CSC3050 Project 4: Cache Simulator Report

Group Members: Runyuan He, Mingshi Deng

Student ID: 123090163, 123090081

Contribution: 50%, 50%

Note: Only leave sections blank if they are not yet implemented. We will verify that your code matches the data in your report. Any inconsistencies will result in double point deductions.

1. Part 1

1.1 Split Cache Design

Split cache is a way of organizing the cache in computer, where cache is split into ICache and DCache. Instruction Cache (ICache) is dedicated to storing instructions, and Data Cache (DCache) is dedicated to storing data. Unified cache does not have such split.

While unified cache has one access path, split cache allows the acquisition of instructions and data to be carried out in parallel, accelerating memory access by increasing throughput.

1.2 Implementation Details

ICache is similar to DCache, except that since instructions are usually unmodifiable, ICache is read-only, which means it should not consider/support the `setByte()` function.

To simulate parallel access, we assume start fetching both caches at same time, the overall consumption should be:

$$\text{Total Cycle} = \max(TotalCycle_I, TotalCycle_D)$$

That is, the total time consumption depends on the value that takes longer in the two caches.

1.3 Result Analysis

Trace File	Cache Type	Miss Count	Expected Miss Count	Total Cycles	Expected Total Cycles
I.trace	Split Cache	512	512	41088	41088
I.trace	Unified Cache	384	384	54912	54912
D.trace	Split Cache	634	634	67584	67584
D.trace	Unified Cache	507	507	68141	68141

By analyzing the data, we can discover the following several phenomena:

- Using Split cache makes the miss count of test results of different trace files increase.

  > Instructions are relatively continuous in memory, so the cache hit rate of instructions is higher. After using split cache, the size of i-cache limits the amount of instruction cache, and reducing the hit rate.

- The rate of growth of miss count for I_Trace is higher than that of D_Trace.

  > Therefore, if the proportion of instructions in the total access is higher, this impact will be correspondingly greater.

- Using Split cache makes the total cycles of test results of different trace files decrease.

  > By increasing the throughput, split cache reduces the total cycles.

2. Part 2

Correct results after bug fixed:

Cache Level	Read Count	Write Count	Hit Count	Miss Count	Miss Rate	Total Cycles
L1	181,708	50,903	177,911	54,700	23.5%	717,519
L2	47,667	7,033	25,574	29,126	53.1%	888,292
L3	26,984	2,142	21,596	7,530	25.9%	1,184,920

My Results after bug fixed:

Cache Level	Read Count	Write Count	Hit Count	Miss Count	Miss Rate	Total Cycles
L1			177,911	54,700	23.5%	717,519
L2			25,608	29,092	53.2%	886,984
L3			21,562	7,530	25.9%	1,184,240

3. Part 3

Optimization Techniques Application and Results

(Choose one optimization technique for each trace file)

No Optimization means, just use the initial code to run simulation.

Trace	Optimization	Miss Count	Miss Rate	Expected Miss Rate
test1.trace	No Optimization	102656	99.26%	None
test1.trace	FIFO	384	0.3713%	0.371%
test2.trace	No Optimization	100013	98.532%	None
test2.trace	Pre-fetching	1272	1.253%	5.803%
test3.trace	No Optimization	51712	50.12%	None
test3.trace	Victim Cache	384	0.3722%	0.372%

4. Part 4

4.1 Performance Comparison (L1 Level)

Implementation	Read Count	Write Count	Hit Count	Miss Count	Miss Rate	Total Cycles
matmul0	786432	262144	716581	331995	31.66%	3665589
matmul1	528384	4096	258741	273739	51.41%	2480901
matmul2	528384	262144	739284	51244	6.48%	1462948
matmul3	528384	262144	231478	559050	70.72%	6800774
matmul4	532480	8192	514453	26219	4.85%	785373

4.2 Analysis of Performance Differences between matmul1, 2, 3

The performance of matrix multiplication implementations (matmul1, matmul2, matmul3) is dictated by their memory access patterns and how well they exploit spatial locality in a row-major storage system. Below is a concise comparison:

1. matmul3 (`jki` order) - Worst Performance

- **Inner Loop**: Iterates over `i`, accessing `C[i][j]` and `A[i][k]`.
- **Access Pattern**:
  - **Column-wise access** for both `C` and `A` (stride = `n * sizeof(double)`).
  - Poor spatial locality due to large strides in row-major storage.
- **Impact**:
  - **70.72% Miss Rate, 6.8M Cycles**
  - Nearly every access misses the cache, as two matrices suffer column-wise traversal in the inner loop.

---

2. matmul1 (`ijk` with scalar `cij`) - Moderate Performance

- **Inner Loop**: Iterates over `k`, accessing `A[i][k]` (row) and `B[k][j]` (column).
- **Access Pattern**:
  - **Good**: `A` is accessed row-wise (stride 1).
  - **Poor**: `B` is accessed column-wise (large stride).
- **Scalar** `cij`: Reduces redundant writes to `C` (temporal locality), but `B`'s column access dominates performance.
- **Impact**:
  - **51.41% Miss Rate, 2.4M Cycles**
  - Bottlenecked by `B`'s poor spatial locality.

---

3. matmul2 (`kij` order) - Best Performance

- **Inner Loop**: Iterates over `j`, accessing `C[i][j]` and `B[k][j]`.
- **Access Pattern**:
  - **Row-wise access** for both `C` and `B` (stride 1).
  - Excellent spatial locality in the inner loop.
- **Outer Loop**: `A[i][k]` is column-wise (poor locality), but this is overshadowed by efficient inner-loop accesses.
- **Impact**:
  - **6.48% Miss Rate, 1.5M Cycles**
  - Inner-loop stride-1 accesses dominate, minimizing cache misses.

Key Takeaways

- **Spatial Locality**: Inner-loop stride-1 accesses maximize cache efficiency.
- **Row-Major Storage**: Column-wise access (large stride) leads to cache thrashing.
- **Hierarchy of Importance**:
  - Inner-loop access patterns > outer-loop patterns.
  - Two stride-1 accesses (matmul2) outperform one good/one poor (matmul1) and two poor (matmul3).

Conclusion: Optimizing memory access patterns for spatial locality is critical. matmul2's design ensures two stride-1 accesses in the inner loop, making it **3–5x faster** than alternatives with poor locality.