Write up on Homework 1, Parallel Programming
by Chandan Kumar (201250808)

General notes:

- Test machine is a 4-core Intel Xeon W3550 @ 3.07 GHz with 8 MB L2 cache: http://ark.intel.com/products/39720/Intel-Xeon-Processor-W3550-(8M-Cache-3_06-GHz-4_80-GTs-Intel-QPI).
- Compiler used was Intel C/C++ Compiler (icc).
- All tests were done using "perf" performance tool on Linux.
- All tests were repeated for 5 times and average times were taken.
- All tests included initialization of the matrix, unless stated otherwise.
- All transpose used integers except for MKL tests.

1. In-place Matrix Transposition Problem

a. (see attached code transposeIP.c) In general, optimization level O3 gave better performance than O2, which gave better performance than O1 and which in turn gave better performance than no optimization (O0). O2 is the default optimization level for icc. Performance gain appeared to be mainly due to a drastic reduction in instructions (from 72 billion to 5 billion) and cycles (from 154b to 57b), and an overall reduction in cache misses (from 99% to 48%) even through L1 cache misses went up (from 2% to 121%). See figure 1, 2, 3, 4.

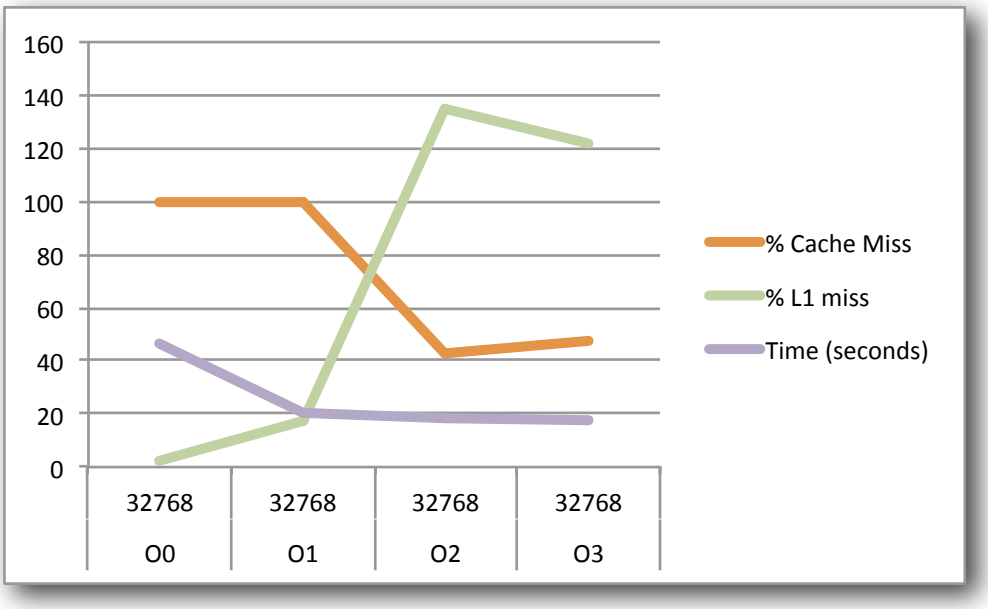

Figure 1: Performance variation of basic in-place transpose algorithm by varying level of optimization available in Intel C Compiler (matrix dimension 32768 x 32768)

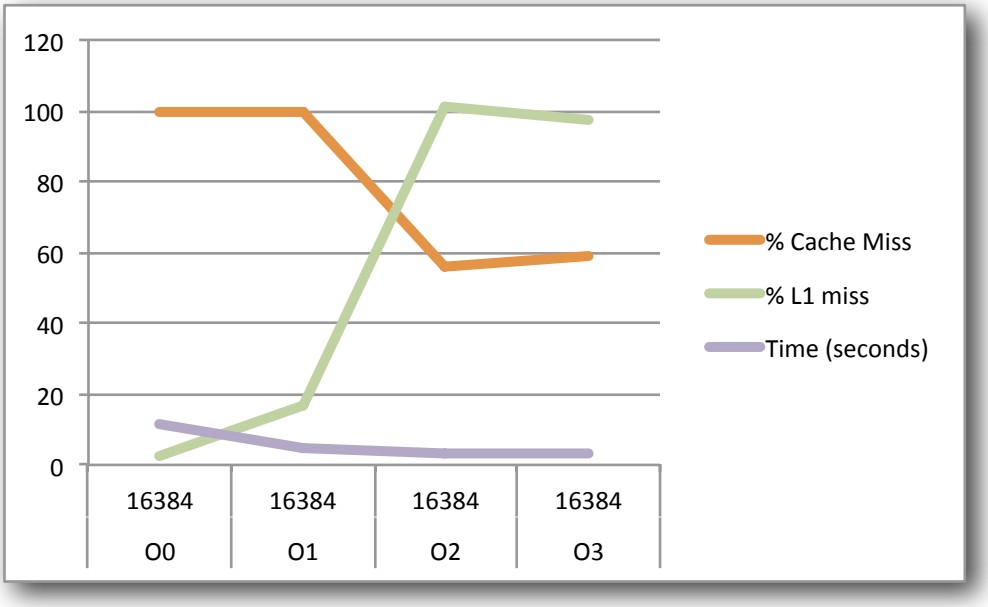

Figure 2: Performance variation of basic in-place transpose algorithm by varying level of optimization available in Intel C Compiler (matrix dimension 16384 x 16384)

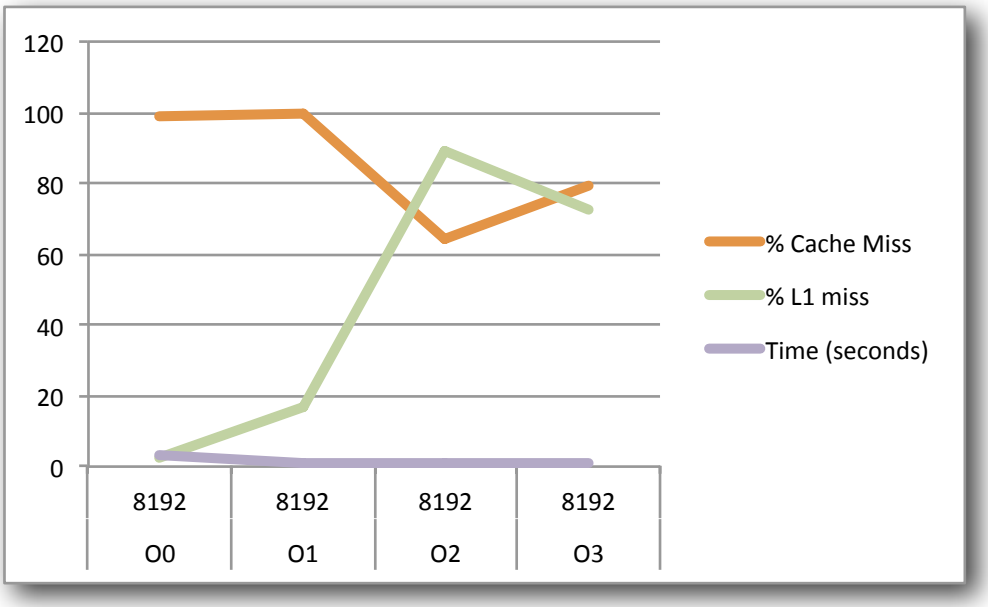

Figure 3: Performance variation of basic in-place transpose algorithm by varying level of optimization available in Intel C Compiler (matrix dimension 8192 x 8192)

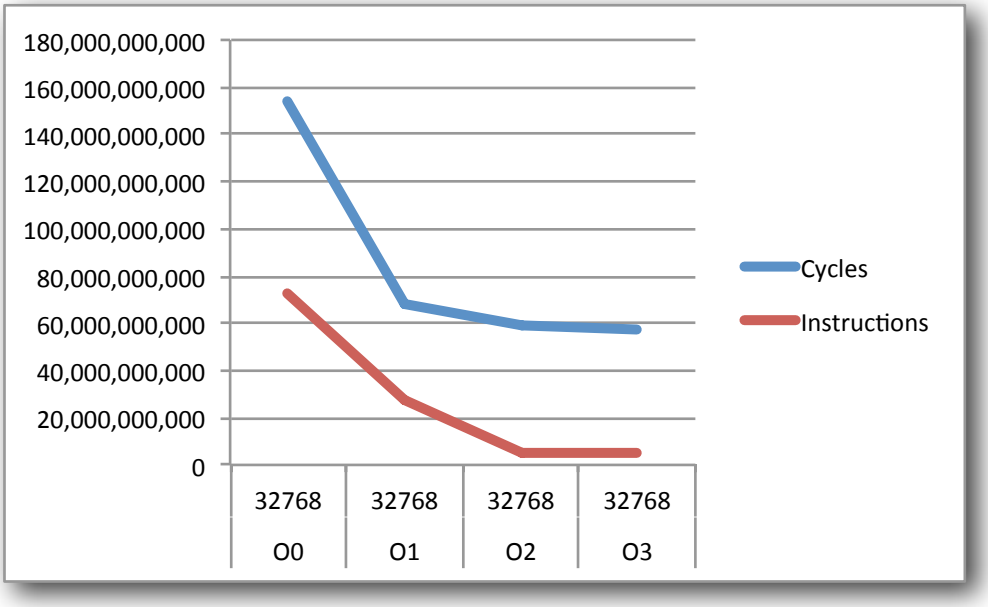

Figure 4: Performance gain by compiler optimization is partly credited to the drastic reduction in number of instructions (1/14th, probably due to vectorized instruction) and cycles (1/3rd).

