in size. This matches the processors specifications.

To combat the effects of virtual memory, we ran the test after a reboot of the machine and when there were no other processes running on the device. To prevent L2 cache from becoming a problem between different buffer sizes, we would create the buffer at the start of each test so that all buffer sizes would suffer the same penalty for a compulsory miss at the start of the test. After the compulsory miss, we assumed that all of the buffers would be contained within L2 cache which would make the timings comparable between tests.

Line Size:

In order to measure the line size of the cache, we use the result of the cache size determination and test different line sizes, starting from 1 and going to 128 bytes, increasing by a power of 2 each time. For each line size we are testing, we create an array twice the size of the cache, and write to elements of the array spaced out by the line size we are currently testing. We write these elements to the array 1000 times to get a good amount of timing data, and we do the whole operation 10 times for each test line size to get an average for that line size. We time the 1000 accesses by counting the number of clock cycles, which is an accurate time measurement. We do the timing calculation 10 times and take the average in order to minimize the time fluctuations between different runs, which is the same reason we measured many array accesses instead of just a few. Because we write to the full array, a smaller line size will result in more accesses, which results in more time for execution. In order to remedy this, we multiply the time it takes to do the accesses by the line size, which offsets the increased amount of accesses for a smaller line size.

After each measurement of line size, we compare to the previous line size's time measurement, and if the time is off by 50ms, which we determined to be a significant enough increase through observation, then that line size is the correct line size. The results of this test is shown in the figure below. This shows that the time to access elements from the cache line increases dramatically after the correct cache line is chosen. What we don't know, is why the time dips when testing a line size of 8 bytes, but we can pass this off as a time fluctuation when running the tests.



The reason this code works, is because accesses within the same line will not cause cache misses, so when the line size we are testing is smaller than the actual line size, we will still be getting hits for at least 50% of the accesses. But when the test line size becomes the correct line size, every single access will be a miss, which means the time to complete the 1000 accesses will increase more drastically than previous. If we wanted to ensure this line size was correct, we could continue with the next test line size, and the timing would be similar to the one before. We did not do this because while testing the code beforehand, this was the case, so the function called to test the line size returns immediately when finding the line size so the associativity can be determined.

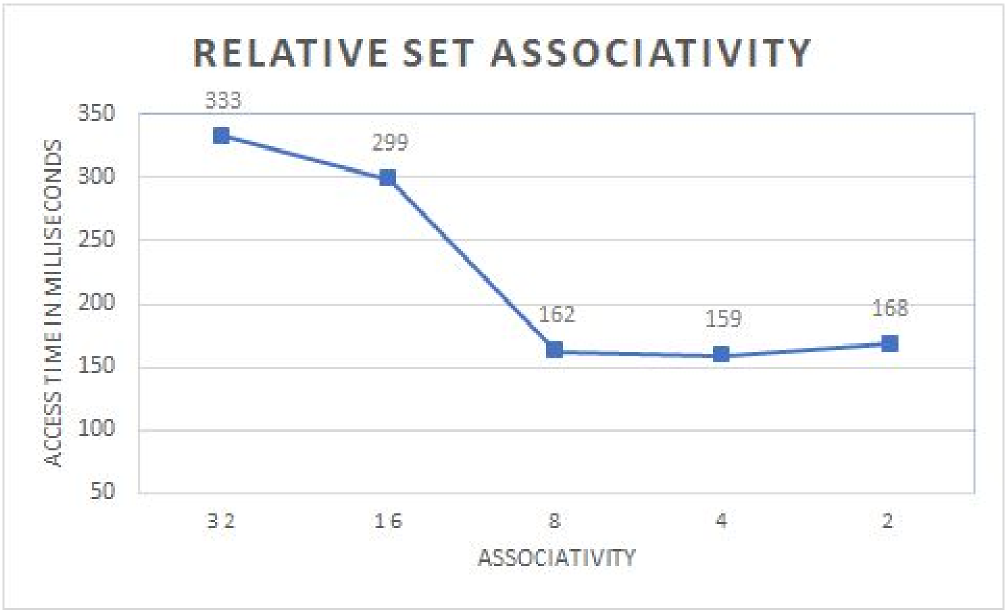
The same precautions were taken when running this test as when running the test before, mainly running soon after the machine was rebooted, and ensuring no other applications were running so the program could have a larger portion of the CPU time.

Associativity:

Our method of measuring the associativity is to first fill up an array sequentially equivalent to twice the cache size and then keep accessing it iteratively until we find the right associativity. The right associativity is the one which gives a great dip in the access time in milliseconds. To make this measurement we started to count the access time for each of the iteration by first assuming that our cache was 32-way set associative and thereby reducing it by two each time until it reaches to 2-way set associative.

