

Project 2: Understanding Cache Memories

Xiaoyu Ma, 523031910469
Peiyao Yang, 523031910732
Mingzhe Yao, 523031910408
Yiru Zhu, 523031910141

1. Introduction

This project explores the impact of cache memories on program performance through 2 main tasks:

Part A: Implementing a cache simulator to analyze hit/miss behavior.

Part B: Optimizing a matrix transpose function to minimize cache misses.

The goal is to deepen understanding of cache architectures and performance optimization techniques.

Our code can be found at https://github.com/xiuyunMarx/Computer_Arch_project2

2. Experiments

2.1 Part A

2.1.1 Analysis

The cache simulator (*csim.c*) processes *valgrind* memory traces and simulates cache behavior using LRU replacement. Key steps included:

1. Parsing command-line arguments (*-s*, *-E*, *-b*, *-t*).
2. Allocating cache structures dynamically (e.g., using *malloc*).
3. Processing trace lines (ignoring "I" accesses, handling "L", "S", "M").
4. Tracking hits, misses, and evictions.

2.1.2 Code

Cache Data Structure:

We use a struct *CacheLine* to represent each cache line, containing:

tag: The tag bits of the memory address

valid: Whether the line contains valid data

dirty: Whether the line has been modified (for write-back policy)

lru: Counter for Least Recently Used replacement policy

```
struct CacheLine{
    unsigned long long tag;
    bool valid, dirty;
    int lru; // Least Recently Used counter
}cacheline[MAXN];
```

Configuration Parameters:

s: Number of set index bits ($S = 2^s$ sets)

E: Associativity (number of lines per set)

b: Number of block-offset bits (block size = 2^b bytes)

Replacement Policy:

We use LRU policy through that each access updates a global timer, and the line with smallest LRU time in a set is evicted when needed.

Function *main*:

Initializes cache data structures with all lines marked invalid:

```
memset(cacheline, 0, sizeof(cacheline));
```

Divide the command in order to get params:

```
if(argc < 5){
    fprintf(stderr, "Usage: %s [-hv] -s <s> -E <E> -b <b> -
t <tracefile>\n", argv[0]);
    return 1;
}else
for(int i=1 ;i<argc ;i++)
    if(strcmp(argv[i], "-h") == 0){
        helper = true;
        continue;
    }else if(strcmp(argv[i], "-v") == 0){
        verbose = true;
        continue;
    }else if(strcmp(argv[i], "-s") == 0){
        assert(i+1<argc);
        assert(argv[i+1][0] >= '0' && argv[i+1][0] <= '9');
        s = atoi(argv[i+1]);
        i++; //skip the next number
    }else if(strcmp(argv[i], "-E") == 0){
        assert(i+1<argc);
        assert(argv[i+1][0] >= '0' && argv[i+1][0] <= '9');
        E = atoi(argv[i+1]);
        i++; //skip the next number
    }else if(strcmp(argv[i], "-b") == 0){
        assert(i+1<argc);
        assert(argv[i+1][0] >= '0' && argv[i+1][0] <= '9');
        b = atoi(argv[i+1]);
        i++; //skip the next number
    }else if(strcmp(argv[i], "-t") == 0){
```

```

    assert(i+1<argc);
    assert(strlen(argv[i+1]) < 100);
    strcpy(tracefile, argv[i+1]);
    i++; //skip the next number
}else{
    fprintf(stderr, "Unknown option: %s\n", argv[i]);
    return 1;
}

```

For each memory access, extracts:

1. Tag bits (higher bits of address)
2. Set index (middle bits)
3. Block offset (lower bits)

```

n_sets = 1 << s; // Number of sets is 2^s
n_ways = E; // Number of ways is E
parseInput();

```

Cache Operations:

1. Load (L)
2. Store (S)
3. Modify (M): Treated as load followed by store

```

for(int i=0;i<tot;i++){
    unsigned long long off_mask = (1ULL << b) - 1; // 低 b 位全 1
    unsigned long long set_mask = (1ULL << s) - 1; // 接下来的位全 1
    unsigned long long addr = cacheOp[i].address;
    unsigned long long tag = addr >> (s + b);
    unsigned int set_index = (unsigned int)((addr >> b) & set_mask);
    unsigned int offset = (unsigned int)(addr & off_mask);
    assert(offset < 1 << b);
    if(cacheOp[i].operation == 'L'){
        handle_L(addr,set_index, tag, cacheOp[i].size);
    }else if(cacheOp[i].operation == 'S'){
        handle_S(addr, set_index, tag,cacheOp[i].size);
    }else{
        assert(cacheOp[i].operation == 'M');
        handle_L(addr, set_index, tag,cacheOp[i].size); // First Load
        handle_S(addr, set_index, tag,cacheOp[i].size); // Then store
        // For 'M', we handle it as a load followed by a store
    }
}