b. (see attached code transposeIP.c)

i. (pending)

ii. Tile size of 8 seems to be optimal for all input sizes tried (n=1024, 4096, 8192, 16384 and 32768) with tile size 16 a distant second (see figure 5, 6). Tile size 8 also had the most optimal instructions per cycle. Number of instructions reduced with increasing tile size but number of cycles first decreased at tile size 8 and then increased. Cache loads went up with increasing tile size but cache misses went down, both as an absolute number and as percentage.

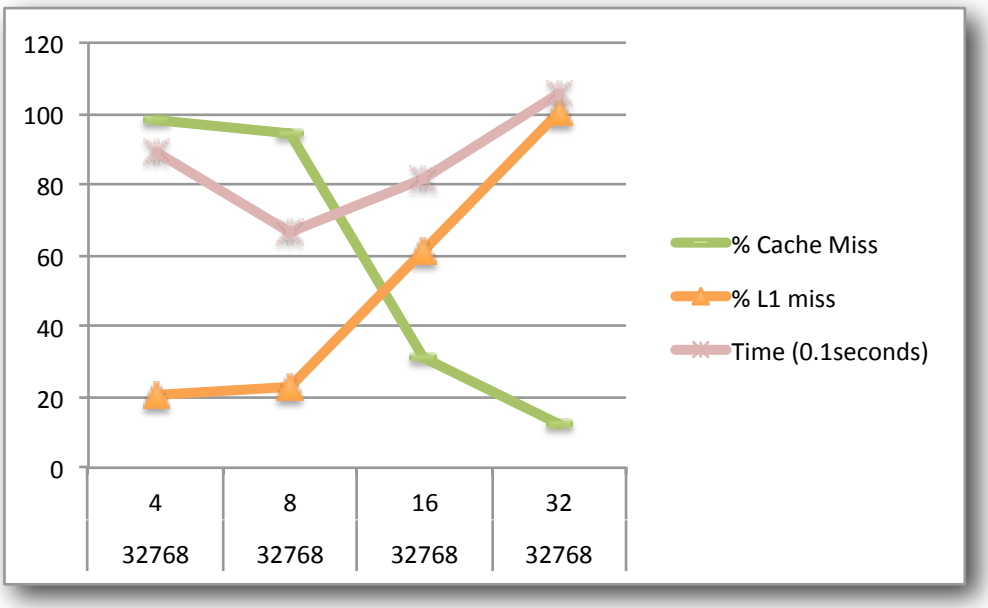

Figure 5: Performance seems to be best at tile size 8 for the 1-tiled in-place matrix transpose algorithm.

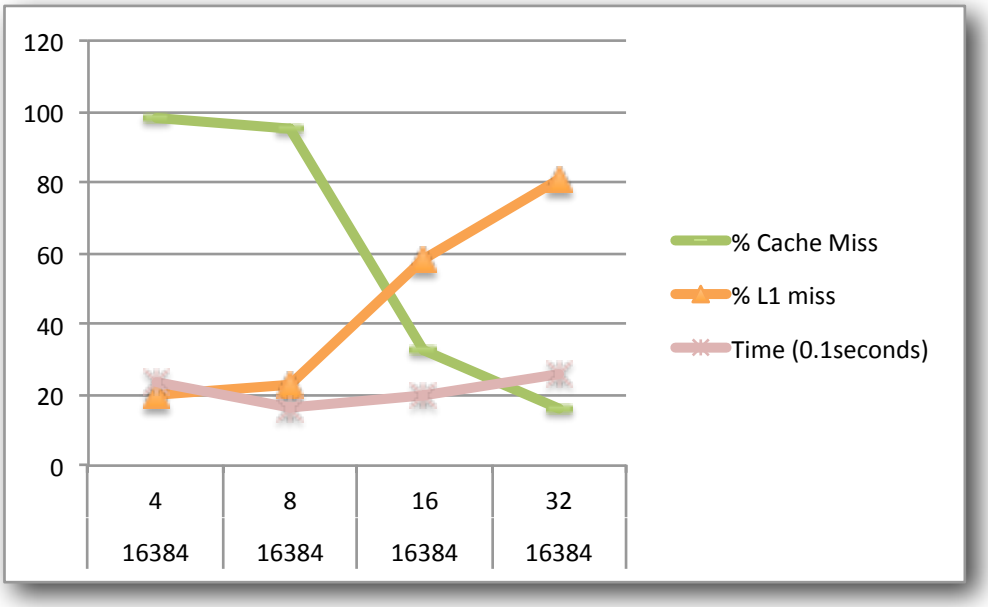

Figure 6: Performance seems to be best at tile size 8
for the 1-tiled in-place matrix transpose algorithm.

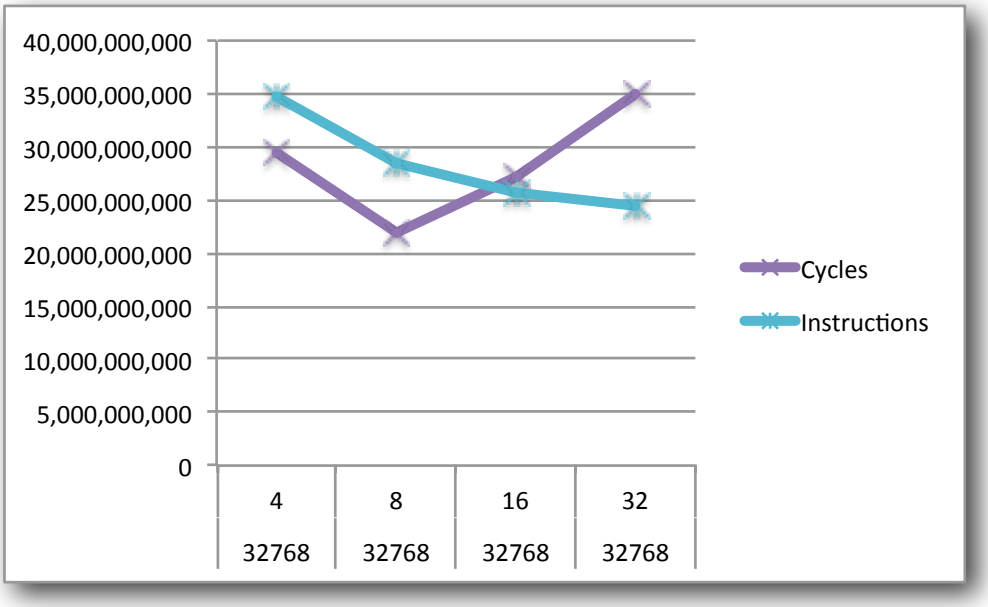

Figure 7: Instructions per Cycle is optimal for tile size 8
(1.32 versus 1.17, 1, 0.74 for tile sizes 4, 16, 32 respectively).

