Kristopher Lopez, Daniel Rodriguez, Darius Olega

Dr. Anup Das

ECEC 412: Modern Processor Design

8 October 2021

Project 1: Implementing Cache Replacement Policy

Objective

The objective of this project is to evaluate the different methods of cache replacement, namely the Least-Recently Used (LRU) and Least-Frequently Used (LFU) replacement policies, which evaluates items in the cache by either their most recent access or frequency of accesses over a period of time, and chooses that item when a new item needs to be accessed.

LRU Replacement Policy

The Least-Recently Used (LRU) replacement policy is already included and provides the foundation for how to handle cache replacement in three steps. First, a for-loop iterates through each of the blocks within the cache to find an invalid block; there would be no need to remove a valid block if there's an unused block already available. If there isn't an invalid block, each block's attribute 'when_touched', or the last occasion when the block was used, is evaluated. The least recent would be saved as the victim block. This victim block would then be written back to memory and invalidated in the cache in order to make space for the incoming block.

LFU Replacement Policy

The Least-Frequently Used (LFU) replacement policy was simple to implement due to the provided framework of LRU and the already stored attribute of frequency associated with each item that exists in the cache. This means that in order to implement LRU, a function with the same input parameters as LRU was defined and follows the same steps as the LRU function, except evaluating the iterator's parameter 'frequency' with the defined victim's frequency to find the victim block, then invalidating the victim after iterating through each of the blocks.

To add a feature in the code for changing between LRU and LFU, an additional variable "mode" was included, along with an if statement in step one for deciding which replacement policy to use for evaluating the trace. This variable is defined by a user input, with an additional argument included in the main function. An input of 0 will call LRU, while 1 will call LFU.

```
53  bool lru(Cache *cache, uint64_t addr, Cache_Block **victim_blk, uint64_t *wb_addr);
54  bool lfu(Cache *cache, uint64_t addr, Cache_Block **victim_blk, uint64_t *wb_addr);
```

Figure 1. Defining the LFU function's parameters. Note the similarity to LRU's.

```
209 bool lfu(Cache *cache, uint64_t addr, Cache_Block **victim_blk, uint64_t *wb_addr)
210 {
211     uint64_t set_idx = (addr >> cache->set_shift) & cache->set_mask;
212     //    printf("Set: %"PRIu64"\n", set_idx);
213     Cache_Block **ways = cache->sets[set_idx].ways;
214
215     // Step one, try to find an invalid block.
216     int i;
217     for (i = 0; i < cache->num_ways; i++)
218     {
219         if (ways[i]->valid == false)
220         {
221             *victim_blk = ways[i];
222             return false; // No need to write-back
223         }
224     }
225
226     // Step two, if there is no invalid block. Locate the LFU block
227     Cache_Block *victim = ways[0];
228     for (i = 1; i < cache->num_ways; i++)
229     {
230         if (ways[i]->frequency < victim->frequency)
231         {
232             victim = ways[i];
233         }
234     }
235
236     // Step three, need to write-back the victim block
237     *wb_addr = (victim->tag << cache->tag_shift) | (victim->set << cache->set_shift);
238 //    uint64_t ori_addr = (victim->tag << cache->tag_shift) | (victim->set << cache->set_shift);
239 //    printf("Evicted: %"PRIu64"\n", ori_addr);
240
241     // Step three, invalidate victim
242     victim->tag = UINTMAX_MAX;
243     victim->valid = false;
244     victim->dirty = false;
245     victim->frequency = 0;
246     victim->when_touched = 0;
247
248     *victim_blk = victim;
249
250     return true; // Need to write-back
251 }
```

Figure 2. The created LFU function, created with the provided LRU structure. Note the change in line 230, evaluating frequency instead of parameter "when_touched" in LRU.

```
/* Constants */
const unsigned block_size = 64; // Size of a cache line (in Bytes)
// TODO, you should try different size of cache, for example, 128KB, 256KB, 512KB, 1MB, 2MB
unsigned cache_size = 128; // Size of a cache (in KB)
// TODO, you should try different association configurations, for example 4, 8, 16
unsigned assoc = 4;

unsigned mode = 0;

void set_arg_vals(unsigned c, unsigned a, unsigned m)
{
    cache_size = c;
    assoc = a;
    mode = m;
}
```