```

Statistics Tracking:

1. Counts hits, misses, and evictions
2. Can print verbose output for each operation when enabled

```

printSummary(HIT, MISS, EVICTION);

```

Parsing memory-visit operation:

Reads memory access traces from a file with format:

[operation] [address],[size]

Operations can be 'L' , 'S' and 'M' , and ignores 'I' operations

```
int parseInput(){
    char filepath[sizeof(tracefile) + 9]; //room for "./traces/" prefix
    strcpy(filepath, tracefile);
    FILE *fp = fopen(filepath, "r");
    if (fp == NULL) {
        fprintf(stderr, "Error opening trace file '%s'\n", filepath);
        return 1;
    }
    char line[256];
    while (fgets(line, sizeof(line), fp)) {
        if(line[0] == 'I') continue; // Ignore instruction loads
        char operation;
        unsigned long long addr;
        unsigned int size;
        if(sscanf(line, " %c %llx,%u", &operation, &addr, &size) == 3){
            cacheOp[tot].operation = operation;
            cacheOp[tot].address = addr;
            cacheOp[tot].size = size;
            tot++;
        }else break; // Stop reading if the line is not in the expected
format
    }
    return 0;
}
```

Load operations:

1. Checks if data is in cache (hit)
2. If not (miss), loads into an empty line or evicts LRU line
3. Updates LRU counters

```
void handle_L(unsigned long long addr, unsigned int set_index, unsigned
long long tag, int size){
    //Load operation
    uint startWay = set_index * n_ways, endWay = startWay + n_ways;
    uint j;
    for(j= startWay ;j < endWay; j++){
        if(cacheline[j].valid && cacheline[j].tag == tag){
            // Hit
            HIT++;
            cacheline[j].lru = timer++; // Update LRU counter
            if(verbose) printf("L %llu %d HIT", addr, size);
            break;
        }
    }
}
```

```

    }else if(!cacheline[j].valid){
        // Miss and empty line found
        MISS++;
        cacheline[j].valid = true; //Load from memory
        cacheline[j].tag = tag;
        cacheline[j].lru = timer++; // Update LRU counter
        cacheline[j].dirty = false; // Not dirty since it's a load
operation
        if(verbose) printf("L %llu %d MISS", addr, size);
        break;
    }
}
if(j == endWay){
    //Miss and no empty line found
    MISS++;
    EVICTION++;
    int minLRU=INT_MAX, pos;
    for(int k = startWay ; k < endWay; k++){
        if(cacheline[k].lru < minLRU){
            minLRU = cacheline[k].lru;
            pos = k; // Find the line with the minimum LRU value
        }
    }
    //Load from memory
    cacheline[pos].tag = tag; // Replace the line with the new tag
    cacheline[pos].lru = timer++; // Update LRU counter
    cacheline[pos].valid = true; // Mark it as valid
    cacheline[j].dirty = false; // Not dirty since it's a Load oper
ation
    if(verbose) printf("L %llu %d MISS EVICTION", addr, size);
}
return ;
}

```

Store operations:

Similar to load operations, but marks the line as dirty.

```

void handle_S(unsigned long long addr, unsigned int set_index, unsigned
long long tag, int size){
    //store operation
    uint startWay = set_index * n_ways, endWay = startWay + n_ways;
    uint j;
    for(j= startWay ;j < endWay; j++){
        if(cacheline[j].valid && cacheline[j].tag == tag){
            // Hit