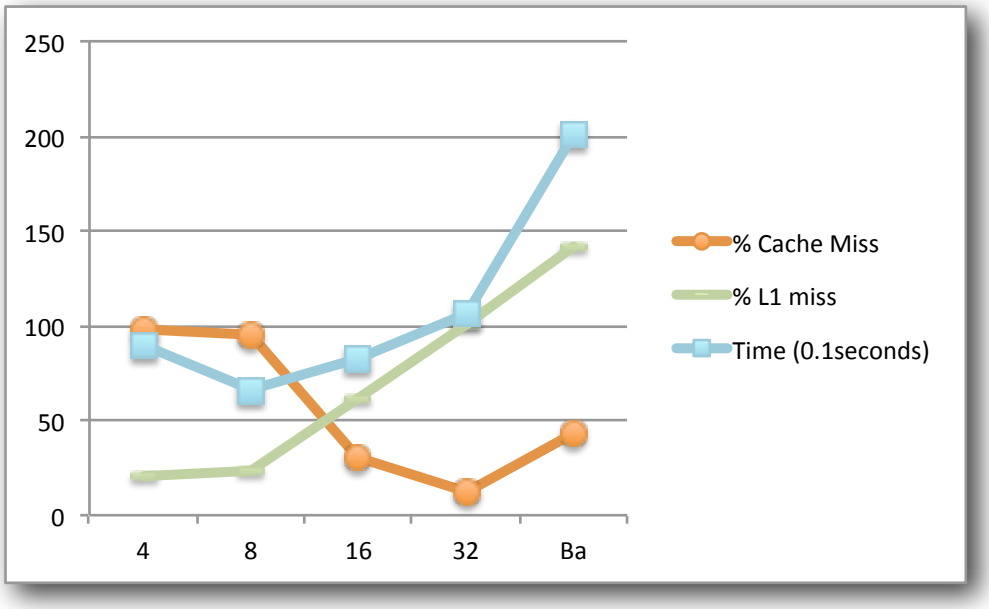

Figure 8: Comparison of 1-Tiled results to Basic.

iii. Four?
iv. The empirical search technique does not necessarily give the optimal tile size, as the search is done for only a small set of input.

c. (see attached code transposeIP.c) Optimal tile size seems to be (32,4) with (32,8) a close second. (16,4) and (16,8) are also close in performance to (32,4) and (32,8). See figure 8,9.

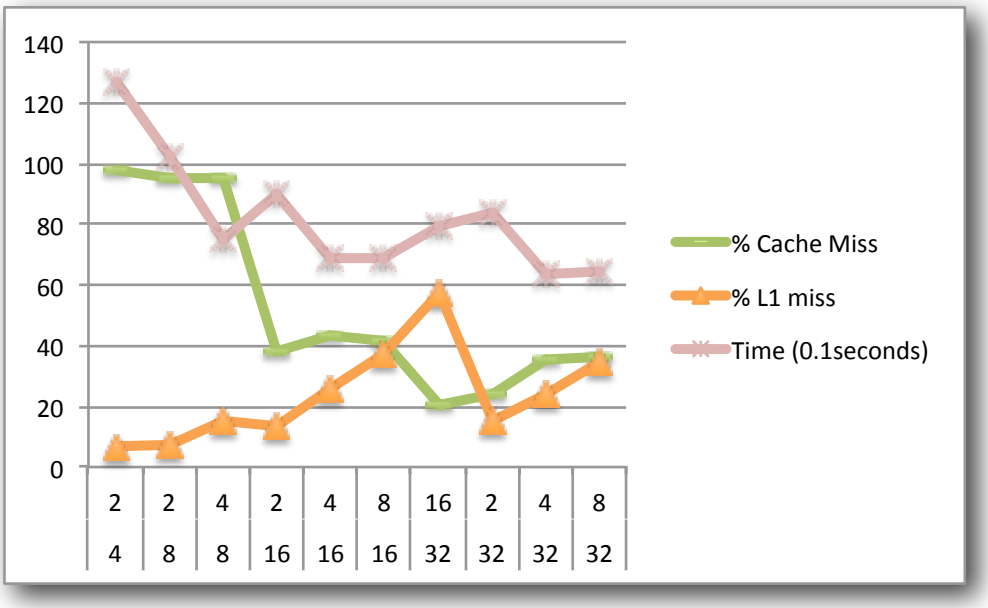

Figure 9: Tile size (32,4) is the best performing, with (32,8) performing equally good for a 32768 x 32768 matrix. (16,4) and (16,8) are also quite close.

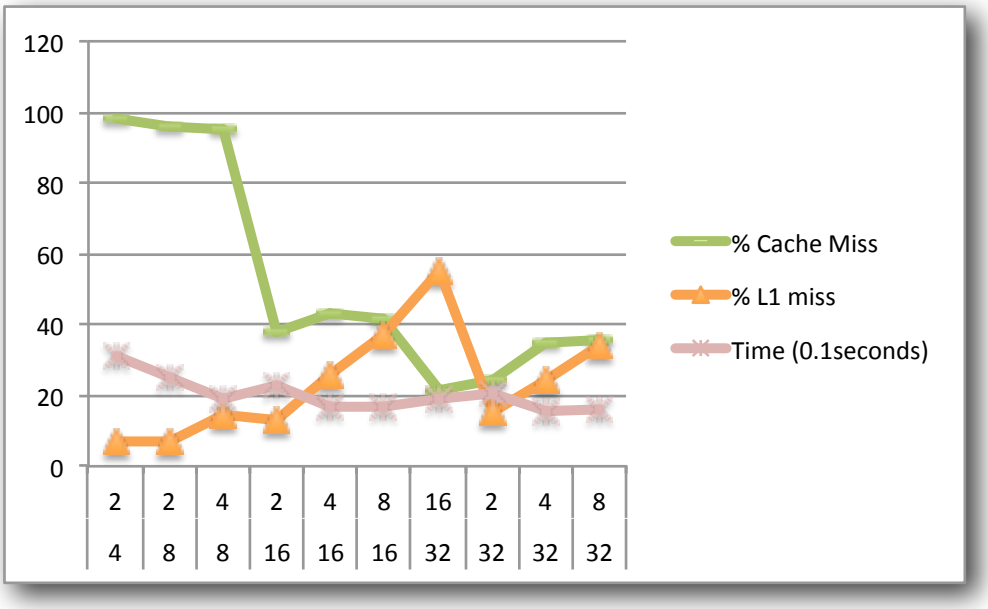

Figure 10: Tile size (32,4) is giving the best performance, which is a marginal improvement over tile size (32,8) for a 16384 x 16384 matrix. (16,4) and (16,8) are also quite close.

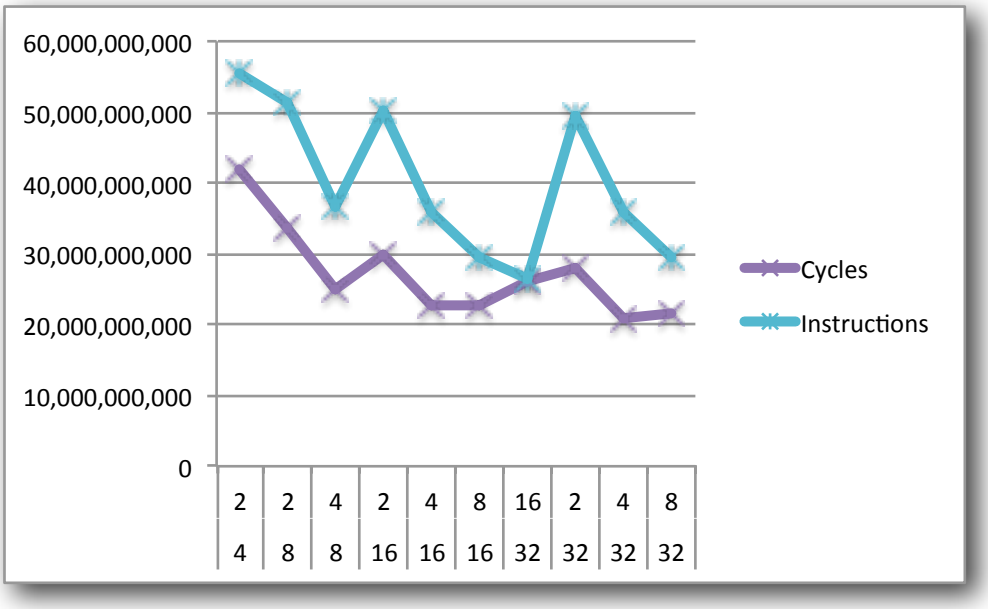

Figure 11: Cycles and Instructions count for matrix of size 32768.

d. (see attached code transposeIP.c) Cache oblivious algorithm's performance is better than the basic transpose, but not as good as 2-tiled or 1-tiled algorithms (see figure 12, 13).

