

2025 Introduction to Computer Software Systems Lab Assignment #4

Doyoung Kim

November 24, 2025

This assignment requires students to implement a cache simulator for analyzing cache behavior and evaluating the hit/miss ratio of a given program. More specifically, students must design their own version of the `csim` program, which accepts a memory trace file and cache configuration (the number of set index bits, the number of block offset bits, and the associativity of the cache) and produces statistics showing the total number of hits, misses, and evictions.

Additionally, students must implement a cache-friendly algorithm for matrix transpose. This requires careful examination of the locality properties related to memory accesses during matrix transpose. Techniques such as blocking and adjusting access patterns can be applied to minimize conflict and capacity misses, thereby improving performance.

This report discusses the design and implementation of the cache simulator and identifies the causes of frequent misses in the naive matrix transpose implementation, which iterates row-by-row through the source matrix while storing elements column-by-column into the destination matrix.

1 Designing the Cache Simulator

This section presents the primary data structures and core algorithms that constitute the cache simulator. Auxiliary details such as command-line argument parsing and input handling are omitted to emphasize the important architectural decisions.

1.1 Data Structures

The primary data structures are `access_t`, which represents a memory access, and `cache_t`, which represents the internal state of the cache. Their definitions are provided below.

```

typedef struct {
    char type;      // Type of a memory access.
    uint64_t addr; // Address of a memory access.
    size_t size;    // Size of a memory access.
} access_t;

```

Listing 1: Structure `access_t`.

```

typedef struct {
    size_t size;      // Total number of lines in the cache.
    bool *valid;     // Array of valid bits.
    uint64_t *tags;   // Array of tag fields.
    uint64_t *ranks;  // LRU ranks used for replacement.
} cache_t;

```

Listing 2: Structure `cache_t`.

An instance of `access_t` corresponds to each line in the trace file input. An instance of `cache_t` models the abstract behavioral state of the cache.

The `size` field in `cache_t` denotes the total number of cache lines, computed as $E \times S$, where E is associativity and S is the number of sets. `valid` tracks whether each line contains valid data, `tags` store the tag field of each line, and `ranks` store values used for the LRU replacement policy. A rank of 0 indicates the most recently accessed line, and a rank of $E - 1$ indicates the least recently accessed line.

No field for storing actual block data is required, since the simulator focuses solely on modeling cache behavior rather than storing memory contents.

1.2 Algorithm

1.2.1 Overall Simulation Loop

The main simulation loop is implemented in `runSimulation`, which repeatedly reads and processes accesses from the input trace file:

1. Initialize the cache state so that all lines are invalid.
2. For each line in the input:
 1. Parse the line into an instance of `access_t` via `parseAccess`.
 2. Update the cache state based on the parsed access through `processAccess`.
3. When the end of file is reached, collect statistics and release allocated memory.

Any errors in configuration or memory allocation trigger immediate program termination.

1.2.2 Manipulating Cache State

Each memory access is processed by `processAccess` using the following steps:

1. Extract the set index and tag from the access's address.
2. Check whether the corresponding line exists in the cache.
3. If a match is found, classify as a hit and invoke `updateRank` to record the recent use.
4. Otherwise, classify as a miss and select a line for replacement using `findLRULine`. A valid line that is overwritten results in an eviction.
5. If the access type is M (modify), a second access always results in a hit, so the hit count is incremented again.
6. Record and optionally print hit/miss/eviction statistics depending on verbosity flags.

This models realistic cache behavior under an LRU replacement policy.

1.2.3 Finding the LRU Line

```
/* Find the least recently used line within the set index `index`  
 * and returns the line index of the line. */  
static uint64_t findLRULine(cache_t *cache, uint64_t index) {  
    assert(cache != NULL && index < cache->size);  
  
    /* The line with rank of `assoc` - 1 is the least recently  
     * used; find it and return it. */  
    for (size_t line = index; line < index + assoc; line++) {  
        if (cache->ranks[line] == assoc - 1)  
            return line;  
    }  
  
    /* This should not happen in principle, so install an  
     * assertion for debugging. */  
    assert(false);  
}
```

Listing 3: The implementation of `findLRULine` function.

This function iterates through the set starting at index `index` and selects the entry whose rank is $E - 1$. The correctness relies on the invariant that the ranks for each set always form a permutation of $\{0, 1, 2, \dots, E - 1\}$. Example cache states are shown in Figure 1, where only the first state is valid under this invariant.

The figure shows two cache states, each with two sets (Set 0 and Set 1) and four lines per set. Each line is represented by a row with columns for Valid, Tag, and Rank.

	Valid	Tag	Rank
Set 0	1	0x1808	1
	1	0x2702	2
	1	0x2406	3
	1	0x1809	0
Set 1	1	0x1809	1
	1	0x6408	3
	1	0x2407	2
	1	0x6407	0

	Valid	Tag	Rank
Set 0	1	0x1808	3
	1	0x2702	2
	1	0x2406	3
	1	0x1809	1
Set 1	1	0x1809	1
	1	0x6408	3
	1	0x2407	2
	1	0x6407	0

Figure 1: Two sample cache states with 2 sets and 4 lines per set. The first state follows LRU rank invariants, while the second state is inconsistent.

Due to this invariant, the function cannot reach the end of it. Hence, an assertion that always fails is installed to terminate the program when the cache has violated the invariant.