```

```

        HIT++;
        cacheline[j].lru = timer++; // Update LRU counter
        cacheline[j].dirty = true; // Mark as dirty since it's a store operation

        if(verbose) printf("S %llu %d HIT", addr, size);
        break;
    }else if(!cacheline[j].valid){
        // Miss and empty line found
        MISS++;
        cacheline[j].valid = true; //load from memory
        cacheline[j].tag = tag;
        cacheline[j].lru = timer++; // Update LRU counter
        cacheline[j].dirty = true; // Mark as dirty since it's a store operation

        if(verbose) printf("S %llu %d MISS", addr, size);
        break;
    }
}
if(j == endWay){
    //Miss and no empty line found, evict a line
    MISS++;
    EVICTION++;
    int minLRU=INT_MAX, pos;
    for(int k = startWay ; k <endWay;k++){
        if(cacheline[k].lru <minLRU){
            minLRU = cacheline[k].lru;
            pos = k; // Find the line with the minimum LRU value
        }
    }
    //load from memory
    if(cacheline[pos].dirty){
        if(verbose) printf("evict dirty %llx block\n", cacheline[pos].tag);
    }
    cacheline[pos].tag = tag; // Replace the line with the new tag
    cacheline[pos].lru = timer++; // Update LRU counter
    cacheline[pos].valid = true; // Mark it as valid
    cacheline[pos].dirty = true; // Mark as dirty since it's a store operation

    if(verbose) printf("S %llu %d MISS EVICTION", addr, size);
}
return ;
}

```

```

} else { // 61x67
    for (temp0 = 0; temp0 < N; temp0 += 16) {
        for (temp1 = 0; temp1 < M; temp1 += 16) {
            temp2 = (temp0 == temp1);
            for (i = temp0; i < temp0 + 16 && i < N; i++) {
                if(temp2 && i < M){
                    for (j = temp1; j < temp1 + 16 && j < M; j++) {
                        if (i != j)
                            B[j][i] = A[i][j];
                        else {
                            diag = A[i][j];
                            // Delay writing diagonal to reduce cache conflicts
                        }
                    }
                    B[i][i] = diag;
                } else {
                    for (j = temp1; j < temp1 + 16 && j < M; j++) {
                        B[j][i] = A[i][j];
                    }
                }
            }
        }
    }
}

```

In further optimization, we divide the tile to 16*16 block, and switch to the optimal tile size base on the test conditions. In that way, we can achieve all test requirements of the test.

2.1.3 Evaluation

The result of *csim.c* is shown as following:

```

uu_matter@Ubuntu: ~/下载/Computer_Arch_project2-main/project2-handout
uu_matter@Ubuntu:~/下载/Computer_Arch_project2-main/project2-handout$ ./test-csim

```

Patterns (s,E,b)	Your simulator			Reference simulator			
	Hits	Misses	Evicts	Hits	Misses	Evicts	
3 (1,1,1)	9	8	6	9	8	6	traces/yi2.trace
3 (4,2,4)	4	5	2	4	5	2	traces/yi.trace
3 (2,1,4)	2	3	1	2	3	1	traces/dave.trace
3 (2,1,3)	167	71	67	167	71	67	traces/trans.trace
3 (2,2,3)	201	37	29	201	37	29	traces/trans.trace
3 (2,4,3)	212	26	10	212	26	10	traces/trans.trace
3 (5,1,5)	231	7	0	231	7	0	traces/trans.trace
6 (5,1,5)	265189	21775	21743	265189	21775	21743	traces/long.trace
27							

```

TEST_CSIM_RESULTS=27
uu_matter@Ubuntu:~/下载/Computer_Arch_project2-main/project2-handout$

```

2.2 Part B

2.2.1 Analysis

The naive row-wise transpose (*Simple row-wise scan*) resulted in excessive misses due to poor spatial locality (e.g., 1183 misses for 32*32).

Optimization methods:

Blocking: Divided the matrix into smaller blocks (e.g. 4*8*8 for 32*32) to exploit temporal locality.

Diagonal Handling: Delay writing diagonal blocks to reduce cache conflict misses.

Loop Unrolling: Reduced loop overhead for fixed-size matrices.

2.2.2 Code

Dealing with common condition, with blocking and diagonal handling:

```
for (temp0 = 0; temp0 < N; temp0 += 8) {
    for (temp1 = 0; temp1 < M; temp1 += 8) {
        for (i = temp0; i < temp0 + 8 && i < N; i++) {
            for (j = temp1; j < temp1 + 8 && j < M; j++) {
                if (i != j)
                    B[j][i] = A[i][j];
                else {
                    diag = A[i][j];
                    // Delay writing diagonal to reduce cache conflicts
                }
            }
            if (temp0 == temp1) {
                B[i][i] = diag;
            }
        }
    }
}
```