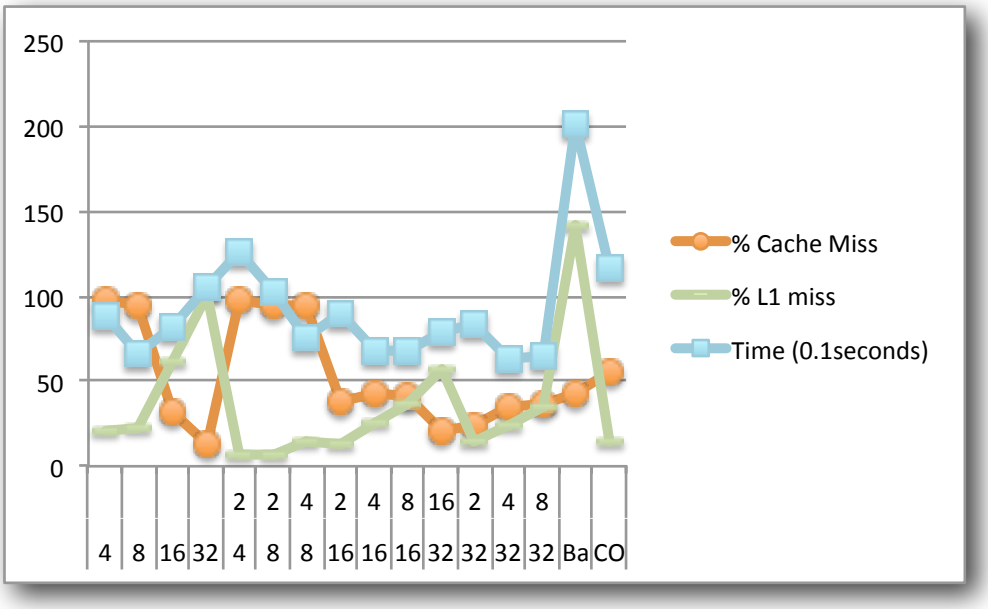

Figure 12: Comparative performance of 1-tiled, 2-tiled, basic and cache-oblivious algorithms

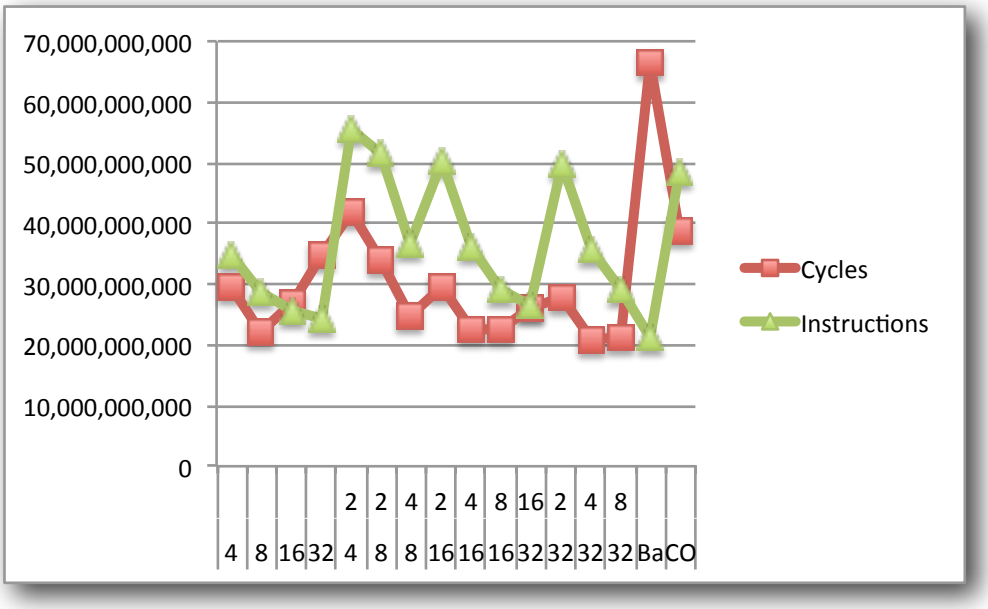

Figure 13: Comparative performance of 1-tiled, 2-tiled, basic and cache-oblivious algorithms

e. (see attached code floattransposeIP.c). All charts below are produced based on data obtained from float matrices. MKL took the longest time, apparently due to highest L1 miss rate which led to poor instructions per cycle value.

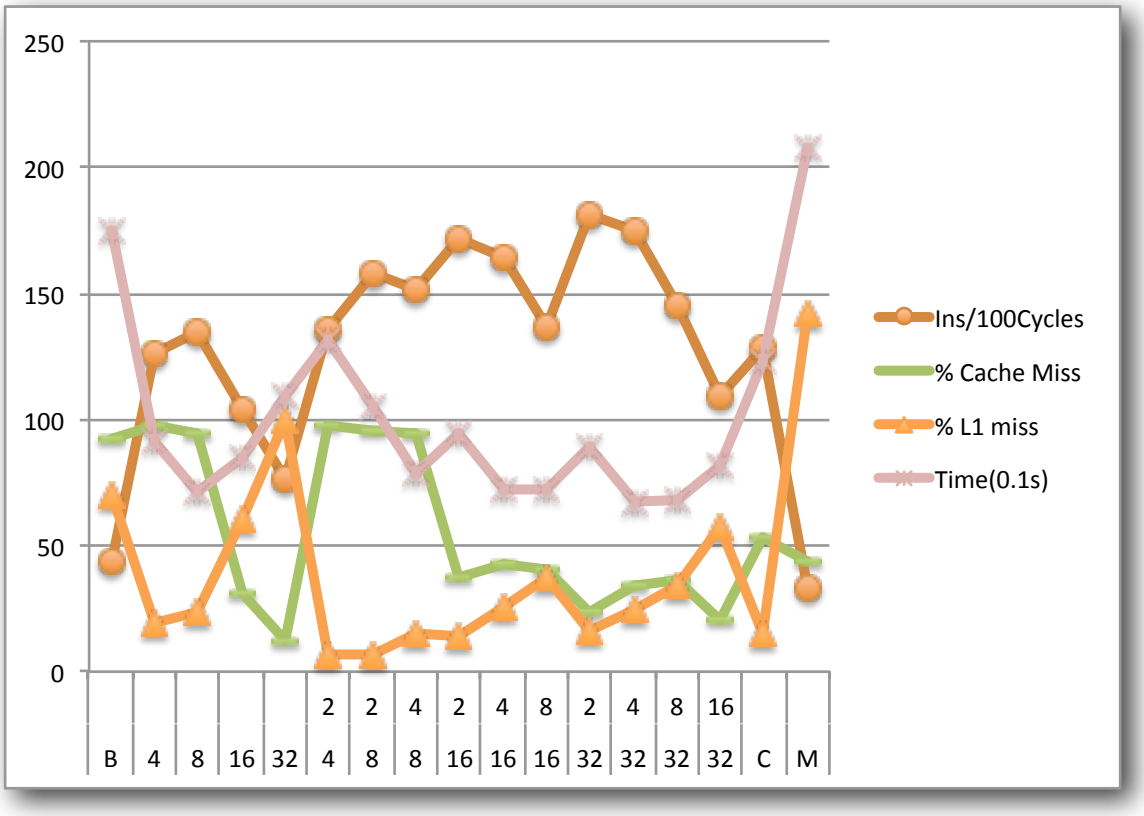

Figure 14: Performance of all five algorithms for 32768 sized matrix
(B=Basic, C=Cache Oblivious, M=MKL)

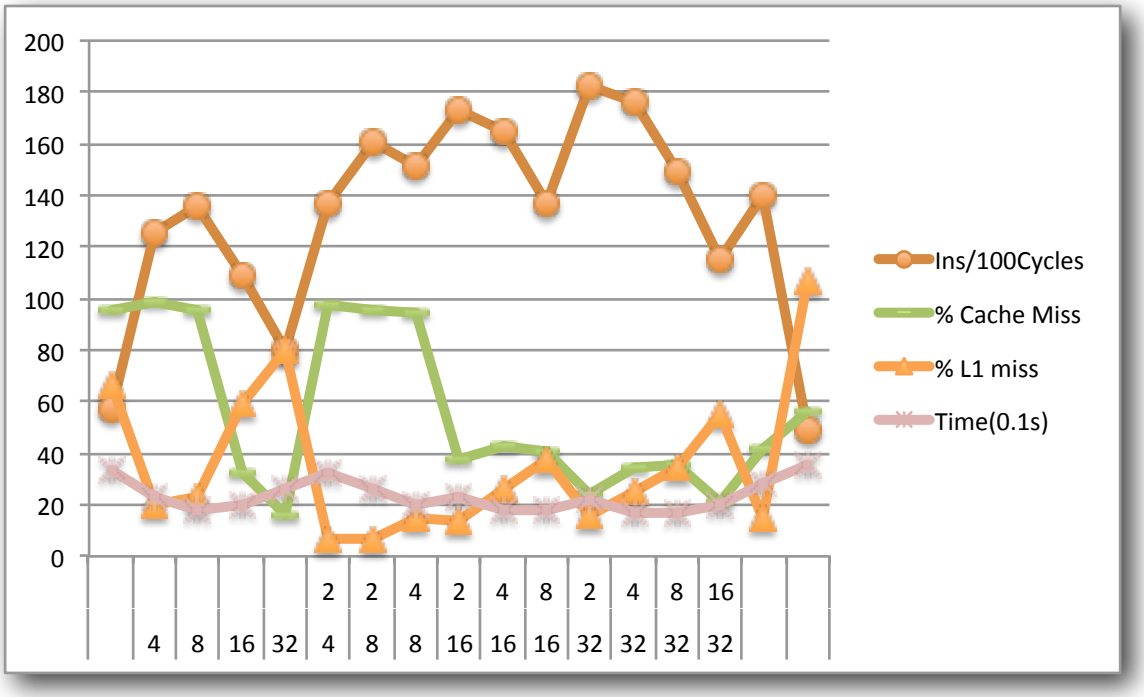

Figure 15: Performance of all five algorithms for 16384 sized matrix
(B=Basic, C=Cache Oblivious, M=MKL)

