Hello,
I am working on the CS:APP cachelab. In a previous conversation, I was able to work with you and solve the three different cases (one solution for 32x32, one solution for 64x64, one solution for 61x67). However, there seemed to be an issue. In our previous conversation, you opted to handle all three cases in one function and use an if statement to check the different cases. In addition, the same solution for 32x32 seemed to get a different number of cache misses depending on what code it was with.

I was wondering if we could remedy some of these issues. I was hoping that you could do the following:
- Given a function that has three cases that provide the correct solution *sometimes*, I was hoping that you could analyze it for any possible issues and hypothesize why your old code may not have worked.
- Using the three cases, create a new trans.c file that contains all of the old information from the old file but instead creates three separate functions, upon which only one is called based on the dimensions.
- Analyze why this code sometimes produces "zero misses" (an impossible result) or "invalid" when running the tester.

Below is the pasted code. As a reminder, please ensure that you are obeying the rules of the cache lab (no more than 12 local variables, etc.). If it would help you before you begin, I can also reattach the assignment PDF.

```
/*
 * trans.c - Matrix transpose B = A^T
 *
 * Each transpose function must have a prototype of the
form:
 * void trans(int M, int N, int A[N][M], int B[M][N]);
 *
 * A transpose function is evaluated by counting the
number of misses
 * on a 1KB direct mapped cache with a block size of 32
bytes.
 */
#include <stdio.h>
```

```
#include "cachelab.h"

int is_transpose(int M, int N, int A[N][M], int B[M][N]);

/*
 * transpose_submit - This is the solution transpose
function that you
 *     will be graded on for Part B of the assignment. Do not
change
 *     the description string "Transpose submission", as the
driver
 *     searches for that string to identify the transpose
function to
 *     be graded.
 */
char transpose_submit_desc[] = "Transpose submission";
void transpose_submit(int M, int N, int A[N][M], int B[M][N])
{
    int i, j, ii, jj;
    int t0, t1, t2, t3, t4, t5, t6, t7;

    /* ---------------- 32 x 32 ---------------- */
    if (M == 32 && N == 32) {
        int block = 8;
        for (ii = 0; ii < N; ii += block) {
            for (jj = 0; jj < M; jj += block) {
                for (i = ii; i < ii + block; ++i) {
                    /* keep diagonal in temp and write after the
inner loop */
                    for (j = jj; j < jj + block; ++j) {
                        if (i != j) {
                            B[j][i] = A[i][j];
                        } else {
                            /* store diagonal temporarily then write
after */
                            t0 = A[i][j];
                            B[j][i] = t0;
                        }
                    }
                }
            }
        }
        return;
    }

    /* ---------------- 64 x 64 ---------------- */
    if (M == 64 && N == 64) {
```

```
for (ii = 0; ii < 64; ii += 8) {
  for (jj = 0; jj < 64; jj += 8) {

    for (i = 0; i < 4; ++i) {
      t0 = A[ii + i][jj + 0];
      t1 = A[ii + i][jj + 1];
      t2 = A[ii + i][jj + 2];
      t3 = A[ii + i][jj + 3];
      t4 = A[ii + i][jj + 4];
      t5 = A[ii + i][jj + 5];
      t6 = A[ii + i][jj + 6];
      t7 = A[ii + i][jj + 7];

      B[jj + 0][ii + i] = t0;
      B[jj + 1][ii + i] = t1;
      B[jj + 2][ii + i] = t2;
      B[jj + 3][ii + i] = t3;

      B[jj + 0][ii + 4 + i] = t4;
      B[jj + 1][ii + 4 + i] = t5;
      B[jj + 2][ii + 4 + i] = t6;
      B[jj + 3][ii + 4 + i] = t7;
    }

    for (j = 0; j < 4; ++j) {
      t0 = A[ii + 4][jj + j];
      t1 = A[ii + 5][jj + j];
      t2 = A[ii + 6][jj + j];
      t3 = A[ii + 7][jj + j];

      t4 = B[jj + j][ii + 4];
      t5 = B[jj + j][ii + 5];
      t6 = B[jj + j][ii + 6];
      t7 = B[jj + j][ii + 7];

      B[jj + j][ii + 4] = t0;
      B[jj + j][ii + 5] = t1;
      B[jj + j][ii + 6] = t2;
      B[jj + j][ii + 7] = t3;

      B[jj + 4 + j][ii + 0] = t4;
      B[jj + 4 + j][ii + 1] = t5;
      B[jj + 4 + j][ii + 2] = t6;
      B[jj + 4 + j][ii + 3] = t7;
    }

    for (i = 4; i < 8; ++i) {
```

```
                    t0 = A[ii + i][jj + 4];
                    t1 = A[ii + i][jj + 5];
                    t2 = A[ii + i][jj + 6];
                    t3 = A[ii + i][jj + 7];

                    B[jj + 4][ii + i] = t0;
                    B[jj + 5][ii + i] = t1;
                    B[jj + 6][ii + i] = t2;
                    B[jj + 7][ii + i] = t3;
                }
            }
        }
        return;
    }

    /* ---------------- 61 x 67 (original working version)
---------------- */
    {
        int block = 16;
        for (ii = 0; ii < N; ii += block) {
            for (jj = 0; jj < M; jj += block) {
                int imax = (ii + block < N) ? ii + block : N;
                int jmax = (jj + block < M) ? jj + block : M;

                for (i = ii; i < imax; i++) {
                    for (j = jj; j < jmax; j++) {
                        B[j][i] = A[i][j];
                    }
                }
            }
        }
        return;
    }
}

/*
 * You can define additional transpose functions below.
We've defined
 * a simple one below to help you get started.
 */

/*
 * trans - A simple baseline transpose function, not
optimized for the cache.
 */
char trans_desc[] = "Simple row-wise scan transpose";
void trans(int M, int N, int A[N][M], int B[M][N])
```

```
{
    int i, j, tmp;

    for (i = 0; i < N; i++) {
        for (j = 0; j < M; j++) {
            tmp = A[i][j];
            B[j][i] = tmp;
        }
    }

}

/*
 * registerFunctions - This function registers your
transpose
 *     functions with the driver.  At runtime, the driver will
 *     evaluate each of the registered functions and
summarize their
 *     performance. This is a handy way to experiment with
different
 *     transpose strategies.
 */
void registerFunctions()
{
    /* Register your solution function */
    registerTransFunction(transpose_submit,
transpose_submit_desc);

    /* Register any additional transpose functions */
    registerTransFunction(trans, trans_desc);

}

/*
 * is_transpose - This helper function checks if B is the
transpose of
 *     A. You can check the correctness of your transpose
by calling
 *     it before returning from the transpose function.
 */
int is_transpose(int M, int N, int A[N][M], int B[M][N])
{
    int i, j;

    for (i = 0; i < N; i++) {
        for (j = 0; j < M; ++j) {
            if (A[i][j] != B[j][i]) {
```

```
                        return 0;
                    }
                }
            }
            return 1;
        }
```

