

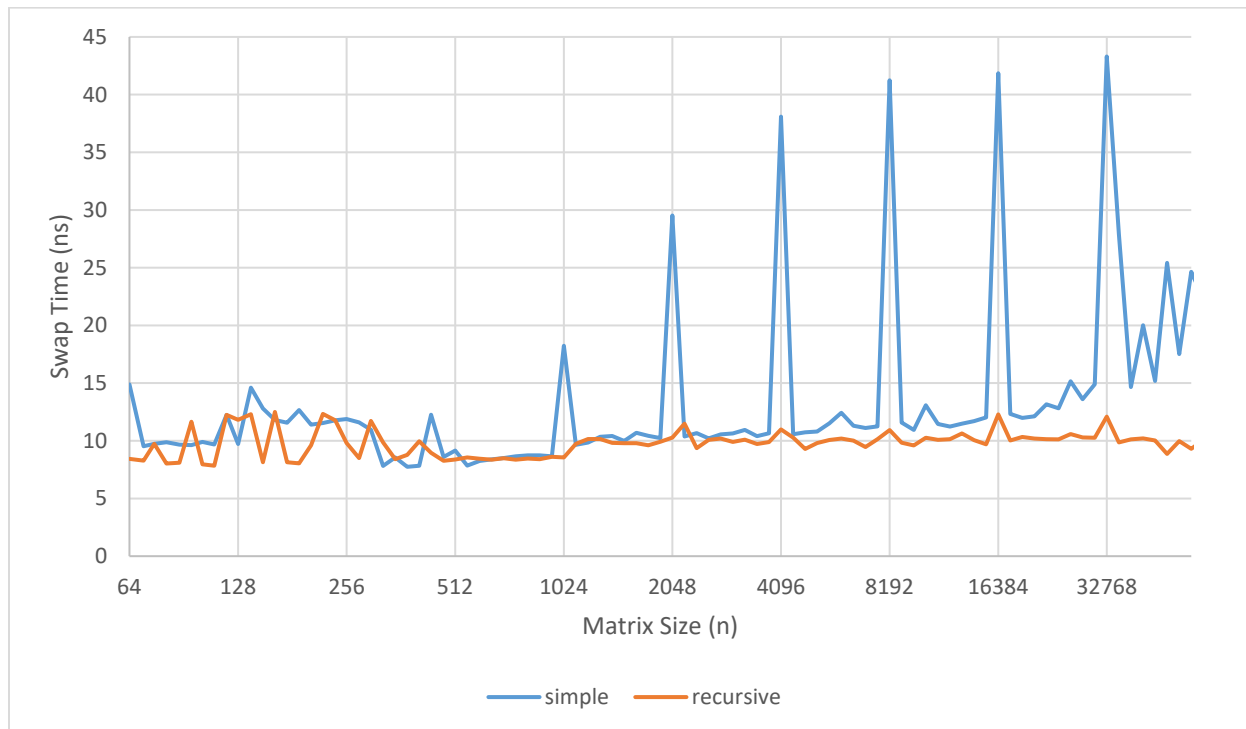
Assignment 3

This assignment analyzes the performance of two matrix transposition algorithms as pertaining to processor cache: a simple nested loop and a recursive cache-oblivious algorithm. Each matrix is in a row-structure of size $n \times n$, where each slot is *int* bytes.

I. Hardware Test

The hardware test plots the average amount of time (in nanoseconds) that a single swap takes in a matrix transposition for a given matrix of size $n \times n$. This test was benchmarked on an Intel Core i7 7700k (4.2 GHz) processor with 8 MiB cache. Note that the horizontal axis uses a log scale.

Figure 1: Matrix Size vs. Average Swap Time



As expected, the recursive (cache-oblivious) matrix transposition algorithm runs faster on average than the simple algorithm due to its ability to use the processor cache effectively. As the matrix size increases, the chance that the elements which the simple algorithm needs will be in the cache decreases substantially, since the memory accesses on the row-based layout of the matrix are far apart. The recursive algorithm tends to transpose elements that are close together, increasing the likelihood that the next swapped elements will also be in the cache (even when the matrix is large).

One curious observation is that in the simple transpose algorithm, the average swap time spikes substantially when the matrix size is a power of two. This occurs due to the cache being set associative (as opposed to being fully associative). Each set in a set associative caches use a fixed number of lines, and thus the set size is a factor of the entire cache size. If memory is accessed in such a way that the boundaries between memory accesses map to the same cache set, then the accessed elements will be evicted from the cache much quicker as we are only storing the recent memory in one set. This boundary is known as the critical stride, which is a power of two in modern processors. Hence, transposing matrices of size of a power of two with the simple algorithm will tend to use the cache in the worst way possible.

