ASSIGMENT 1 CS423

Pramodh Gopalan
190933
CSE, Indian Institute of Technology Kanpur
{pramodh}@iitk.ac.in

Ayush Kumar
190213
CSE, Indian Institute of Technology Kanpur
{kumayu}@iitk.ac.in

How to run the code

After entering into the directory, you can simply execute the command `make` to compile the cache simulator. To run the simulator on any trace, say bzip2, we simply do:
`./cache bzip2.log_l1misstrace 2`
as described by the assignment, or you can alternatively do -

```
g++ -std=c++17 -O3 cache.cpp -o cache
./cache bzip2.log_l1misstrace 2
```

Question 1

The tables for the hit and miss of each section is shown in Tables 1-6 shown on the next page. Here are some observations about the results.

- Exclusive Caches perform better than Inclusive and NINE caches. Exclusive Caches only allow a block to be placed in L2 or L3 at a time. This causes an increase in the effective cache size of the Cache hierarchy as a whole, causing misses to be lesser when compared to NINE and Inclusive policies. In the latter, the effective capacity as a whole is smaller due to L2 being a full/partial subset of L3.
- NINE caches perform better than Inclusive Caches. When a block is evicted from L3 in Inclusive caches, it is also evicted from L2. This can cause undue misses when we try and access the same block again. However, when a block is evicted from L3 in NINE, it still allows for cache hits in L2, causing lesser misses depending on access pattern.
- Cache misses in L2 are the same in NINE and Exclusive for all cases. This is because of the L3 evictions being independant of L2 evictions in both these cases, so there is no difference in this aspect.
- In the case of programs like bzip2 and h264ref, the number of L3 misses of Exclusive cache is almost less than half of L3 misses of Inclusive and NINE caches. when we evict a block from L2 in Exclusive, we write it back to L3, thus allowing the opportunity for it to be fetched again into L2 when required. This when combined with the increased cache size in exclusive provides superior miss rates in some cases.

Question 2

We observe that the misses for the belady implementation are lower than those of LRU case by a significant margin in most of the cases. This is also somewhat expected since Belady knows the future access patterns of the code and

Inclusion Policy	L2 Misses	L3 Misses
Exclusive	5397576	889221
Inclusive	5398166	1446388
Nine	5397576	1445846

Table 1: Cache simulator on BZIP2

Inclusion Policy	L2 Misses	L3 Misses
Exclusive	3029809	1242824
Inclusive	3036461	1373402
Nine	3029809	1366248

Table 2: Cache simulator on GCC

Inclusion Policy	L2 Misses	L3 Misses
Exclusive	336724	159302
Inclusive	336851	170531
Nine	336724	170459

Table 3: Cache simulator on GROMACS

Inclusion Policy	L2 Misses	L3 Misses
Exclusive	1735322	300046
Inclusive	1743421	391226
Nine	1735322	376344

Table 4: Cache simulator on HMMER

Inclusion Policy	L2 Misses	L3 Misses
Exclusive	8815130	7220776
Inclusive	8820349	8207362
Nine	8815130	8205144

Table 5: Cache simulator on SPHINX

Inclusion Policy	L2 Misses	L3 Misses
Exclusive	965624	143681
Inclusive	969678	342146
Nine	965624	333583

Table 6: Cache simulator on H264REF

Program	Cold misses	Capacity misses	Conflict misses
bzip2	119753	1241648	84987
gcc	773053	596871	3478
gromacs	107962	61406	1163
hmmer	75844	301140	14202
sphinx	122069	8265179	5138782
h264ref	63703	272177	6266

Table 7: LRU policy on fully associative cache

Program	Cold misses	Capacity misses	Conflict misses
bzip2	119753	417083	909552
gcc	773053	166236	434113
gromacs	107962	35292	27277
hmmer	75844	77563	237779
sphinx	122069	2946511	-179886
h264ref	63703	47902	230541

Table 8: Belady policy on fully associative cache

knows which block would be accessed farthest in out trace. LRU works well when there is a high degree of temporal locality, which means that recently accessed pages are likely to be accessed again in the near future. However, it may not perform well in situations where there is a low degree of temporal locality, however belady would still ensure in that

case that the cache contains the blocks that would be accessed in the near future.
Assuming a case similar to ours, (with inclusive cache and L2 is 4 times smaller than L3). Consider the access pattern - a0, a1, a2, a3, a4, a5, a6, a7, a8, a0. with 1 set 2 assoc. L2 and 1 set 8 assoc L3, then on a8, LRU evicts a0 from L3 while Belady would know that since a0 has the least future access time, it should not get evicted. The only case when LRU's eviction could be good is when a0 is actually accessed late into the trace, but in that case, Belady would again evict a0. So, the performance of LRU would be as good as Belady here. NOTE: In our implementation, we fix the trace to the L2 and not the L3.

References