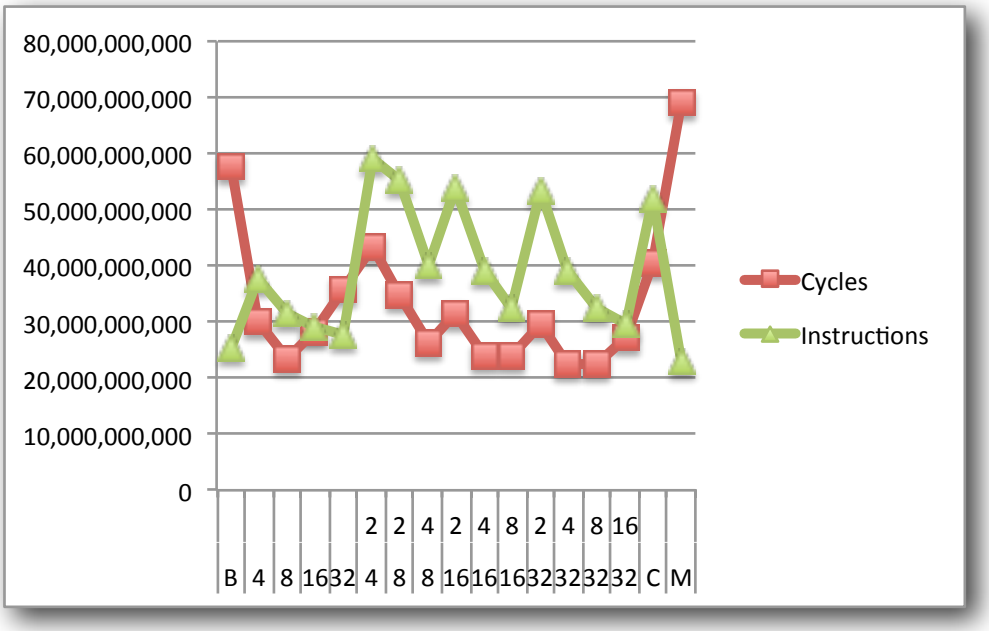

Figure 16: CPU cycles executed and Instruction count of all 5 algorithms for 32768-sized matrix

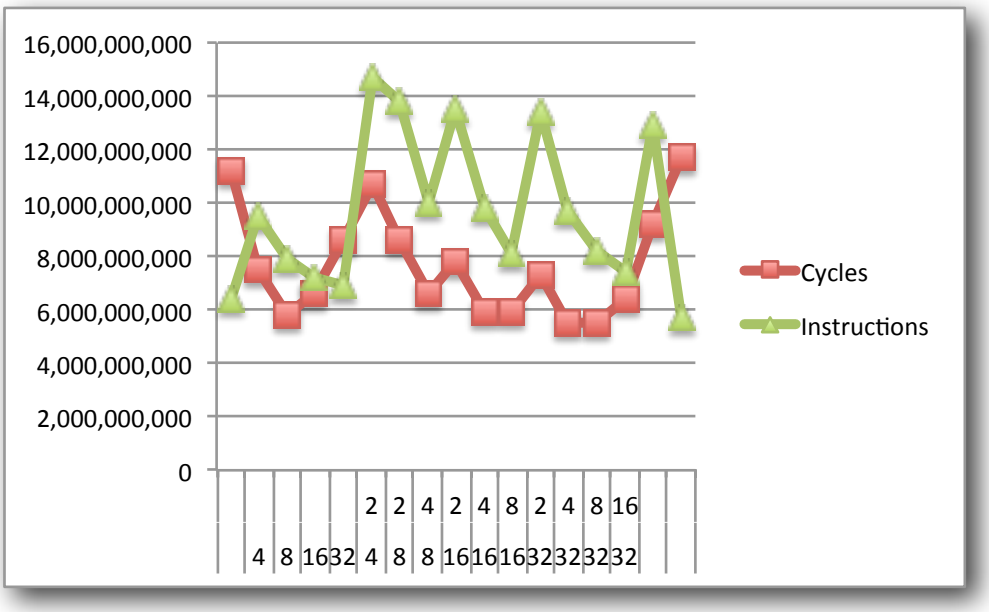

Figure 17: CPU cycles executed and Instruction count of all 5 algorithms for 16784-sized matrix

2. **Out-of-place Matrix Transposition Problem**
   a. (see attached code transposeOP.c) None of the optimization options seem to make any significant performance improvement which is very odd. It probably means that the basic unoptimized algorithm is well-supported by hardware prefetching. O2 appears to give a little better performance. See figures 14, 15 and 16.

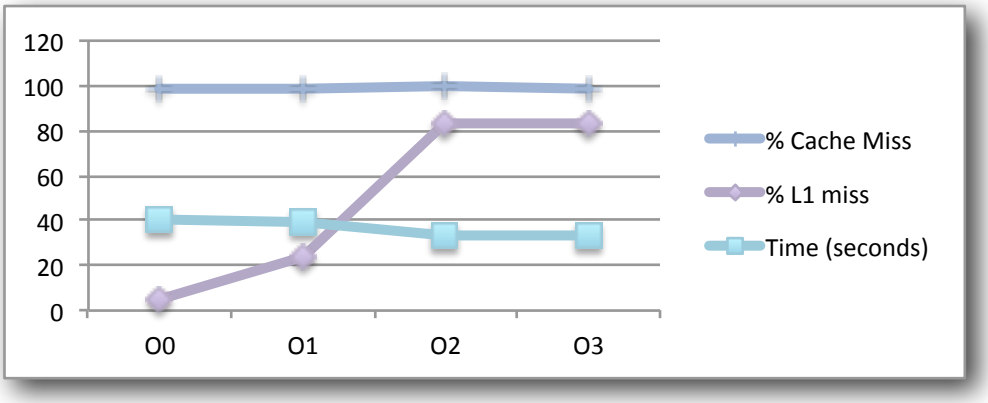

Figure 18: Performance using compiler optimization options for matrix of dimension 32768

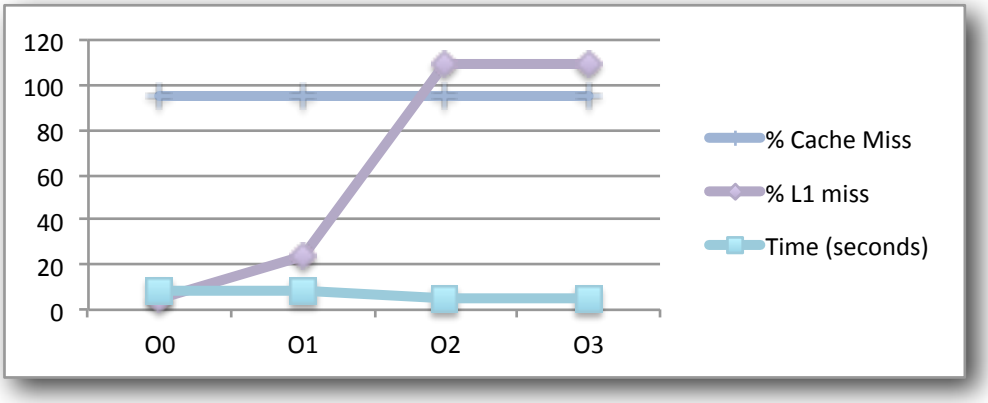

Figure 19: Performance using compiler optimization options for matrix of dimension 16384

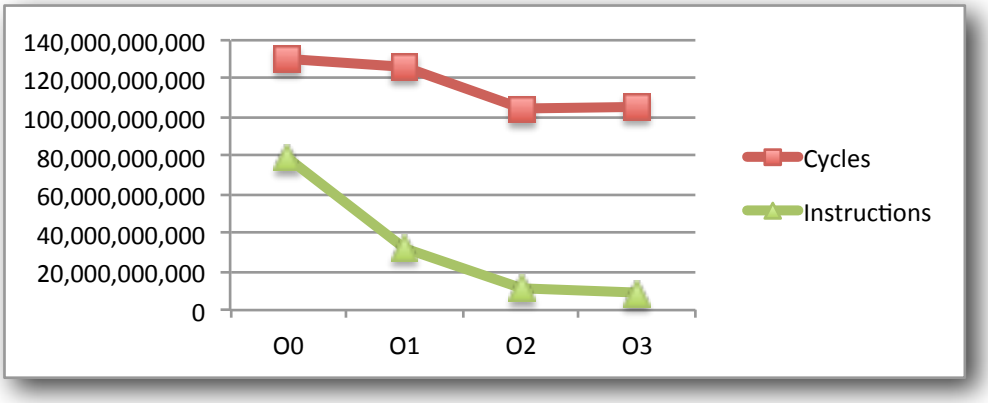

Figure 20: Cycle and Instruction count using compiler

optimizations for matrix of dimension 32768

b. (see attached code transposeOP.c)
   i. (pending)
   ii. Tile size of 8 seems to be optimal for all input sizes tried (n=1024, 4096, 8192, 16384 and 32768) with tile size 4 a distant second.

