Kristopher Lopez, Daniel Rodriguez, Darius Olega

Dr. Anup Das

ECEC 412: Modern Processor Design

5 November 2021

Project 2: Implementing a PC-Signature based Hit Predictor

Objective

The objective of this project is to evaluate the correlation between the reuse behavior of a cache block and its respective Program Counter (PC) that inserted it into the cache. A cache replacement policy is created based on the PC signature and compared to other established policies such as Least-Recently Used (LRU) and Least-Frequently Used (LFU).

Program Counter (PC) Signature

Program Counter (PC) Signature refers to the instructions that reference memory, and how cache references can be grouped as such. These instructions can then be hashed and stored as a parameter of each cache block, alongside a predictor value of whether or not it would be accessed again. This leads into the Signature-based Hit Predictor algorithm also discussed in the provided research.

Signature-based Hit Predictor (SHiP) involves predicting whether a cache line will be re-referenced based on a certain set of parameters, namely the outcome of the latest cache insertion and a history counter known as a Signature History Counter Table (SHCT) with how many times the block has been re-referenced. With this information, a predictor label is set for either distant or intermediate, with how far the next re-reference would potentially be. The figure below showcases the pseudocode of the SHiP algorithm as provided in the research paper.

```
if hit then
        cache_line.outcome = true;
        Increment SHCT[signature_m];
else
        if evicted_cache_line.outcome != true
                Decrement SHCT[signature_m];
        cache_line.outcome = false;
        cache_line.signature_m = signature;
        if SHCT[signature] == 0
             Predict distant re-reference;
        else
             Predict intermediate re-reference;
end if
```

Figure 1. Provided pseudo-code of SHiP Algorithm.

In order to implement this into the pre-existing code, additional parameters of the cache block object “outcome” and “prediction” are added, being a boolean value and integer determining predictor value, respectively. The instruction “PC” has already been included in the object.

Additionally, another struct SHCT is created, with the number of elements equal to the hash divider number utilized for hashing each program counter. Each element’s value refers to the number of calls related to their respective cache line, which determines the cache line’s predictor.

```
15    #define hash_divider 4096
```

```
51    // Replacement Policies
52    bool pcshp(Cache *cache, uint64_t addr, Cache_Block **victim_blk, uint64_t *wb_addr);
53
54    void set_arg_vals(unsigned c, unsigned a);
```

```
10    uint64_t SHCT[hash_divider];
```

Figure 2. Additional Functions and Definitions included in the Cache files. First Two are located in Cache.h, while SHCT is located in Cache.c.

```
82  bool accessBlock(Cache *cache, Request *req, uint64_t access_time)
83  {
84      bool hit = false;
85
86      uint64_t blk_aligned_addr = blkAlign(req->load_or_store_addr, cache->blk_mask);
87
88      Cache_Block *blk = findBlock(cache, blk_aligned_addr);
89
90      if (blk != NULL)
91      {
92          hit = true;
93
94          blk->outcome = true;
95          SHCT[blk->signature_m]+=1;
96          blk->prediction = 0;
97
98          if (req->req_type == STORE)
99          {
100             blk->dirty = true;
101         }
102     }
103
104     return hit;
105 }
```

Figure 3. Modified accessBlock() function. Note the changed outcome to True and incrementing the respective SHCT element.

```
107 bool insertBlock(Cache *cache, Request *req, uint64_t access_time, uint64_t *wb_addr)
108 {
109     // Step one, find a victim block
110     uint64_t blk_aligned_addr = blkAlign(req->load_or_store_addr, cache->blk_mask);
111
112     Cache_Block *victim = NULL;
113     bool wb_required = pcshp(cache, blk_aligned_addr, &victim, wb_addr);
114     assert(victim != NULL);
115
116     // Step two, insert the new block
117     uint64_t tag = req->load_or_store_addr >> cache->tag_shift;
118     victim->tag = tag;
119     victim->valid = true;
120
121     if (victim->outcome != true)
122     {
123         SHCT[victim->signature_m]-=1;
124     }
125     victim->outcome = false;
126     victim->signature_m = req->PC % hash_divider;
127     if (SHCT[victim->signature_m] == 0)
128     {
129         victim->prediction = 2;
130     } else
131     {
132         victim->prediction = 1;
133     }
134
135     if (req->req_type == STORE)
136     {
137         victim->dirty = true;
138     }
139
140     return wb_required;
141 //    printf("Inserted: %"PRIu64"\n", req->load_or_store_addr);
142 }
```