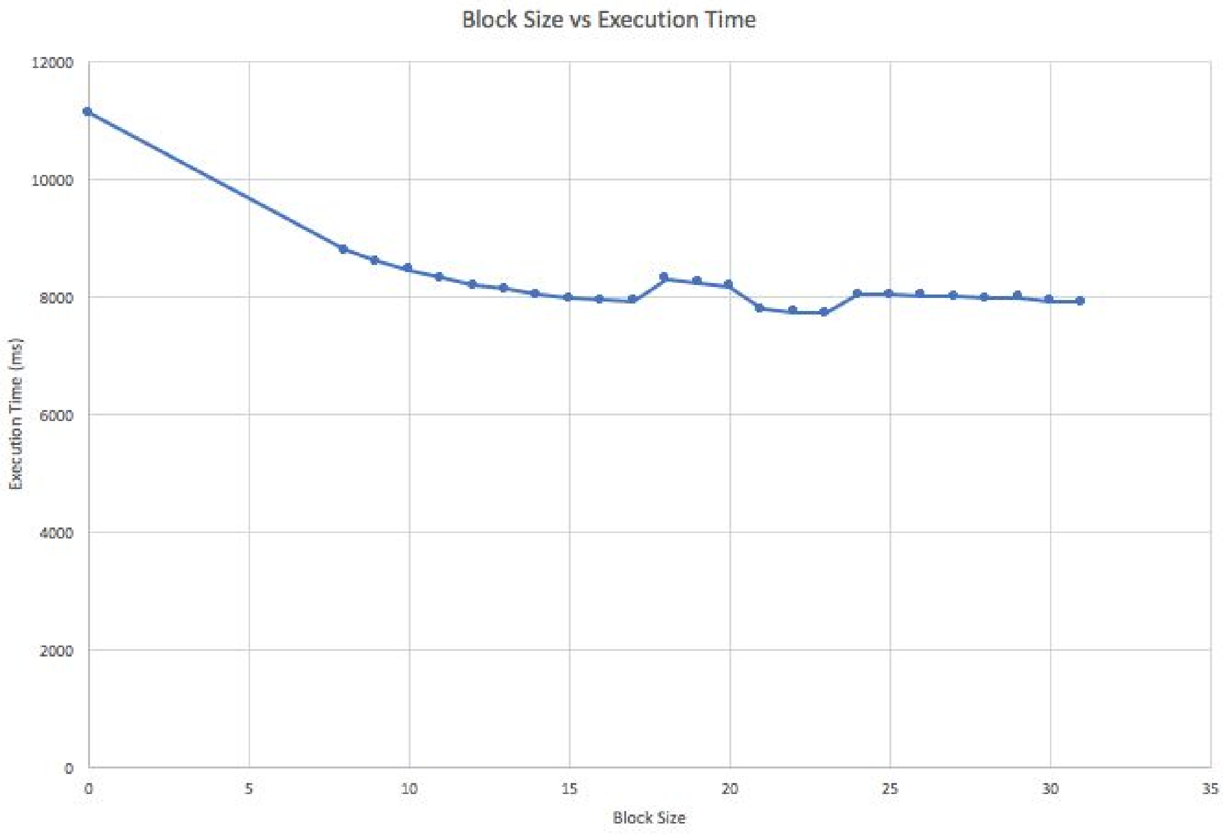
When this measurement was made, our 4-way set associative had the lekast access time which could be as a result of our timing function, but the difference between 8 to 4-way set associative was meagre. 8-way set associative had the greatest dip in access time of approximately 137 Milliseconds from the previous one among all the predictions carried out. This confirmed for us that our system's cache was an 8-way set associative cache.

Below is the graph and the snippet of the access time for each of the associativity experiment which was carried out.



Part 2:

Methodology: To calculate the execution time of a matrix multiplication, we used the average time from 10 runs for both non-blocking and blocking with block size 8-31. The computer was restarted prior to running the test and we tried to ensure that as little applications were running as possible. Additionally, the screen saver was turned off so the computer would not have to deal with it (the whole testing took over an hour). We would expect the optimal value for our cache to be somewhere around sqrt(M/3), where M is the side length. This would place our optimal value at roughly 18 for our side length of 1024. We were looking for where there was a dip in execution time, followed by an increase in execution time. This would indicate to us that the 3 matrices (2 input and 1 output) were filling up the cache maximally while not exceeding the cache size.



Non-Blocking: To calculate the time it takes to execute a matrix multiplication with a non-blocking cache, we used the shared code provided in class which is three nested loops that iterate over each matrix. As expected, the time that it took for a non-blocking cache (Block size 0 on the graph above) was the slowest out of all of the runs by a significant margin, being roughly 26% slower than the next slowest, which had a block size of 8.

```
for(i = 0; i < sideLength; i++)
      for(j = 0; j < sideLength; j++){
         sum = 0;
         for(k = 0; k < sideLength; k++){
            sum = sum + a[i][k] * b[k][j];
         }
         c[i][j] = sum;
      }
   }
```

Blocking: To calculate the time that a blocking matrix takes, we use the six nested for loops as shown in the below code snippet. These loops are iterating over the blocks, and the rows and columns inside of the blocks, the product of which is accumulated in c. The results of testing the blocking cache are a bit confusing because of the double local minimums that we had. The first local minimum occurred at with a block size of 17. This makes sense because the 18 we estimated earlier did not take into account the local variables that we also have to store. However, the second local minimum occurred at 23, which does not make much sense based on the fact that a block size of 23 would not allow the blocks to reside in the L1 cache. One explanation could be the L2 cache kicking in to hold more of the matrix in memory. Another possible explanation is that the lines of cache are conflicting for block sizes of 18, 19, and 20, leading to the “hump” in the graph.

```
  for (ii = 0; ii < sideLength; ii += blockSize) {
      for(jj = 0; jj < sideLength; jj += blockSize) {
         for (kk = 0; kk < sideLength; kk += blockSize){
            for (i = ii; i < min(sideLength, ii + blockSize); i++){
               for(j = jj; j < min(sideLength, jj + blockSize); j++){
                  for(k = kk; k < min(sideLength, kk + blockSize); k++){
                     c[i][j] = c[i][j] + a[i][k] * b[k][j];
 }}}}}}
```

Code From: http://www.netlib.org/utk/papers/autoblock/node2.html