Figure 3. Defining variables such as the changing cache size and associativity, but also "mode", which determines which replacement policy is used to evaluate the trace.

```
110  bool insertBlock(Cache *cache, Request *req, uint64_t access_time, uint64_t *wb_addr)
111  {
112      // Step one, find a victim block
113      uint64_t blk_aligned_addr = blkAlign(req->load_or_store_addr, cache->blk_mask);
114
115      bool wb_required;
116      Cache_Block *victim = NULL;
117      if(mode == 0)
118      {
119          bool wb_required = lru(cache, blk_aligned_addr, &victim, wb_addr);
120      } else {
121          bool wb_required = lfu(cache, blk_aligned_addr, &victim, wb_addr);
122      }
123      assert(victim != NULL);
124
```

Figure 4. The first step of "insertBlock". Note the if statement inserted in line 117, namely for choosing which replacement policy to run when evaluating the trace.

```
1   int main(int argc, const char *argv[])
2   {
3       if (argc != 5)
4       {
5           printf("Usage: %s %s %s %s %s\n", argv[0], "<mem-file>", "<cache-size>", "<assoc>", "<mode>");
6
7           return 0;
8       }
9
0       set_arg_vals(atoi(argv[2]), atoi(argv[3]), atoi(argv[4]));
1
```

Figure 5. The main function, edited to insert an additional argument for changing the replacement policy based on user input.

Results and Comparison

After comparing the two replacement policies, the details have been listed in the tables in Appendix A, with the terminal output run via a script to test varying levels of cache size and assoc values located in Appendix B.

Based on the results from the provided AI traces, the following conclusions were reached:

- For some cases, LFU will perform better than LRU, but is not guaranteed depending on the traces.
- As the associativity value increases, the hit rate percentage increases as well.
- As cache size increases, so does the hit rate.

While the latter two trends are expected, the first one is dependent on the traces provided. While LFU seems like a more optimized policy in theory, there are ways to trick the algorithm based on the sequence of load and store instructions processed.

Most notably, if there is an item that is loaded repeatedly for a significant number of instructions in a short period of time, but never accessed after that, the item could stay in the cache for an unnecessary period of time. However, in the case of LRU, this block would be sent back sooner. There aren't too many ways that the algorithm could fail in LRU that it wouldn't for LFU, potentially a cache block that is checked periodically could be removed if a wide volume of load instructions are run at once.

Appendix A: Table of Hit Rates

548.exchange2_r_llc.mem_trace		assoc-size		
LRU		4	8	16
cache-size	128	99.93%	99.96%	99.99%
	256	99.98%	99.99%	99.99%
	512	99.99%	99.99%	99.99%
	1024	99.99%	99.99%	99.99%
	2048	99.99%	99.99%	99.99%
548.exchange2_r_llc.mem_trace		assoc-size		
LFU		4	8	16
cache-size	128	86.32%	72.67%	99.99%
	256	99.98%	99.99%	99.99%
	512	99.99%	99.99%	99.99%
	1024	99.99%	99.99%	99.99%
	2048	99.99%	99.99%	99.99%
sample.mem_trace		assoc-size		
LRU		4	8	16
cache-size	128	42.58%	42.78%	43.74%
	256	56.60%	60.87%	62.25%
	512	65.07%	65.41%	65.60%
	1024	65.58%	65.60%	65.60%
	2048	65.60%	65.60%	65.60%
sample.mem_trace		assoc-size		
LFU		4	8	16
cache-size	128	47.52%	50.74%	51.69%

	256	59.45%	63.42%	64.84%
	512	65.30%	65.52%	65.60%
	1024	65.60%	65.60%	65.60%
	2048	65.60%	65.60%	65.60%
531.deepsjeng_r_llc.mem_trace		assoc-size		
LRU		4	8	16
cache-size	128	76.94%	79.38%	81.20%
	256	88.97%	90.79%	91.51%
	512	93.84%	94.86%	95.04%
	1024	95.52%	95.68%	95.75%
	2048	95.91%	95.95%	95.96%
531.deepsjeng_r_llc.mem_trace		assoc-size		
LFU		4	8	16
cache-size	128	75.30%	75.95%	74.91%
	256	88.24%	88.80%	88.17%
	512	93.51%	93.95%	93.18%
	1024	95.33%	95.04%	94.47%
	2048	95.81%	95.60%	95.21%
541.leela_r_llc.mem_trace		assoc-size		
LRU		4	8	16
cache-size	128	42.55%	42.59%	43.73%
	256	70.74%	75.74%	78.82%
	512	92.36%	94.44%	96.02%
	1024	98.22%	98.87%	99.01%
	2048	99.46%	99.60%	99.63%