Thought for 14s

Nice — this is a great, detailed prompt. I reviewed your pasted `trans.c` and wrote a cleaned-up version that:

- Splits the three special-case strategies into three separate functions (`trans_32` , `trans_64` , `trans_61x67`),
- Keeps `transpose_submit` as the dispatcher (the driver looks for the description string so we keep `"Transpose submission"`),
- Leaves the simple `trans` baseline function,
- Preserves the intended optimizations while keeping local-variable counts within the lab constraints,
- Adds a couple of defensive checks / comments and suggestions for debugging.

Below you'll find the full `trans.c` file (drop it into your lab directory, compile normally). After the code I explain the likely causes of the weird behavior you saw (different miss counts depending on surrounding code, impossible “zero misses”, and occasional `invalid`), plus debugging advice.

---

c   Copy code

```c
/*
 * trans.c - Matrix transpose B = A^T
 *
 * Each transpose function must have a prototype of the form:
 * void trans(int M, int N, int A[N][M], int B[M][N]);
 *
 * A transpose function is evaluated by counting the number of
misses
 * on a 1KB direct mapped cache with a block size of 32 bytes.
 */
#include <stdio.h>
#include "cachelab.h"

int is_transpose(int M, int N, int A[N][M], int B[M][N]);
```

```
/* forward declarations for the separate-case helpers */
void trans_32(int M, int N, int A[N][M], int B[M][N]);
void trans_64(int M, int N, int A[N][M], int B[M][N]);
void trans_61x67(int M, int N, int A[N][M], int B[M][N]);

/*
 * transpose_submit - Dispatcher (keeps description string for
driver).
 */
char transpose_submit_desc[] = "Transpose submission";
void transpose_submit(int M, int N, int A[N][M], int B[M][N])
{
    /* dispatch to specialized functions so each function is small
and stable */
    if (M == 32 && N == 32) {
        trans_32(M, N, A, B);
        return;
    }
    if (M == 64 && N == 64) {
        trans_64(M, N, A, B);
        return;
    }
    /* default / given case in the assignment */
    if (M == 61 && N == 67) {
        trans_61x67(M, N, A, B);
        return;
    }

    /* fallback (generic) */
    trans_61x67(M, N, A, B);
}

/* ------------------- 32x32 implementation -------------------
   Typical 8x8 blocking; keep diagonals out-of-place until end of
block
   to avoid repeatedly evicting the diagonal element from cache.
   Local var count kept small.
*/
void trans_32(int M, int N, int A[N][M], int B[M][N])
{
    int ii, jj, i, j;
    int block = 8;
    int tmp;      /* store diagonal element temporarily */
    int diag = -1;

    for (ii = 0; ii < N; ii += block) {
        for (jj = 0; jj < M; jj += block) {
            for (i = ii; i < ii + block; ++i) {
                /* careful: when block along diagonal, we save
diagonal element
                   and write it after the inner loop so we don't
thrash the cache. */
                for (j = jj; j < jj + block; ++j) {
                    if (i == j) {
                        tmp = A[i][j];
                        diag = i;   /* remember diagonal index */
                    } else {
                        B[j][i] = A[i][j];
                    }
                }
                if (diag == i) {
                    B[i][i] = tmp;
                    diag = -1;
                }
            }
        }
    }
}

/* ------------------- 64x64 implementation -------------------
   Carefully tuned variant for 8x8 sub-blocks that avoids conflict
misses
   on a 1KB direct-mapped, 32-byte blocks cache (classic solution).
   Keep variable count within limits.
*/
void trans_64(int M, int N, int A[N][M], int B[M][N])
{
    int ii, jj, i, j;
    int t0, t1, t2, t3, t4, t5, t6, t7;

    /* iterate over 8x8 blocks */
    for (ii = 0; ii < 64; ii += 8) {
        for (jj = 0; jj < 64; jj += 8) {

            /* Stage 1: read the top-left 4x8 strip and place the
left half
               into B's top-left and the right half temporarily into
B's top-right */