Dealing with $M\%8==0$ and $N\%8==0$ condition, with blocking, diagonal handling and loop unrolling:

```
for (i = 0; i < N; i += 8) {
    for (j = 0; j < M; j += 8) {
        // upper 4 rows
        for (k = 0; k < 4; k++) {
            temp0 = A[i + k][j + 0];
            temp1 = A[i + k][j + 1];
            temp2 = A[i + k][j + 2];
            temp3 = A[i + k][j + 3];
            temp4 = A[i + k][j + 4];
            temp5 = A[i + k][j + 5];
            temp6 = A[i + k][j + 6];
            temp7 = A[i + k][j + 7];
```



```

        // write first half into B directly
        B[j + 0][i + k] = temp0;
        B[j + 1][i + k] = temp1;
        B[j + 2][i + k] = temp2;
        B[j + 3][i + k] = temp3;
        // write second half in temporary positions in B
        B[j + 0][i + k + 4] = temp4;
        B[j + 1][i + k + 4] = temp5;
        B[j + 2][i + k + 4] = temp6;
        B[j + 3][i + k + 4] = temp7;
    }
    // lower 4 rows
    for (k = 0; k < 4; k++) {
        temp0 = B[j + k][i + 4];
        temp1 = B[j + k][i + 5];
        temp2 = B[j + k][i + 6];
        temp3 = B[j + k][i + 7];
        // write second half into B directly
        B[j + k][i + 4] = A[i + 4][j + k];
        B[j + k][i + 5] = A[i + 5][j + k];
        B[j + k][i + 6] = A[i + 6][j + k];
        B[j + k][i + 7] = A[i + 7][j + k];
        // write first half in temporary positions in B
        B[j + k + 4][i + 0] = temp0;
        B[j + k + 4][i + 1] = temp1;
        B[j + k + 4][i + 2] = temp2;
        B[j + k + 4][i + 3] = temp3;
    }
    // the lower-right 4x4 block
    for (temp0 = 4; temp0 < 8; temp0++) {
        for (temp1 = 4; temp1 < 8; temp1++) {
            B[j + temp1][i + temp0] = A[i + temp0][j + temp1];
        }
    }
}
}

```

2.2.3 Evaluation

The result is shown as following(since 61×67 is not optimized through loop unrolling, the performance is poor than the others):

As the result shows, we have achieved the optimal trans performance.

```

yaomz@Ubuntu-22:~/桌面/ComputerArchitecture/Computer_Arch_project2-fixed_driver_bugs/project2-handout$ python3 driver
Part A: Testing cache simulator
Running ./test-csim

```

Points (s,E,b)	Your simulator			Reference simulator			
	Hits	Misses	Evicts	Hits	Misses	Evicts	
3 (1,1,1)	9	8	6	9	8	6	traces/yi2.trace
3 (4,2,4)	4	5	2	4	5	2	traces/yi.trace
3 (2,1,4)	2	3	1	2	3	1	traces/dave.trace
3 (2,1,3)	167	71	67	167	71	67	traces/trans.trace
3 (2,2,3)	201	37	29	201	37	29	traces/trans.trace
3 (2,4,3)	212	26	10	212	26	10	traces/trans.trace
3 (5,1,5)	231	7	0	231	7	0	traces/trans.trace
6 (5,1,5)	265189	21775	21743	265189	21775	21743	traces/long.trace

```

27

Part B: Testing transpose function
Running ./test-trans -M 32 -N 32
Running ./test-trans -M 64 -N 64
Running ./test-trans -M 61 -N 67

Cache Lab summary:

```

	Points	Max pts	Misses
Csim correctness	27.0	27	
Trans perf 32x32	8.0	8	287
Trans perf 64x64	8.0	8	1275
Trans perf 61x67	10.0	10	1985
Total points	53.0	53	

3. Conclusion

3.1 Problems

There are 2 main issues we encountered:

1. Correctly simulating associativity and LRU eviction;
2. Verbose mode debugging.

We handled the problems by:

1. We used a queue-based approach to track recency within each set;
2. We added `-v` flag support to log individual accesses.

3.2 Achievements

Part A: We successfully simulated cache behavior with LRU policy.

Part B: We achieved optimal cache performance through blocking and diagonal handling.

Learning Outcome: Through coding and simulating the behavior of cache, we've got practical understanding of cache architectures and optimization trade-offs.

Future Work: We would explore non-power-of-two matrices and adaptive blocking strategies.