541.leela_r_llc.mem_trace	assoc-size		
LFU	4	8	16
cache-size 128	45.55%	47.17%	46.78%
256	69.85%	71.79%	71.41%
512	89.24%	87.74%	85.24%
1024	96.63%	96.41%	94.45%
2048	99.13%	99.28%	99.06%

Appendix B: Sample Terminal Output

FILE NAME cache-size assoc-size mode
mode = 1 | 0 -> 0 is LRU and 1 is LFU

548.exchange2_r_llc.mem_trace 128 4 0
Hit rate: 99.926982%
548.exchange2_r_llc.mem_trace 128 8 0
Hit rate: 99.958232%
548.exchange2_r_llc.mem_trace 128 16 0
Hit rate: 99.985094%
548.exchange2_r_llc.mem_trace 256 4 0
Hit rate: 99.984771%
548.exchange2_r_llc.mem_trace 256 8 0
Hit rate: 99.985094%
548.exchange2_r_llc.mem_trace 256 16 0
Hit rate: 99.985094%
548.exchange2_r_llc.mem_trace 512 4 0
Hit rate: 99.985094%
548.exchange2_r_llc.mem_trace 512 8 0
Hit rate: 99.985094%
548.exchange2_r_llc.mem_trace 512 16 0
Hit rate: 99.985094%
548.exchange2_r_llc.mem_trace 1024 4 0
Hit rate: 99.985094%
548.exchange2_r_llc.mem_trace 1024 8 0
Hit rate: 99.985094%
548.exchange2_r_llc.mem_trace 1024 16 0
Hit rate: 99.985094%
548.exchange2_r_llc.mem_trace 2048 4 0
Hit rate: 99.985094%
548.exchange2_r_llc.mem_trace 2048 8 0
Hit rate: 99.985094%
548.exchange2_r_llc.mem_trace 2048 16 0
Hit rate: 99.985094%
548.exchange2_r_llc.mem_trace 128 4 1
Hit rate: 86.318316%
548.exchange2_r_llc.mem_trace 128 8 1
Hit rate: 72.666252%
548.exchange2_r_llc.mem_trace 128 16 1

Hit rate: 99.985094%
548.exchange2_r_llc.mem_trace 256 4 1
Hit rate: 99.984948%
548.exchange2_r_llc.mem_trace 256 8 1
Hit rate: 99.985094%
548.exchange2_r_llc.mem_trace 256 16 1
Hit rate: 99.985094%
548.exchange2_r_llc.mem_trace 512 4 1
Hit rate: 99.985094%
548.exchange2_r_llc.mem_trace 512 8 1
Hit rate: 99.985094%
548.exchange2_r_llc.mem_trace 512 16 1
Hit rate: 99.985094%
548.exchange2_r_llc.mem_trace 1024 4 1
Hit rate: 99.985094%
548.exchange2_r_llc.mem_trace 1024 8 1
Hit rate: 99.985094%
548.exchange2_r_llc.mem_trace 1024 16 1
Hit rate: 99.985094%
548.exchange2_r_llc.mem_trace 2048 4 1
Hit rate: 99.985094%
548.exchange2_r_llc.mem_trace 2048 8 1
Hit rate: 99.985094%
548.exchange2_r_llc.mem_trace 2048 16 1
Hit rate: 99.985094%

sample.mem_trace 128 4 0
Hit rate: 42.580000%
sample.mem_trace 128 8 0
Hit rate: 42.780000%
sample.mem_trace 128 16 0
Hit rate: 43.740000%
sample.mem_trace 256 4 0
Hit rate: 56.600000%
sample.mem_trace 256 8 0
Hit rate: 60.870000%
sample.mem_trace 256 16 0
Hit rate: 62.250000%
sample.mem_trace 512 4 0

Hit rate: 65.070000%
sample.mem_trace 512 8 0
Hit rate: 65.410000%
sample.mem_trace 512 16 0
Hit rate: 65.600000%
sample.mem_trace 1024 4 0
Hit rate: 65.580000%
sample.mem_trace 1024 8 0
Hit rate: 65.600000%
sample.mem_trace 1024 16 0
Hit rate: 65.600000%
sample.mem_trace 2048 4 0
Hit rate: 65.600000%
sample.mem_trace 2048 8 0
Hit rate: 65.600000%
sample.mem_trace 2048 16 0
Hit rate: 65.600000%
sample.mem_trace 128 4 1
Hit rate: 47.520000%
sample.mem_trace 128 8 1
Hit rate: 50.740000%
... (150 lines left)