            for (i = 0; i < 4; ++i) {
                t0 = A[ii + i][jj + 0];
                t1 = A[ii + i][jj + 1];
                t2 = A[ii + i][jj + 2];
                t3 = A[ii + i][jj + 3];
                t4 = A[ii + i][jj + 4];
                t5 = A[ii + i][jj + 5];
                t6 = A[ii + i][jj + 6];
                t7 = A[ii + i][jj + 7];

                B[jj + 0][ii + i] = t0;
                B[jj + 1][ii + i] = t1;
                B[jj + 2][ii + i] = t2;
                B[jj + 3][ii + i] = t3;

                /* place the right half of the row into B's block
rows 0..3 but columns ii+4..ii+7
                   (these are temporary positions that we will move
into final place below) */
                B[jj + 0][ii + 4 + i] = t4;
                B[jj + 1][ii + 4 + i] = t5;
                B[jj + 2][ii + 4 + i] = t6;
                B[jj + 3][ii + 4 + i] = t7;
            }

            /* Stage 2: move the temporarily stored pieces and
finish transpose for the block */
            for (j = 0; j < 4; ++j) {
                t0 = A[ii + 4][jj + j];
                t1 = A[ii + 5][jj + j];
                t2 = A[ii + 6][jj + j];
                t3 = A[ii + 7][jj + j];

                /* read the temp that we stored earlier */
                t4 = B[jj + j][ii + 4];
                t5 = B[jj + j][ii + 5];
                t6 = B[jj + j][ii + 6];
                t7 = B[jj + j][ii + 7];

                /* place left-bottom */
                B[jj + j][ii + 4] = t0;
                B[jj + j][ii + 5] = t1;
                B[jj + j][ii + 6] = t2;
                B[jj + j][ii + 7] = t3;

                /* move the temporary upper-right pieces into the
lower-left area */
                B[jj + 4 + j][ii + 0] = t4;
                B[jj + 4 + j][ii + 1] = t5;
                B[jj + 4 + j][ii + 2] = t6;
                B[jj + 4 + j][ii + 3] = t7;
            }
```

```c
            /* Stage 3: finish the bottom-right 4x4 corner */
            for (i = 4; i < 8; ++i) {
                t0 = A[ii + i][jj + 4];
                t1 = A[ii + i][jj + 5];
                t2 = A[ii + i][jj + 6];
                t3 = A[ii + i][jj + 7];

                B[jj + 4][ii + i] = t0;
                B[jj + 5][ii + i] = t1;
                B[jj + 6][ii + i] = t2;
                B[jj + 7][ii + i] = t3;
            }
        }
    }
}

/* ------------------- 61 x 67 implementation -------------------
   Use 16x16 blocking and bounds checks.
*/
void trans_61x67(int M, int N, int A[N][M], int B[M][N])
{
    int ii, jj, i, j;
    int block = 16;
    int imax, jmax;

    for (ii = 0; ii < N; ii += block) {
        for (jj = 0; jj < M; jj += block) {
            imax = (ii + block < N) ? ii + block : N;
            jmax = (jj + block < M) ? jj + block : M;
            for (i = ii; i < imax; ++i) {
                for (j = jj; j < jmax; ++j) {
                    B[j][i] = A[i][j];
                }
            }
        }
    }
}

/*
 * trans - A simple baseline transpose function, not optimized for
the cache.
 */
char trans_desc[] = "Simple row-wise scan transpose";
void trans(int M, int N, int A[N][M], int B[M][N])
{
    int i, j, tmp;
    for (i = 0; i < N; i++) {
        for (j = 0; j < M; j++) {
            tmp = A[i][j];
            B[j][i] = tmp;
        }
    }
}

/*
 * registerFunctions - This function registers your transpose
 *     functions with the driver.
 */
void registerFunctions()
{
    /* Register your solution function (the driver looks for this
desc) */
    registerTransFunction(transpose_submit, transpose_submit_desc);

    /* Register any additional transpose functions (optional
helpers/baseline) */
    registerTransFunction(trans, trans_desc);
}

/*
 * is_transpose - This helper checks if B is the transpose of A.
 */
int is_transpose(int M, int N, int A[N][M], int B[M][N])
{
    int i, j;

    for (i = 0; i < N; i++) {
        for (j = 0; j < M; ++j) {
            if (A[i][j] != B[j][i]) {
                return 0;
            }
        }
    }
    return 1;
}
```