Figure 4. Modified insertBlock() Function. Note the if-statements starting at line 121 that follows the SHiP algorithm stated in Figure 1.

```
175  bool pcshp(Cache *cache, uint64_t addr, Cache_Block **victim_blk, uint64_t *wb_addr)
176  {
177      uint64_t set_idx = (addr >> cache->set_shift) & cache->set_mask;
178      //    printf("Set: %"PRIu64"\n", set_idx);
179      Cache_Block **ways = cache->sets[set_idx].ways;
180
181      // Step one, try to find an invalid block.
182      int i;
183      for (i = 0; i < cache->num_ways; i++)
184      {
185          if (ways[i]->valid == false)
186          {
187              *victim_blk = ways[i];
188              return false; // No need to write-back
189          }
190      }
191
192      // Step two, if there is no invalid block. Locate the LFU block
193      Cache_Block *victim = ways[0];
194      for (i = 1; i < cache->num_ways; i++)
195      {
196          if (ways[i]->prediction > victim->prediction)
197          {
198              victim = ways[i];
199          }
200      }
201
202      // Step three, need to write-back the victim block
203      *wb_addr = (victim->tag << cache->tag_shift) | (victim->set << cache->set_shift);
204
205      // Step three, invalidate victim
206      victim->tag = UINTMAX_MAX;
207      victim->valid = false;
208      victim->dirty = false;
209
210      *victim_blk = victim;
211
212      return true; // Need to write-back
213  }
```

Figure 5. Created PC Signature-based Hit Predictor Policy function. Follows similar ideas of LFU's implementation with the prediction value as the parameter.

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.

Similar results were reached regarding the correlation between cache size and hit rate, along with associativity value and hit rate, and the PCSHP policy performed at a similar level in comparison to the previous LRU and LFU replacement policies. This could be due to the implementation and how not every instruction set may appear the same.

The idea behind the predictor is that it may not always be correct. Because of this, an influx of instructions may raise the SHCT to a high amount, but if the cache line is never called again, the predictor would need to be fixed. Reuse of a cache line will set it to 0, which will make it hard to remove from the queue.

Appendix A: Table of Hit Rates

sample.mem_trace		assoc-size		
PCSHP		4	8	16
cache-size	128	46.990000%	50.910000%	51.110000%
	256	59.670000%	62.320000%	63.350000%
	512	65.220000%	65.420000%	65.600000%
	1024	65.600000%	65.600000%	65.600000%
	2048	65.600000%	65.600000%	65.600000%

531.deepsjeng_r_llc.mem_trace		assoc-size		
PCSHP		4	8	16
cache-size	128	67.847025%	68.833640%	69.103608%
	256	78.964951%	78.462533%	78.496586%
	512	84.970258%	84.282560%	83.420048%
	1024	90.522282%	88.437553%	87.109655%
	2048	93.165624%	91.752007%	90.197955%

541.leela_r_llc.mem_trace		assoc-size		
PCSHP		4	8	16
cache-size	128	42.500744%	47.744720%	47.128149%
	256	64.346592%	65.749138%	66.603545%
	512	80.385190%	79.681297%	78.127946%
	1024	90.292542%	87.377282%	85.745442%
	2048	96.011387%	95.966365%	94.371904%

548.exchange2_r_llc.mem_trace	assoc-size		
PCSHP	4	8	16
cache-size 128	99.949165%	99.970673%	99.985094%
256	99.984933%	99.985094%	99.985094%
512	99.985094%	99.985094%	99.985094%
1024	99.985094%	99.985094%	99.985094%
2048	99.985094%	99.985094%	99.985094%