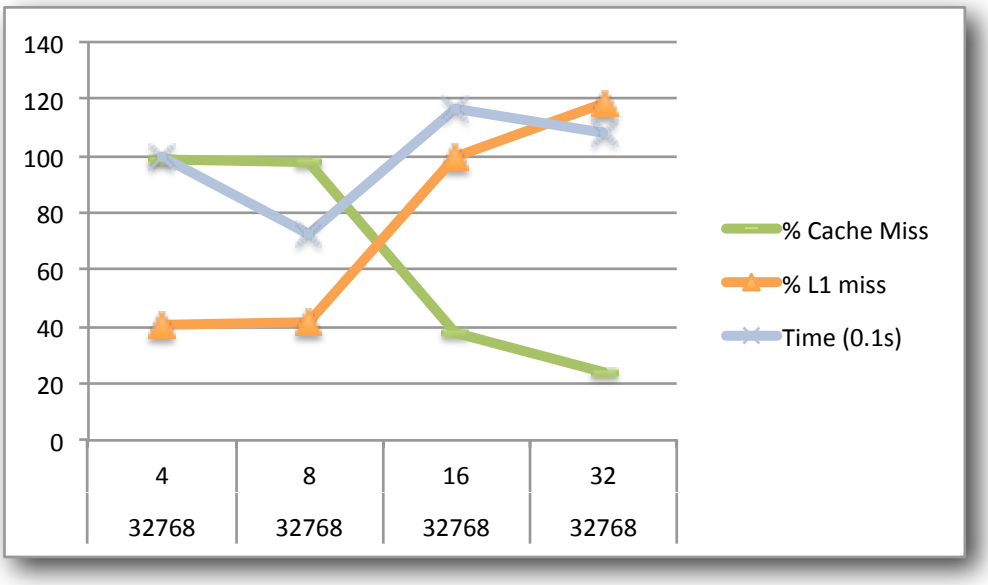

Figure 21: Tile size 8 appears to be optimal for 32768 x 32768 matrix.

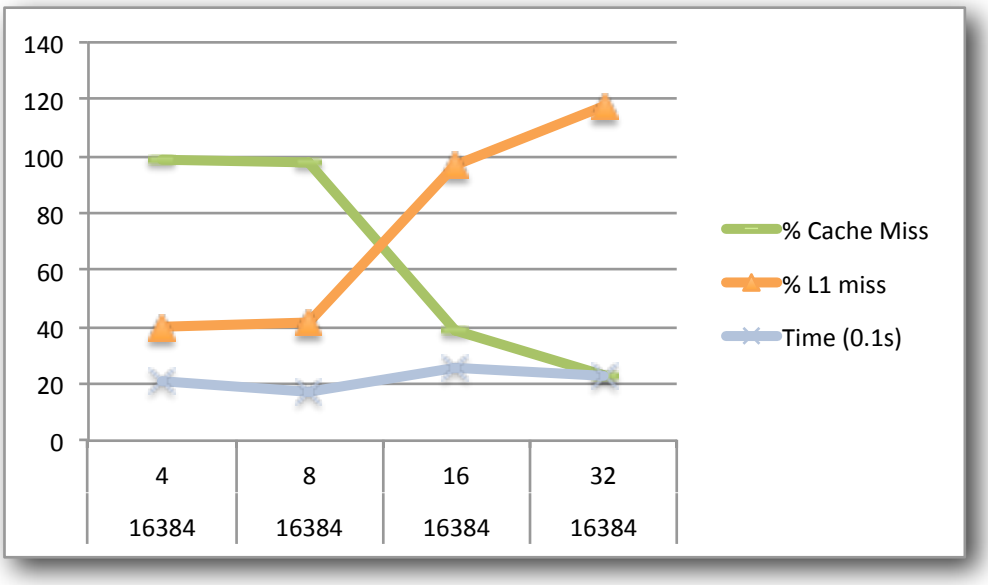

Figure 22: Tile size 8 appears to be optimal for 16384 x 16384 matrix.

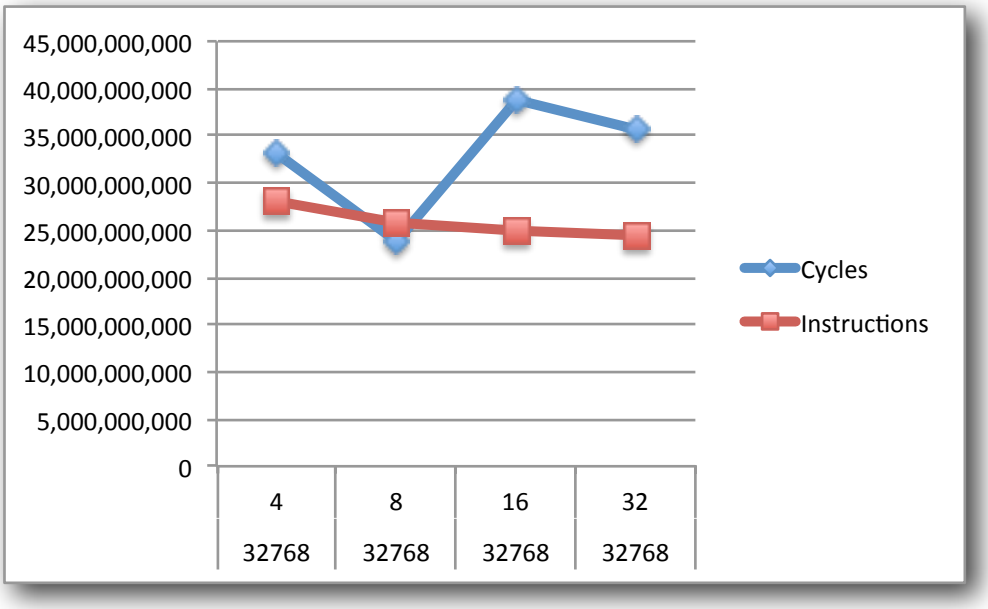

Figure 23: Cycles and Instructions count for different tile sizes

iii. Four?

iv. The empirical search technique does not necessarily give the optimal tile size, as the search is done for only a small set of input.

c. (see attached code transposeOP.c) Optimal tile size seems to be (32,8).

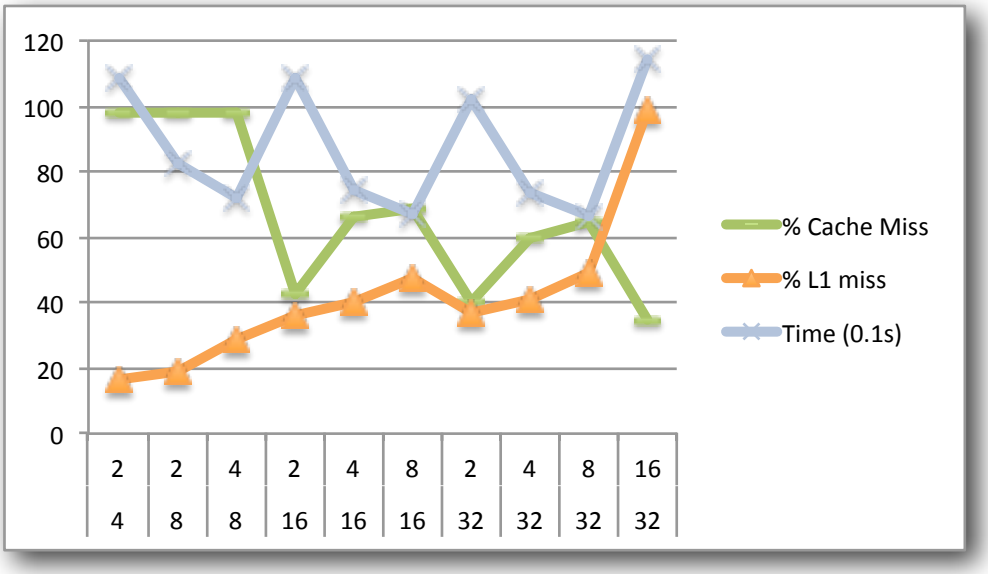

Figure 24: Tile size 32,8 and 16,8 seem optimal for
2-tiled matrix transpose (dimension 32768)