Why the original sometimes behaved oddly (hypotheses & likely causes)

1. **Compiler codegen / register allocation differences**

When you keep all three cases inside one large function, the compiler’s register allocation and instruction scheduling can change in ways that affect memory-access ordering and register spills. On a direct-mapped cache small enough to be sensitive to access pattern, these small codegen differences can change the measured miss-counts. Splitting into separate functions gives each strategy a stable, smaller body and more consistent code generation.

2. **Diagonal handling bug (32×32)**
   In your posted 32×32 snippet you stored the diagonal into a temp variable `t0` but immediately wrote `B[j][i] = t0;` in the same inner iteration — that defeats the purpose of delaying the diagonal write to avoid cache conflict. The usual approach writes the diagonal only after finishing the inner `j` loop for that `i`. Writing it immediately can cause more thrashing or unexpected eviction behavior. My code saves the diagonal and writes it after the inner loop.
3. **Out-of-bounds / accidental memory corruption**
   The occasional `invalid` result almost always comes from writing outside the declared `B` array bounds (e.g., off-by-one or iterating past M/N in some block-edge logic), which corrupts memory used by the checker and yields `invalid`. Splitting into small, clearly-bounded helper functions and carefully guarding the 61×67 blocks with min checks removes this risk. If you ever used fixed block sizes without checking edges for a non-multiple-of-block-size matrix, you can overwrite past `B` or `A`.
4. **Too many local variables in a single function**
   The lab enforces a limit on local variables for the function you submit (so the driver can accurately compare). You were at the boundary (12 local vars). Putting everything in one function can push you over *or* change register spilling, which affects cache footprint and behavior. Splitting into separate functions keeps everything comfortably within limits per function.
5. **“Zero misses” — most likely not real**
   Zero misses is basically impossible for a non-trivial transpose on that cache. When you see that, it usually means the cache simulator didn’t actually see the memory references you expected (for example, the compiler optimized the code away, or your function didn't run the loops that touch memory due to a logic bug), or the driver timed out / was given garbage pointers because of memory corruption. The two most likely root causes:
   - your function returned before performing any memory writes (logic path), or
   - memory corruption (out-of-bounds write) broke the simulator’s bookkeeping.
     To catch that, compile with `-O0` and use the driver to run the function; also add an explicit `if (!is_transpose(...)) printf("bad\n");` while debugging.

Practical debugging steps I recommend

- Compile with `-O0 -g` and run the driver to see stable behavior while debugging.
- Add an `is_transpose` assertion or print right before returning during debugging

- Add an `is_transpose` assertion or print right before returning during debugging (remove later).
- If you get `invalid`, run under Valgrind (or `asan`) to find out-of-bounds writes.
- Test each helper individually by temporarily changing `transpose_submit` to call the helper directly for the size you want; that isolates behavior.
- Register only one function while debugging (comment out `registerTransFunction(trans,...)`) to avoid cross-test confusion.

If you want, I can:

- Run through a small step-by-step trace of one tile for the 64×64 routine to show exactly which memory addresses are referenced (useful for visualizing cache lines),
- Add `#ifdef DEBUG` prints that output the indices when `is_transpose` fails (so you can quickly find the first mismatch),
- Or, if you prefer, attach the assignment PDF and I'll double-check constraints such as "no more than 12 locals" for each submitted function.

Which follow-up would you like?