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

sample.mem_trace 128 4
Hit rate: 46.990000%
sample.mem_trace 128 8
Hit rate: 50.910000%
sample.mem_trace 128 16
Hit rate: 51.110000%
sample.mem_trace 256 4
Hit rate: 59.670000%
sample.mem_trace 256 8
Hit rate: 62.320000%
sample.mem_trace 256 16
Hit rate: 63.350000%
sample.mem_trace 512 4
Hit rate: 65.220000%
sample.mem_trace 512 8
Hit rate: 65.420000%
sample.mem_trace 512 16
Hit rate: 65.600000%
sample.mem_trace 1024 4
Hit rate: 65.600000%
sample.mem_trace 1024 8
Hit rate: 65.600000%
sample.mem_trace 1024 16
Hit rate: 65.600000%
sample.mem_trace 2048 4
Hit rate: 65.600000%
sample.mem_trace 2048 8
Hit rate: 65.600000%
sample.mem_trace 2048 16
Hit rate: 65.600000%
531.deepsjeng_r_llc.mem_trace 128 4
Hit rate: 67.847025%
531.deepsjeng_r_llc.mem_trace 128 8
Hit rate: 68.833640%
531.deepsjeng_r_llc.mem_trace 128 16
Hit rate: 69.103608%
531.deepsjeng_r_llc.mem_trace 256 4
Hit rate: 78.964951%
531.deepsjeng_r_llc.mem_trace 256 8

Hit rate: 78.462533%
531.deepsjeng_r_llc.mem_trace 256 16
Hit rate: 78.496586%
531.deepsjeng_r_llc.mem_trace 512 4
Hit rate: 84.970258%
531.deepsjeng_r_llc.mem_trace 512 8
Hit rate: 84.282560%
531.deepsjeng_r_llc.mem_trace 512 16
Hit rate: 83.420048%
531.deepsjeng_r_llc.mem_trace 1024 4
Hit rate: 90.522282%
531.deepsjeng_r_llc.mem_trace 1024 8
Hit rate: 88.437553%
531.deepsjeng_r_llc.mem_trace 1024 16
Hit rate: 87.109655%
531.deepsjeng_r_llc.mem_trace 2048 4
Hit rate: 93.165624%
531.deepsjeng_r_llc.mem_trace 2048 8
Hit rate: 91.752007%
531.deepsjeng_r_llc.mem_trace 2048 16
Hit rate: 90.197955%
541.leela_r_llc.mem_trace 128 4
Hit rate: 42.500744%
541.leela_r_llc.mem_trace 128 8
Hit rate: 47.744720%
541.leela_r_llc.mem_trace 128 16
Hit rate: 47.128149%
541.leela_r_llc.mem_trace 256 4
Hit rate: 64.346592%
541.leela_r_llc.mem_trace 256 8
Hit rate: 65.749138%
541.leela_r_llc.mem_trace 256 16
Hit rate: 66.603545%
541.leela_r_llc.mem_trace 512 4
Hit rate: 80.385190%
541.leela_r_llc.mem_trace 512 8
Hit rate: 79.681297%
541.leela_r_llc.mem_trace 512 16
Hit rate: 78.127946%
541.leela_r_llc.mem_trace 1024 4

Hit rate: 90.292542%
541.leela_r_llc.mem_trace 1024 8
Hit rate: 87.377282%
541.leela_r_llc.mem_trace 1024 16
Hit rate: 85.745442%
541.leela_r_llc.mem_trace 2048 4
Hit rate: 96.011387%
541.leela_r_llc.mem_trace 2048 8
Hit rate: 95.966365%
541.leela_r_llc.mem_trace 2048 16
Hit rate: 94.371904%
548.exchange2_r_llc.mem_trace 128 4
Hit rate: 99.949165%
548.exchange2_r_llc.mem_trace 128 8
Hit rate: 99.970673%
548.exchange2_r_llc.mem_trace 128 16
Hit rate: 99.985094%
548.exchange2_r_llc.mem_trace 256 4
Hit rate: 99.984933%
548.exchange2_r_llc.mem_trace 256 8
Hit rate: 99.985094%
548.exchange2_r_llc.mem_trace 256 16
Hit rate: 99.985094%
548.exchange2_r_llc.mem_trace 512 4
Hit rate: 99.985094%
548.exchange2_r_llc.mem_trace 512 8
Hit rate: 99.985094%
548.exchange2_r_llc.mem_trace 512 16
Hit rate: 99.985094%
548.exchange2_r_llc.mem_trace 1024 4
Hit rate: 99.985094%
548.exchange2_r_llc.mem_trace 1024 8
Hit rate: 99.985094%
548.exchange2_r_llc.mem_trace 1024 16
Hit rate: 99.985094%
548.exchange2_r_llc.mem_trace 2048 4
Hit rate: 99.985094%
548.exchange2_r_llc.mem_trace 2048 8
Hit rate: 99.985094%
548.exchange2_r_llc.mem_trace 2048 16

Hit rate: 99.985094%