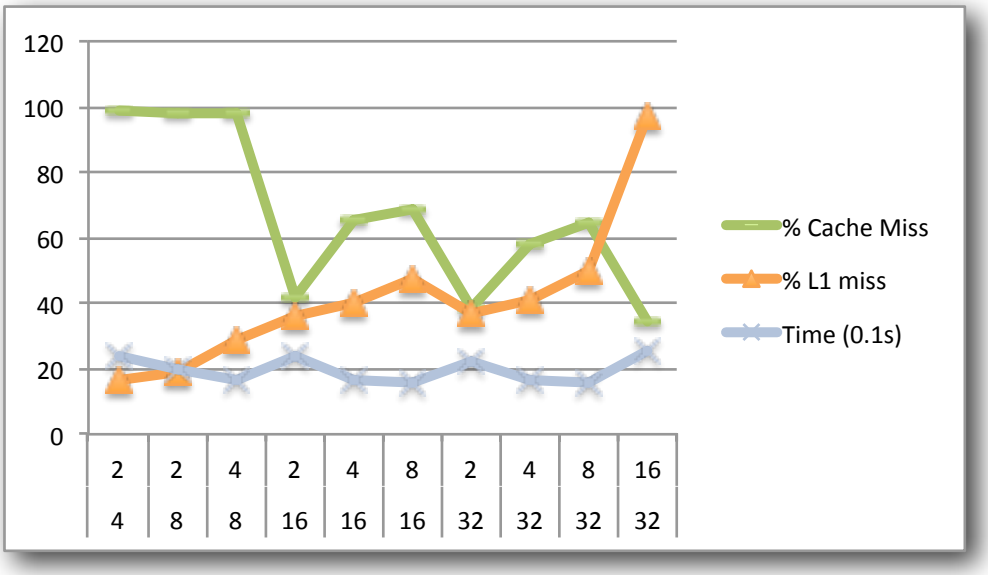

Figure 25: Tile size 32,8 and 16,8 seem optimal for 2-tiled matrix transpose (dimension 16384)

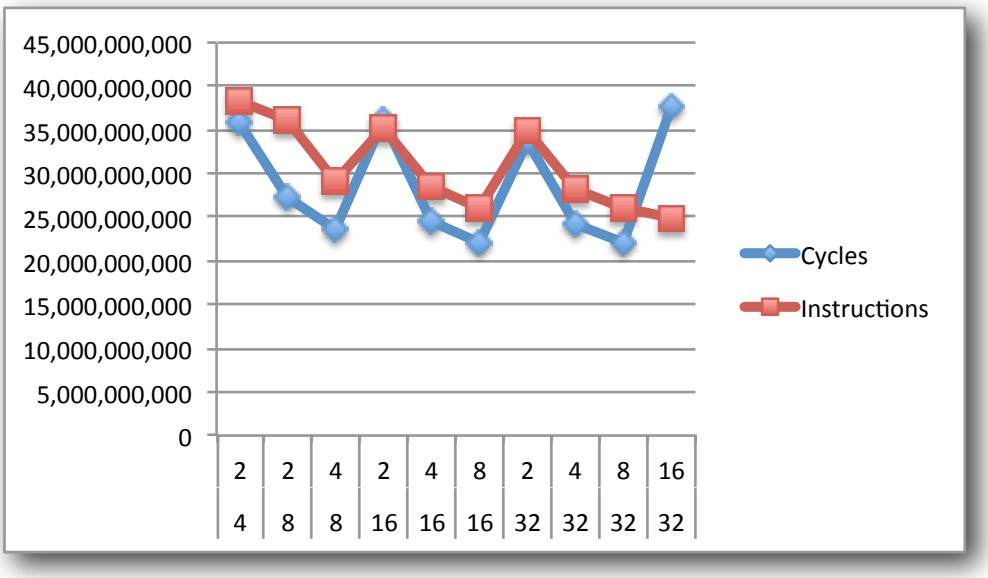

Figure 26: Cycles and Instructions count for 2-tiled matrix transpose (dimension 32768)

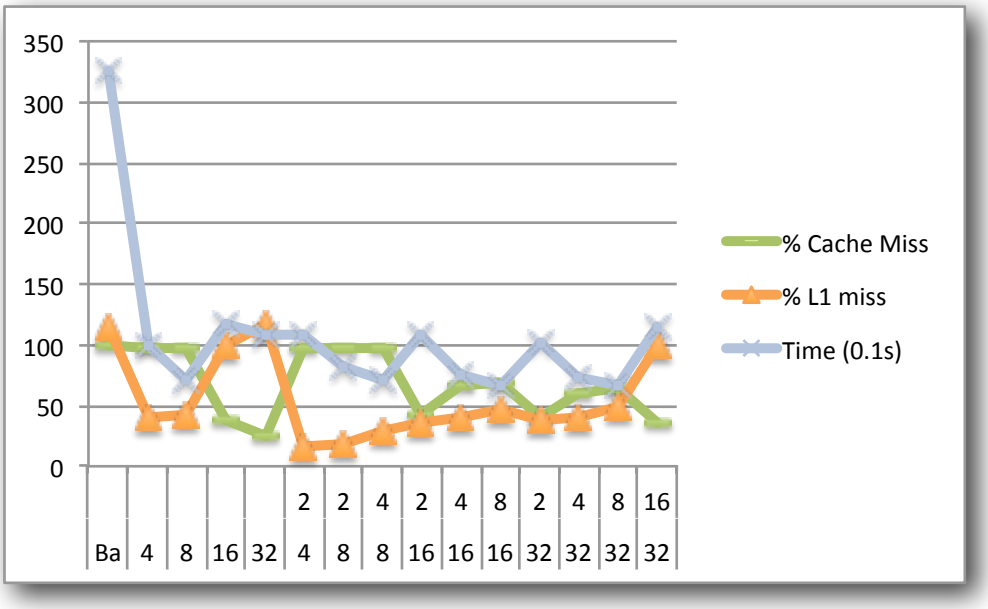

Figure 27: Comparative performance of Basic, 1-tiled and 2-tiled out-of-place matrix transpose (dimension 32768)

d. (see attached code transposeOP.c)
e. (see attached code floattransposeOP.c) All charts below are produced based on data obtained from float matrices. MKL implementation seems to be the fastest. Basic algorithm is the slowest.

