Lab4 implementation details – 25-1 Computer Organization

2023-11534 조현우

Part I

In our cache_base_c::access() function, each incoming address is first split into (offset, set index, tag) using the cache's line size and number of sets. We look up the tag in the target cache_set_c by calling its find method, which naively looks through all of the set's entries for a valid match. On a hit appropriate stat variables are updated and LRU_update is called to move that line to the MRU position in the set's LRU_queue. On a miss, we call evict which returns an invalid slot index if it exists, or the entry at the very back of the LRU. If the victim was dirty, we increment the writeback counter. We then overwrite the victim's tag and valid bit. At the very end we appropriately change the values of hit/miss and write counters. The stats for 16kB, 1/2/4/8 way, 64 line size cache is given as follows.

1way	95.1%
2way	96.4078%
4way	96.8525%
8way	97.1274%

Part II

In Part II, the cache was implemented with four queues—in_queue, out_queue, fill_queue, and wb_queue—to model the memory access pipeline. Incoming requests are added to in_queue, processed after the cache latency, and on a miss, forwarded to out_queue for access to the next level. Responses and victims are handled in fill_queue, where they are installed into the cache. Write-backs are issued via wb_queue, which feeds into out_queue. Queues are processed in the order: write-back, fill, out, and in, enabling staged and latency-aware cache behavior. PartII stats for the configuration of 16KB, 4-way SA, LRU, 4-cycle are given as follows:

	L1U
Hitrate	96.8525
#access	1000000
#hit	968525
#miss	31475
#write	80669
#wb	2950

The performance stats are given as follows:

CPI	10.7347
#cycles	8367825
#insts	779515
#mem-insts	220485

It can be seen that the stats do match to the PartI case. Also running the code on 1, 2, 8-way config yields identical results to partI

Part III

Num_backinvals indicate the number of back invalidations generated by eviction at L2 cache. The writebacks due to back invalidations count the number of writebacks caused by back invalidation(=number dirty entries removed due to back invalidation). Notice that even though num_writebacks_backinval is incremented on the corresponding l1 cache, none are observed for l1 instruction caches, as there are no writes(thus no dirty lines) on instruction memory. The stats for the separate instr. and data cache, with the configuration of 2KB, 2-way, LRU, 64B line size, 4cycle latency l1 caches, and 16KB, 4-way, inclusive, LRU, 64B line size, 10cycle latency unified l2 cache, is given as follows:

	L1i	L1d	L2
Hitrate	93.4004	88.2069	58.8364
#access	779515	220485	77447
#hit	728070	194483	45567
#miss	51445	26002	31880
#write	0	80669	5824
#wb	0	8837	2402
#backinv	0	0	3937
#wb-bi	0	1212	0

The performance stats were given as follows:

CPI	11.1934
#cycles	8725454
#insts	779515
#mem-insts	220485

Note that the stats are also submitted in the PartIII commit in github in txt format.

Part IV

Running single_request=0 on this simulator we can be faced with the double counting/execution problem, fill conflict problem, and incorrect in-order request assumption. Here the problem that affects the stats most severely is the request conflicts. When we issue many requests at once, duplicate fill requests may be sent(as many requests may ask for the same data that is not present at the cache in one cycle, each of those will generate a fill request, which will corrupt the fill queue and statistics. To avoid this, I implemented a duplicate checking logic to ensure that no duplicate requests are sent for read/write/wb/fill types. The stats obtained from the identical settings as PartIII are given as follows:

	L1i	L1d	L2
Hitrate	93.3838	88.1915	58.937
#access	779515	220485	77610
#hit	727941	194449	45741
#miss	51574	26036	31869
#write	0	80669	5606
#wb	0	8832	2472
#backinv	0	0	4254
#wb-bi	0	1397	0

The performance stats were given as follows: