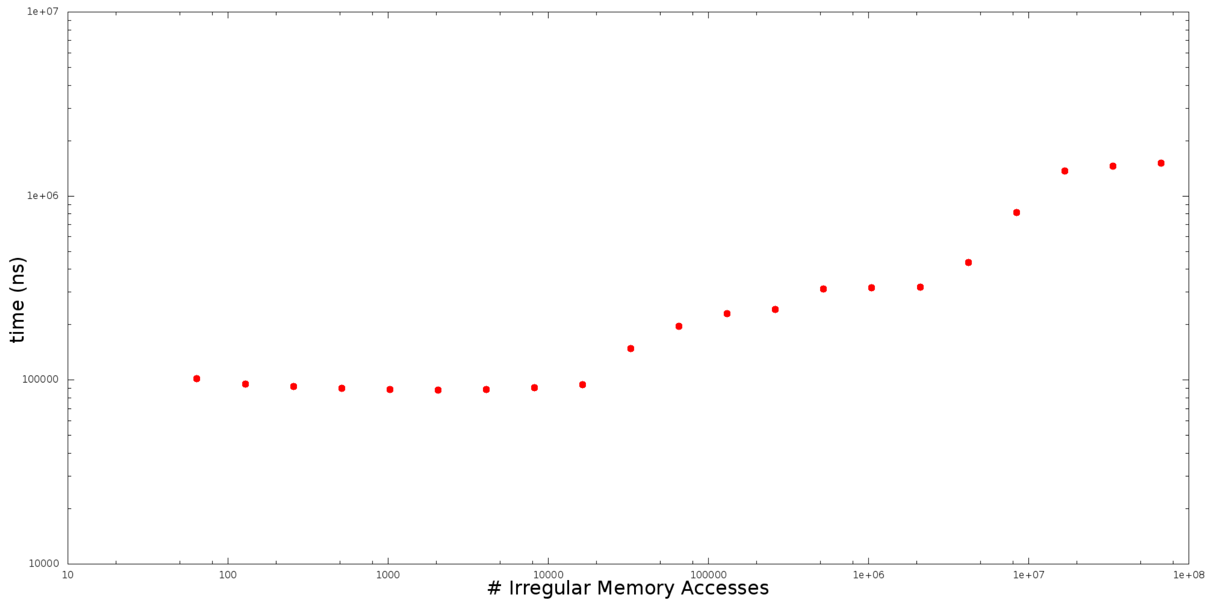
MP0

Name: John Simone
Net ID: jcsimon2
Date: 9/13/2017

Part A

Running mp0_a and plotting the results gave the following graph:



As we can see in the above plot, there appear to be distinct regions where the time is constant, and then it jumps up to a new flat region. The values where this happens are approximately 30,000 and 1,000,000. Since the time taken does not appear to be directly dependent on the size used of the arrays, these jumps must be caused by differences in accessing the arrays from memory. Given the array sizes where these jumps happen, we can estimate the sizes of the different caches, as the time will jump once the cache is full and the program will have to access slower memory. The memory needed to store both of these arrays is $2 \times 4(size(int)) \times length(array) =$ 234kB and 7.62 for the L2 and L3 caches, respectively. The values found on the nodes are 256kB and 20MB for these caches, which are not far off from our estimated values. The relatively large amount of error for the size of the L3 cache can be partially attributed to the slow transition between the last two levels. Given the data we have, this value could have been read to be up to 2.1 million, which would give a size estimate of 16MB, much closer to the true value.

Part B

The following table records the running time for each program at each optimization level.

mp0_b Results

(seconds)	O0	O1	O2	O3
test1	0.00204	0.000323	0.0	9.5e-8
test2.1	0.00313	0.00107	0.00107	0.00116
test2.2	0.00239	0.00104	0.00111	0.00115
test3.1	0.0183	0.00494	0.00493	0.00184
test3.2	0.00322	0.000971	0.000679	0.000503
test4	0.0179	0.00513	0.00523	0.00520
test5(1023)	6.555	1.401	1.236	1.234
test5(1024)	8.494	1.909	1.879	1.866
test(1025)	6.578	1.415	1.255	1.268

Test1

For test1, we see the time drop by a factor of 10 at the first optimization level, and then the time drops to below the accuracy of the timer. This second drop comes from the inner loop being skipped, as the variables updated in the loop are never printed or written.

Test2

For test2, there were two versions. The first had a for loop for 1 million iterations, the second had a for loop for a quarter of the first, with 4 lines in the loop. By unrolling this loop, all of the iterations are still done, but the condition is checked only 20% of the time, speeding up the loop. Once up optimizations are applied, both versions take the same amount of time, and this time doesn't change as the optimization level is increased, indicating that the proper unrolling size is enabled immediately.

Test3

For test3, we iterate through an array, adjusting all the values as we go. In the first part, we take strides of 1024; in the second the stride is 1. This can be seen from the unoptimized times, the stride-1 version is over 5 times faster due to less cache misses. First level optimization reduces running time by a factor of $\sim$4, most likely by unrolling the for loop.

Test4

For test4, we transpose a matrix. Again, we see $\sim$4 times reduction in time after the first optimization flag, again by unrolling the for loop

Test5

For test5, we have three arrays and for each entry in the first two, multiply them and assign that value to the third array. From the unoptimized run, we see that this is slower when the size is 1024, a power of two. This is caused the workings of the set associative cache. Since the array size is a power of two, the the memory locations we are accessing are often assigned to the same set in the cache. This results in extra cache misses, as memory will be kicked out of the cache before we are done using it. For sizes not a power of two, this problem does not occur. Compiling this with optimization flags again uses unrolling, resulting in a $\sim 4\times$ time reduction.