1.2.4 Updating the Ranks

While `findLRULine` is responsible for finding the LRU line, `updateRank` takes charge of updating the ranks of lines so that it appropriately shows the time passed after the last access and is in consistent state.

```
/* Updates LRU ranks of `cache` for cache access for `line` and
 * set index `index` . */
static void updateRank(cache_t *cache, uint64_t index,
                      uint64_t line) {
    assert(cache != NULL && index < cache->size
          && line < cache->size);

    if (cache->ranks[line] == 0)
        return;

    for (size_t i = index; i < index + assoc; i++) {
        if (cache->ranks[i] < cache->ranks[line])
            cache->ranks[i]++;
    }

    cache->ranks[line] = 0;
}
```

Listing 4: The implementation of `updateRank` function.

The accessed line becomes rank 0, and all lines that were more recently used get their rank increased by one, ensuring a consistent strict ordering.

2 Optimizing Matrix Transpose

The naive transpose implementation performs poorly due to limited spatial locality. While reading rows of A provides good spatial locality, writing individual elements into different rows of B causes frequent cache line evictions.

2.1 Blocking

Blocking divides the matrix into $B \times B$ submatrices, as illustrated in Figure 2, enabling improved data reuse and reducing cache misses.

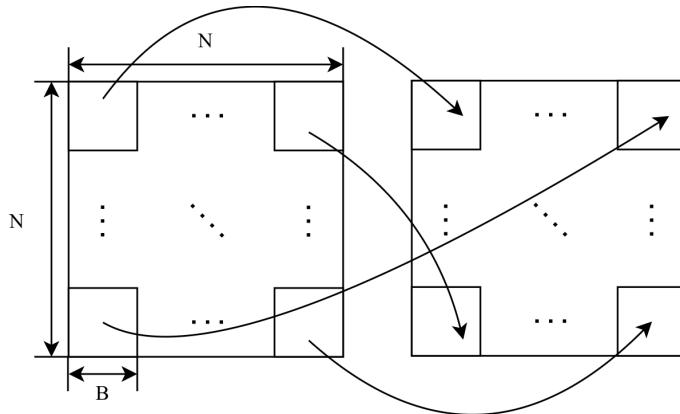


Figure 2: Blocking used in matrix transpose. Each $B \times B$ submatrix is loaded and transposed independently to improve locality.

Assuming a system with sufficient associativity, two blocks fitting fully inside the cache, and a block size of 8 elements per line, the naive implementation incurs approximately $\frac{9N^2}{8}$ misses. The blocked version reduces this to $\frac{N^2}{4}$, a substantial performance benefit for the same computational complexity.

However, block size selection is critical. Empirical evaluation on this simulator showed that optimal $B = 8$ for 32×32 and 64×64 matrices and optimal $B = 16$ for 61×67 matrices. This demonstrates that block size must be tuned to cache configuration and matrix dimensions.

2.2 Deferring Diagonal Accesses

Since the simulated cache is direct-mapped, rows of A and columns of B corresponding to the same indices frequently map to identical cache sets, especially within diagonal blocks. Writing diagonal elements immediately therefore causes repetitive eviction of useful lines.

To avoid this, diagonal writes are deferred until all non-diagonal elements of the block have been processed. Combining this altogether gives the following transpose algorithm for 32×32 and 61×67 matrices.

```

void transpose_32(int M, int N, int A[N][M], int B[M][N]) {
    int ib, jb, i, j, temp;

    for (ib = 0; ib < N; ib += 8) {
        for (jb = 0; jb < M; jb += 8) {
            for (i = ib; i < ib + 8 && i < N; i++) {
                for (j = jb; j < jb + 8 && j < M; j++) {
                    if (i == j)
                        temp = A[i][j];
                    else
                        B[j][i] = A[i][j];
                }

                if (ib == jb)
                    B[i][i] = temp;
            }
        }
    }
}

```

Listing 5: Transpose code for 32×32 matrix. The version for 61×67 uses the same strategy but with a different block size.

2.3 Buffering to the Destination

To further reduce conflict misses for the 64×64 case, the submitted version additionally employs buffering in the destination matrix. This technique temporarily stores elements in cache-friendly regions of B , avoiding thrashing caused by repeated accesses to sets that overlap between A and B . As a result, the miss count is significantly reduced below that of naive blocking. The submission code for 64×64 matrix, `transpose_64` makes use of this strategy.

3 Conclusion

In this assignment, a fully functional cache simulator based on a least recently used replacement policy was designed and implemented. The simulator correctly tracked hits, misses, and evictions under various cache configurations and memory access patterns. An optimized matrix transpose algorithm was also developed by applying several locality-enhancing techniques.

Through blocking, deferred diagonal writes, and careful memory access ordering, the optimized implementations achieved significant reductions in cache misses. Performance results demonstrated that:

- Optimal block size depends on cache capacity and matrix layout.
- Direct-mapped caches are highly susceptible to conflict misses arising from structural alignment.
- Minor changes to access patterns can yield dramatic improvements in cache utilization.

This exercise highlights the central role of the memory hierarchy in system performance. Even simple operations such as matrix transpose can exhibit widely different runtimes depending on how they interact with the cache. A deeper understanding of cache mapping, locality, and replacement algorithms enables software designers to improve program efficiency without modifying core computational logic.