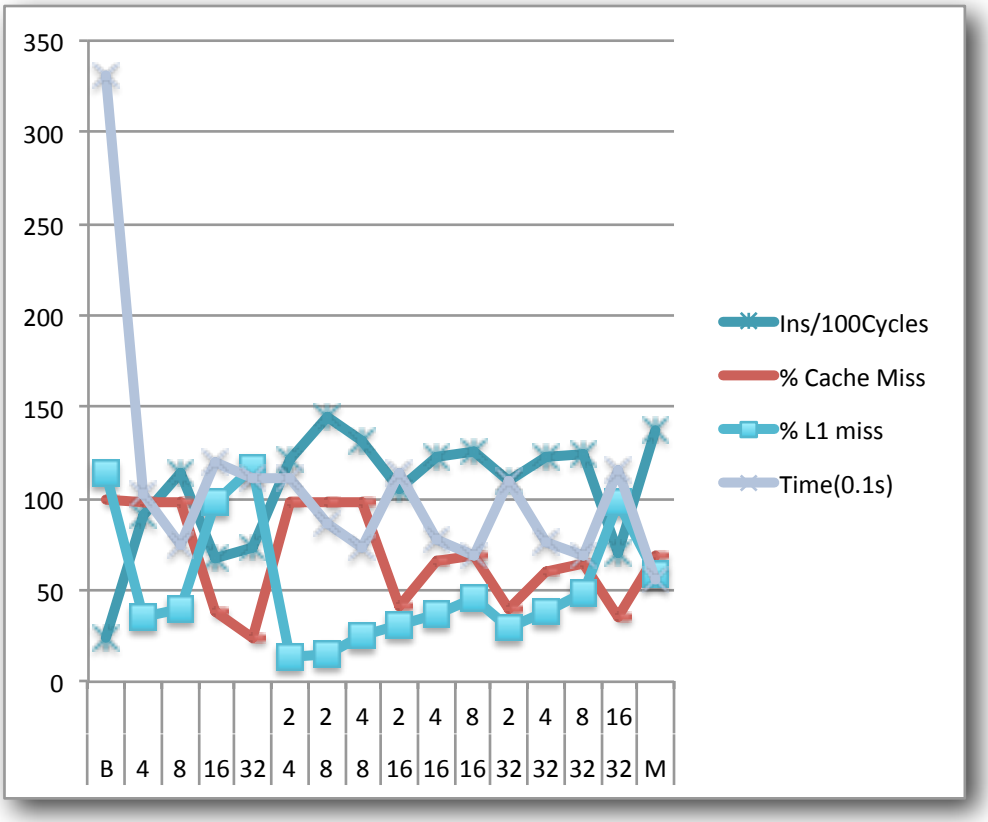

Figure 28: Comparative performance of all 5 algorithms for 32768-sized matrix

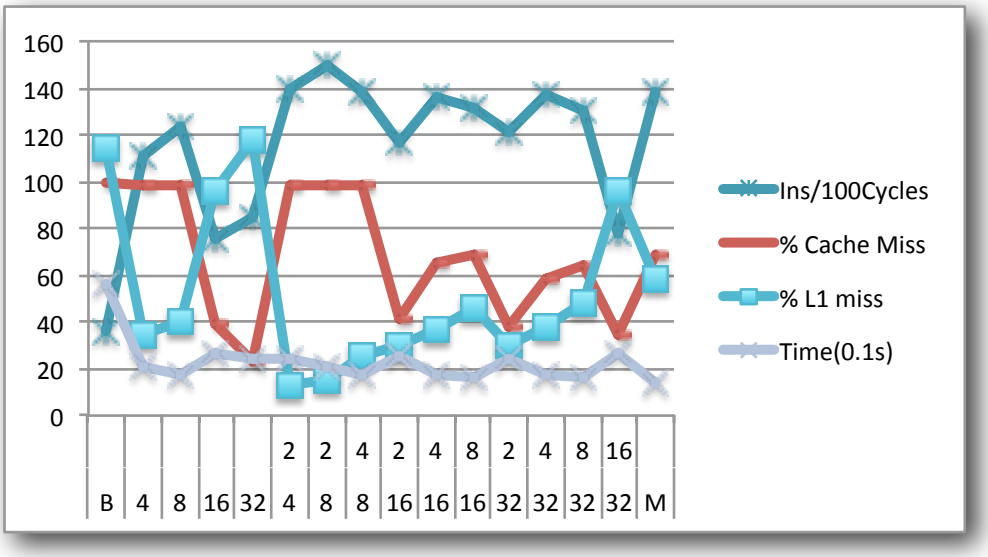

Figure 28: Comparative performance of all 5 algorithms for 16384-sized matrix

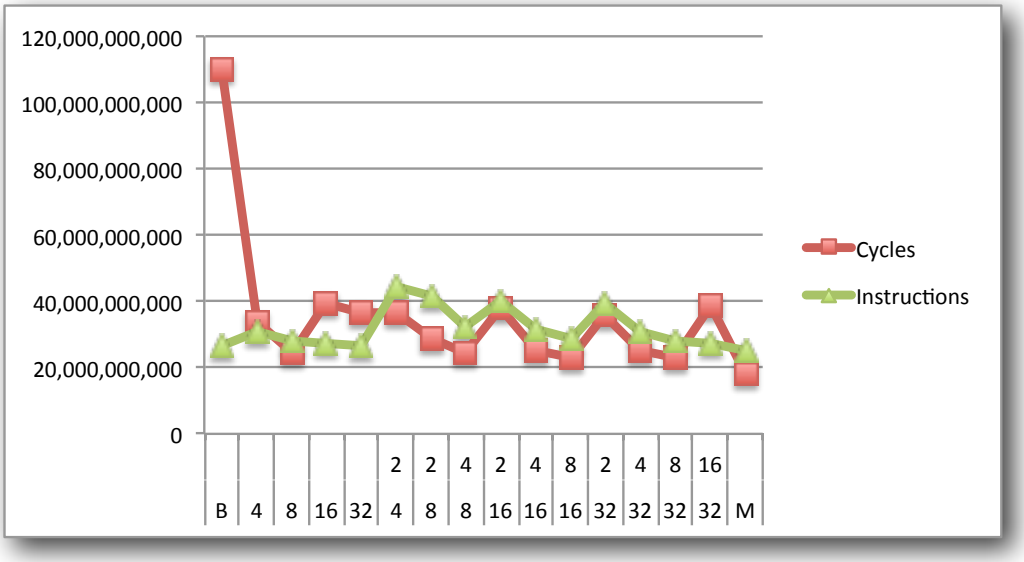

Figure 29: Cycles consumed and Instructions executed of all 5 algorithms for 32768-sized matrix.

3. **Prefix Sum Problem**
   a. (see attached code prefixSum.c) Time does not include initialization time.
   b. The algorithm is cache oblivious. But performance is not as good as basic prefix sum algorithm.

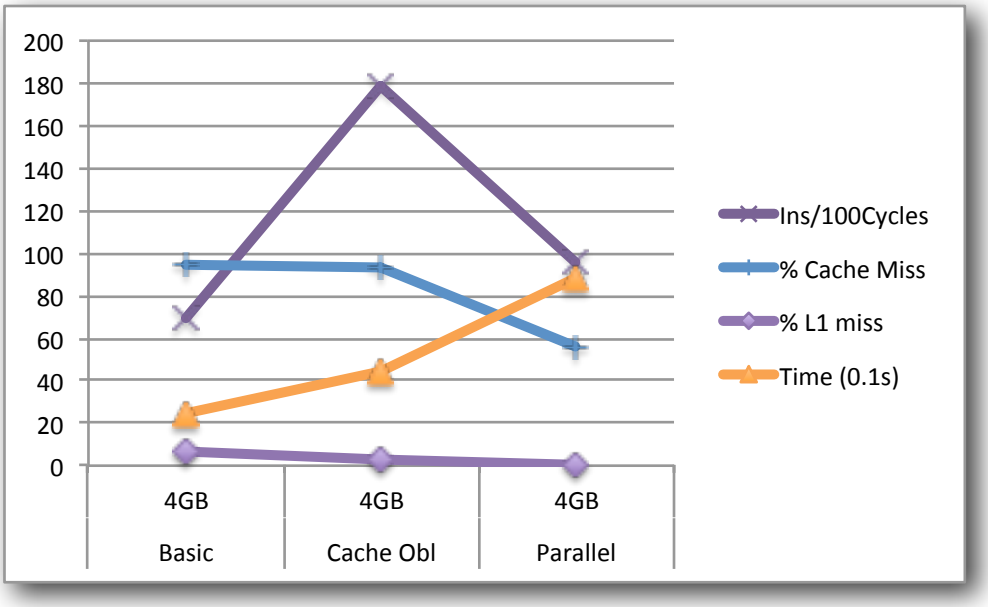

Figure 30: Comparison of various parameters for basic, cache oblivious and parallel algorithms for list of size 4GB

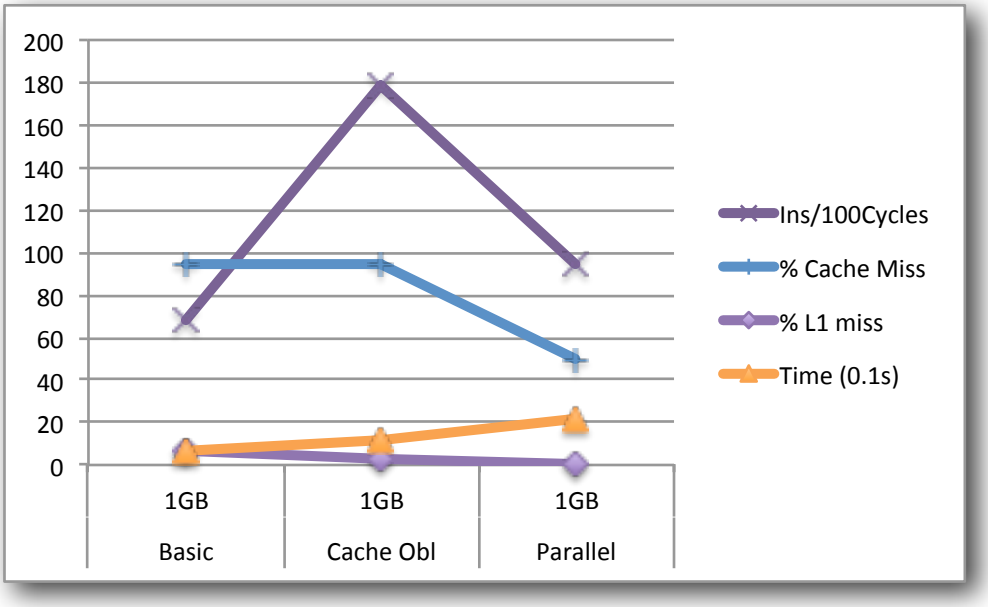

Figure 31: Comparison of various parameters for basic, cache oblivious and parallel algorithms for list of size 4GB

   c. (pending)
   d. (pending)
   e. Challenge problem: (see attached code prefixSum.c)
   f. Extra: Parallel version using Cilk (see attached code prefixSumCilk.c)