II. Cache Simulator

The cache simulator simulates the number of page faults that would occur in an LRU fully associative cache. The following plots display the number of cache misses that would occur for both algorithms as a function of the matrix width n . Multiple plots are shown to examine the behavior of changing cache parameters B and C , where B is the size of one page (in *ints*) and C is the number of pages. Each curve represents varying parameters B, C .

To improve readability, the horizontal axes use a log scale.

Figure 2: Simple Algorithm (B, C)

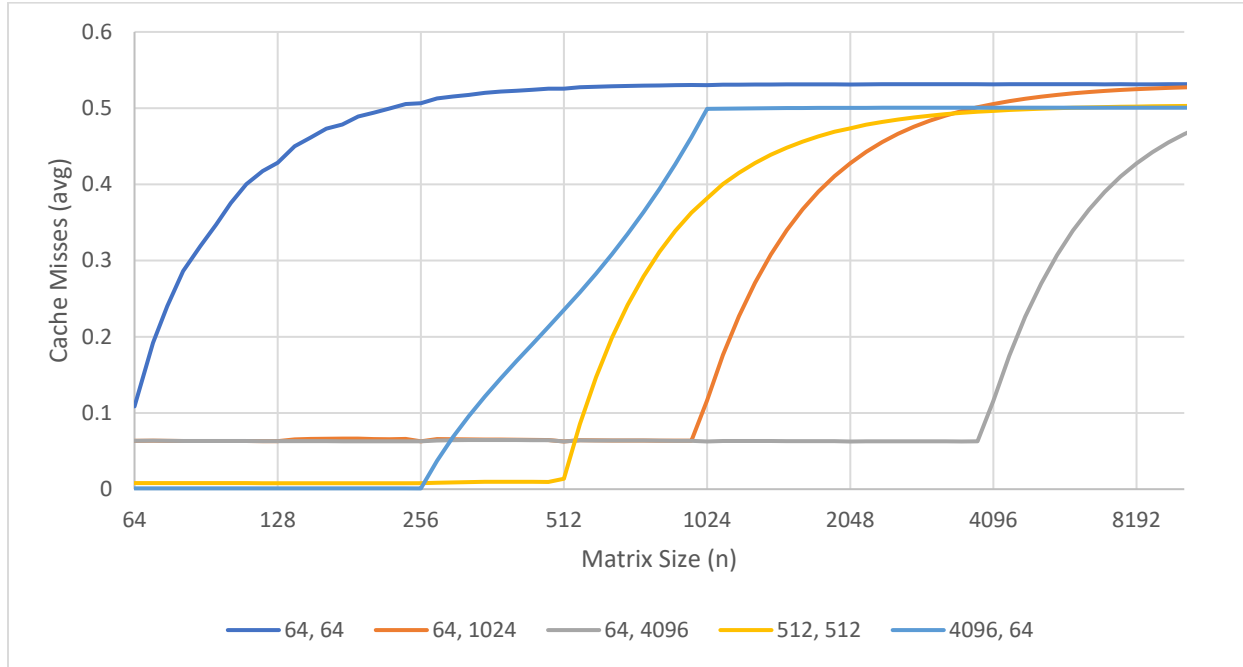
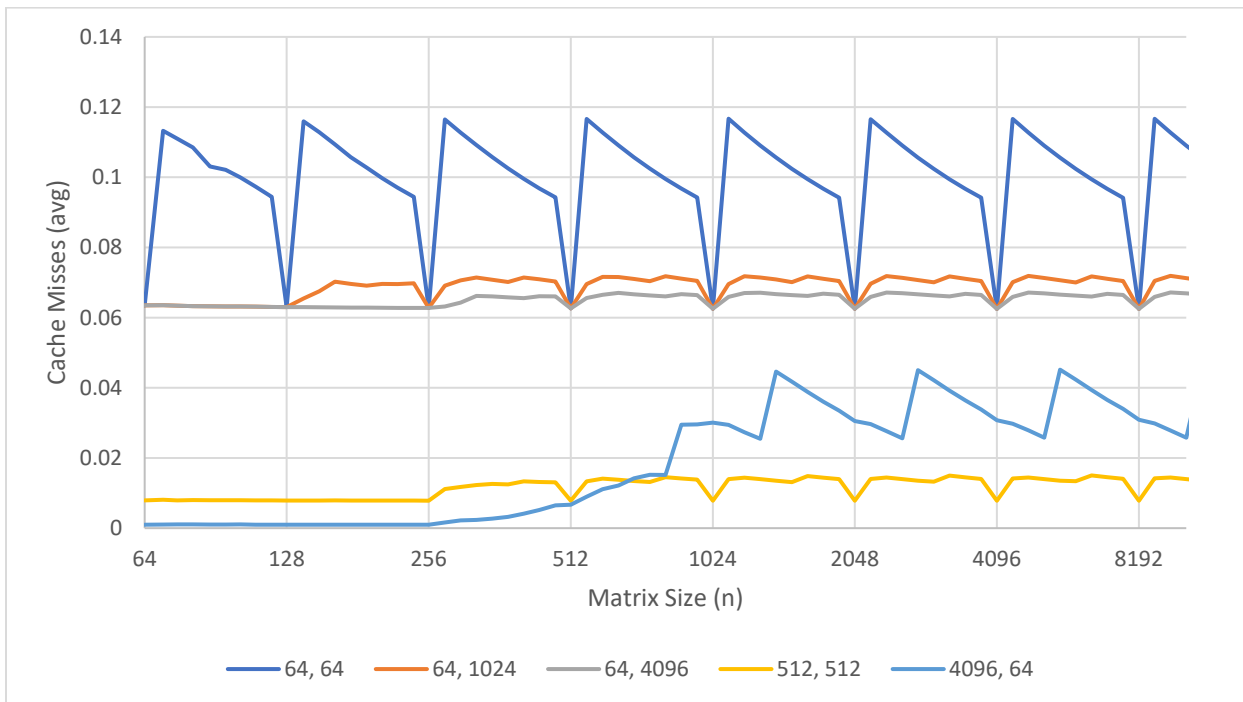


