Aaron Berns
Project 3: False Sharing

Machine
MacBook Pro, i5, 2 cores

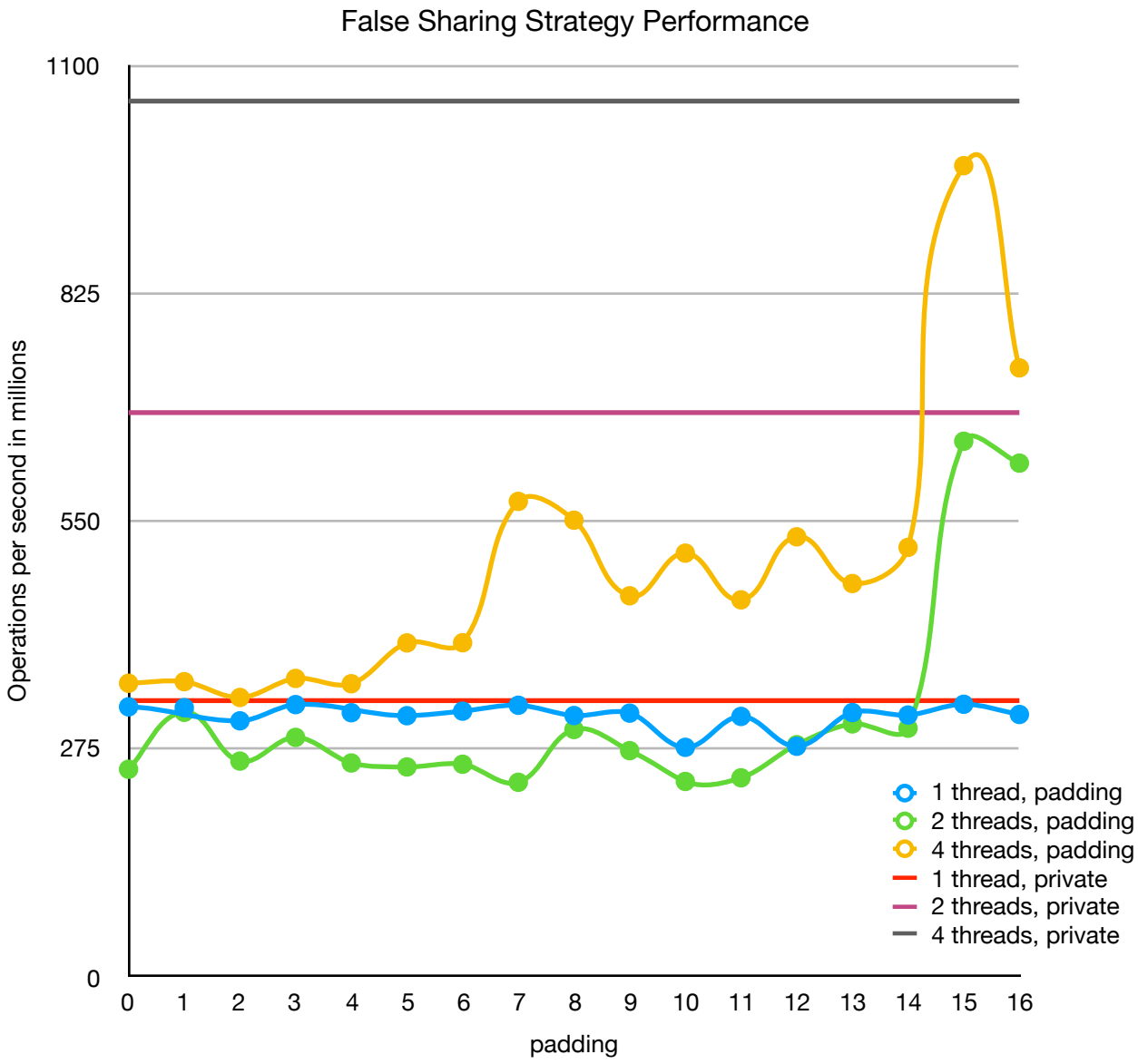
Results

Operations per Second in Millions: Fix 1, Cache Boundary Forced

	1 thread	2 threads	4 threads
padding = 0	325.6	250.2	354.1
1	324.6	319	356
2	308.7	260.1	336.9
3	328.3	288.7	359.9
4	318.5	257.8	353.5
5	314.9	253	402.8
6	320.4	256.3	403.0
7	327.5	234.5	573.7
8	315.1	298.2	551
9	318.0	272.7	459.7
10	276.7	235.5	511.2
11	314.0	239.9	454.8
12	278.0	279.9	530.9
13	318.7	305.0	474.5
14	315.7	299.8	518.2
15	328.7	646.3	979.1
16	316.4	619.9	734.9

Operations per Second in Millions: Fix 2

1 thread	2 threads	4 threads
336.2	680.9	1057.1



Discussion

Performance was measured by calculating the number of operations in total (someBigNumber * number of threads) and dividing it by the time in seconds multiplied by 1,000,000 to get millions of operations per second. The first time I ran the program I got some weird fluctuations in performance that did not match what I expected from what I knew about padding from the cache lecture. I decided that the differences in performance were the result of the array not being allocated on a cache line. To try to clean up the results I used the method described in

the cache lecture to make sure the array began on a cache line. This made the performance trends somewhat clearer.

Most aspects of the performance data for fix #1 followed the trends set out in the cache lecture. Using one thread led to a relatively straight line as performance was not affected by false sharing among threads. For two threads, I expected a large jump in performance once the gap became 15 as this would push the second struct in the array onto its own cache line. This jump in performance was seen. There were several fluctuations before the 15 mark but I'm not sure they can be attributed to caching issues as opposed to other unknown factors because both structs would be on the same cache line the entire time. Had the number of structs been four and the number of threads two, I think the performance would mirror that of the lecture data.

The performance data for four threads and four structs closely resembles the lecture trends. There is a distinct jump in performance at a padding of 7 as two of the four structs are bumped onto another cache line. There is a significant bump in performance at a padding of 15 as each struct in the array gets it's own cache line.

When the padding reaches 16, the performance drops with both two and four threads, most significantly with four. The dip isn't huge with two but it's noticeable. This might be because the padding from the structs would start to spill over onto the next cache line although my understanding is that this shouldn't really be a problem in terms of false sharing as the memory locations of the 'value' portion of the structs would never again be on the same cache line. It could be a result of the structs not starting on cache lines as they did with a padding of 15. If the entire struct needed to be loaded into cache in order to operate on the 'value' because it they are each an element of a larger array, then each struct would require fetching two cache lines at a padding of 16. If this is the case, performance should take a hit at a padding of 32 as well. A padding of 31 would have each struct at exactly two cache lines each. 32 would make it three. To test out this hypothesis I looked at the performance of two and four threads around the 14 - 18 and the 27 - 33 range and while there is performance fluctuation I didn't find anything conclusive. Not being sure about how structs are treated when being cached makes it difficult to know.

The fix #2 data also followed the trend of the lecture data. For each number of threads, using local variables gave maximum performance right away. This was expected as the cache lines containing the variables were loaded from the separate stacks, making sure that false sharing didn't occur as the variables, being in separate stacks, are not stored contiguously.