Lab 2:
Cache

NAME: YUHAO ZHANG
STUDENT NUMBER: 2021533141
EMAIL: ZHANGYH7@SHANGHAITECH.EDU.CN

1 INTRODUCTION

In this lab, we tested the performance of the cache under different replacement policies and inclusion policies. After that, we added a victim cache to increase the performance of the cache.

- Implement two inclusion policies in a two level cache.
- Implement two replacement policies and compare their performance.
- Add a victim cache to the cache hierarchy.
- Test the cache implementations with different algorithms.

2 INCLUSION POLICY

2.1 Original Policy

In the original simulator, the cache uses non-inclusive policy, it does not restrict the behavior of the two levels of cache. Data can be present in either the L1 or L2 cache, both, or neither.

2.2 Inclusive Policy

the lower-level cache (e.g., L2 cache) contains a superset of the data that is present in the higher-level cache (e.g., L1 cache). So when we evict a cache block in the lower level, we should do the back invalidation to check if the same data is in the upper level.

2.3 Exclusive Policy

Under an exclusive cache policy, the data stored in the lower-level cache is exclusive of that in the higher-level cache, so no data element is duplicated across the cache levels.

2.4 Test and Analysis

We design a case to validate the inclusive policy in the *inclusive.trace* file. In this case, the associativity of cache is 2, and all data has the same index. For simplicity, we label the data as A, B, C in order.

First two access A, B will cause compulsory miss for sure, then the third access A will hit in the L1 cache. When C comes, it will take the B's place, besides, it will evict A in L2 cache. It's all right now if we use non-inclusive policy, but if we use inclusive policy, we should also invalidate the A in L1 cache. So the fifth access A will lead to a cache miss. We will see the difference if we use the commands below:

- ./InclusionPolicyTest ../cache-trace/inclusive.trace NON-INCLUSIVE
- ./InclusionPolicyTest ../cache-trace/inclusive.trace INCLUSIVE

We also design a test case for exclusive policy, you can see it in *exclusive.trace* file. The setting remains the same. We still label the data as A, B, C for simplicity. After the fourth access complete, the data in L1 cache is A, C, the data in L2 cache is C, B. Then another B comes and replace A in L1. If we use non-inclusive policy now, L2 will stay the same, but if we use exclusive policy, the data B in L2 will be replaced by A, who just evicted from L1. Then when another A comes, there will be a L2 hit in this cache instead of a cache miss. You can test it using the commands below:

- ./InclusionPolicyTest ../cache-trace/exclusive.trace NON-INCLUSIVE
- ./InclusionPolicyTest ../cache-trace/exclusive.trace EXCLUSIVE

3 REPLACEMENT POLICY

3.1 LRU Policy

LRU-Least Recently Used, evicts the least recently used item first. It's a very intuitive approach, but if we want to implement it precisely, it will be hardware-intensive as it requires maintaining arecord of the usage order for all cache entries.

3.2 RRIP Policy

RRIP-Re-Reference Interval Prediction, is a more modern approach that predicts the likelihood of an item being re-referenced soon. It assigns a re-reference value to each cache line. The value is updated based on accesses, and when a replacement is needed, the line with the highest re-reference value (indicating it's least likely to be used soon) is evicted. To implement it we need to maintain a rrpv variable for each cache block and find the block to be replaced based on it.

3.3 Optimal Policy

Optimal policy evicts the item that will not be used for the longest time in the future. It requires future knowledge of the access pattern, which is impossible in practical systems. To implement it we need to get the cache trace before run the program. When a block is about to be replaced, we look at all the blocks that could be replaced and find the one that appears last in cache trace.

3.4 Test and Analysis

We can test it by running *ReplacementPolicyTest*, the associativity is 8 in this case:

- ./ReplacementPolicyTest ../cache-trace/optimal.trace LRU
- ./ReplacementPolicyTest ../cache-trace/optimal.trace RRIP

- ./ReplacementPolicyTest ../cache-trace/optimal.trace OPTIMAL

Performance difference: Optimal > RRIP > LRU in this case.

4 VICTIM CACHE

A victim cache is an extra cache to reduce the miss rate, it works like a filter. When a block is evicted from the L1 cache, it is placed in the victim cache instead of being discarded immediately. If there's a subsequent request for this data, it can be quickly retrieved from the victim cache, thereby reducing the access time and improving overall cache performance. Since the data in the victim cache already has a backup in the L2 cache, so when a block replacement occurs in the victim cache, we can throw the evicted block directly.

To show the performance improvement after adding the victim cache, we can use the commands:

- ./VictimCacheTest ../cache-trace/victim.trace
- ./VictimCacheTest ../cache-trace/victim.trace V
- ./VictimCacheTest ../cache-trace/conflict.trace
- ./VictimCacheTest ../cache-trace/conflict.trace V

The number of memory accesses in these two files is basically the same, but data in *conflict.trace* will cause a certain number of conflict misses, and the victim cache performs better in this case. We can infer that the victim cache will significantly increase the workloads that leads to high conflict rates.

5 EVALUATION WITH APPLICATION

For the cross test we chose *quicksort.riscv* as our test case, you can test it with the commands:

- ./CrossTest ../riscv-elf/quicksort.riscv
- ./CrossTest -v ../riscv-elf/quicksort.riscv

The result is shown below:

L1 policy	**L2 policy**	**inclusion**	**L1 cycles**	**L2 cycles**
RRIP	RRIP	inclusive	1116264	73519128
RRIP	LRU	inclusive	1116264	73519148
RRIP	RRIP	exclusive	1118912	73688208
RRIP	LRU	exclusive	1118912	73688208
RRIP	RRIP	non-inc	1118912	73688280
RRIP	LRU	non-inc	1118912	73688308
LRU	RRIP	inclusive	1050824	69333372
LRU	LRU	inclusive	1050824	69333536
LRU	RRIP	exclusive	1050480	69310840
LRU	LRU	exclusive	1050480	69310840
LRU	RRIP	non-inc	1050480	69311324
LRU	LRU	non-inc	1050480	69311520

Table 1. quicksort case without victim cache.

In the quicksort test case, the LRU policy performs better.

You can find all the test commands in *README.md*.

L1 policy	**L2 policy**	**inclusion**	**L1 cycles**	**L2 cycles**
RRIP	RRIP	inclusive	1089342	71549944
RRIP	LRU	inclusive	1089342	71549996
RRIP	RRIP	exclusive	1050339	69292388
RRIP	LRU	exclusive	1050339	69292388
RRIP	RRIP	non-inc	1091206	71661752
RRIP	LRU	non-inc	1091206	71661812
LRU	RRIP	inclusive	1050810	69332292
LRU	LRU	inclusive	1050810	69332152
LRU	RRIP	exclusive	1050466	69309840
LRU	LRU	exclusive	1050466	6930840
LRU	RRIP	non-inc	1050466	69310244
LRU	LRU	non-inc	1050466	69310496

Table 2. quicksort case without victim cache.