Figure 3: Recursive Algorithm (B, C)



It can be seen from the plots that as the number of pages C increase, the more a matrix can be fully stored in cache; the number of cache misses of the simple algorithm decreases up to the point at which the matrix can no longer be fully stored in the cache. Up until that point, the simple algorithm behaves nearly identically to the recursive algorithm.

Due to the row-based layout of the matrix, we can see that the number of cache misses in the simple algorithm approaches 0.5. Intuitively, the simple algorithm will be iterating over each row, which will be loaded into cache after the first page fault, reducing the number of cache misses when the next row element is accessed. But it also iterates over the columns, which will cause a page fault if the corresponding row has been evicted from cache. For large matrices, this will occur for each column access, resulting in an average one out of every two accesses in a single swap missing the cache.

By increasing the size of each page B (and holding the total cache size BC constant), it is apparent that the simple algorithm causes fewer cache misses for small matrices. This is because there are fewer cache boundaries in which a cache miss can occur, i.e., more rows in the matrix are loaded after a cache miss, reducing the likelihood of a subsequent cache miss. The whole matrix is fully stored in the cache initially, but as the matrix size exceeds the cache size the number of cache misses returns to its base rate of slightly above 0.5. This presents a relationship between B and C for caches of equal size (in the simple algorithm): increasing B decreases the number of cache misses for small matrices, and increasing C allows for larger matrices before the number of cache misses starts increasing.

On the other hand, the recursive algorithm shows improvement across the entire spectrum of matrix sizes, indicating that it can utilize the cache better even when the entire matrix does not fit into cache. For instance, if the matrix size is a power of two, then the cache can be used most

efficiently because the submatrices transposed in the recursion step will perfectly align with the cache size and so are less prone to cache misses on row boundaries. This is different from the first graph, since we are swapping small submatrices.

As C increases, the number of cache misses of the recursive algorithm decrease on boundaries not of a power of two. This is because alignment issues become less of an issue of there are more pages to spare. More rows of a submatrix can be fit into the cache.

By increasing the size of each page B (and holding the total cache size BC constant), the recursive algorithm runs in overall fewer cache misses if B is close in value to C . As B increases, the larger the rows of a submatrix can fit entirely into a page, decreasing cache misses overall. If the matrix is especially small, then it can fit entirely into one page. But as the matrix gets larger, the cache boundaries become more apparent and alignment issues crop up.

Therefore, it would be most beneficial to have a cache with B and C close in value.