2016314827 김규연

Cache Simulator Document

0. 파일 설명 및 실행 방법

- C++ 코드를 작성하였다. 컴파일은 Makefile에 이미 구현해두었으므로 make 커맨드만 입력하면 알아서 컴파일이 완료된다. 소스 코드는 cache.cpp 내에 구현되어 있다.

- 실행 커맨드는 ./cache [file_name] 이다. 만약 trace1.din을 실행시키고 싶다면 ./cache trace1.din을 입력하면 된다.

- 실행 후에는 문제에 제시된 바에 따라 주어진 표의 형태로 4 x 5의 output이 뜨게 된다.

1. class Trace

Trace class는 trace1.din, trace2.din 내부의 인풋을 받는 형식이다. Trace 내에는 address의 type과, 해당 address가 저장되어있다.

2. class CacheBlock

CacheBlock은 말 그대로 각 cache의 block을 나타낸다. CacheBlock 내에는 address의 tag가 저장되어있고, 해당 데이터가 dirty한지 여부가 저장되어있다.

3. global variables

input을 담아두는 traces_1이라는 vector와 miss ratio나 write count의 표를 구성하는 array 전역 변수, 시행할 cache size, block size, associativity가 선언되어있다.

4. main function

파일을 열어 traces_1 vector에 저장하고, cache_proc 함수로 넘어가게 된다.

5. cache_proc function

특정 cache size, 특정 block size, 특정 associativity 마다 iteration을 도는 부분이다. 매 iteration 마다 inclusive, exclusive 두 가지 버전으로 도는 것을 볼 수 있다. 모든 경우에 대해 cache simulation을 돌린 후에, 최종적인 결과를 표로 출력하는 부분도 구현되어 있다.

6. cache_simulate_in function-

6.1 L2 cache를 inclusive로 시행했을 때 실행되는 함수이다.

6.2 각 set을 list를 활용하여 queue로 구현하였다.

6.3 우선 L1, L2의 set 개수를 계산하여 L1_nset, L2_nset에 담았다.

6.4 그 후에 L1 I-cache, L1 D-cache, L2 cache의 내부에 있는 set를 list의 array로 만들었다. 이 list는 각각 cache block을 담을 것이며, queue 형태로 활용할 것이다.

6.5 이제 traces_1 vector에 담긴 하나 하나의 데이터마다 iteration을 돈다. 주소 값을 가져와서 L1, L2 각각의 index값을 get_index 함수를 통해 계산한다.

6.6 이제 각 address type에 따라 과정이 나누어진다.

6.7 모든 과정이 끝나고 해당 miss rate와 memory write 데이터를 해당 하는 cache size, block size, associativity에 찾아가 저장해준다.

6-A. cache_simulate_in (type = 0)

우선 L1의 D-cache에 접근하여 hit인지 miss인지 판단한다.

1. L1 D-cache hit일 경우, LRU를 업데이트 하기 위해 해당 주소를 지운 후에 다시 list queue의 맨 뒤로 삽입하는 식으로 업데이트 하면 된다. 이렇게 하면 큐의 맨 앞에는 LRU address가 남게 된다. 그 대신 각 queue의 size는 associativity의 수로 제한 된다. size가 associativity를 넘어가게 되면, LRU에 해당하는 block을 지우고 새로운 block을 넣어주게 된다.

2. L1 D-cache miss 일 경우, L2 cache에 이어서 찾는다. 이 때 L1 cache에 해당 block을 넣어 업데이트한다.

3. 만약 L2 cache hit 일 경우, L1과 같은 방법으로 L2 cache를 LRU 업데이트 해준다.

4. 만약 L2 cache miss가 났을 경우, L2에도 해당 주소를 cache에 넣어줘야 한다. 이 때 해당 list가 꽉 차 있어서 LRU에 따라 queue의 맨 앞의 원소를 지운다면, L1에서 또한 지워야 하므로, L2 block의 tag와 index를 L1에 맞게 거꾸로 계산해준다. 그 후에 해당 L1 block의 index와 tag에 찾아가 지운다. 이 때 L1 D-cache, L1 I-cache를 모두 탐색하여 지우도록 한다.

5. 만약 L2에서 탈락된 block의 dirty bit가 true라면, memory에 write을 해야 하므로 카운트를 하나 늘려준다.

6. L1에서 탈락된 block을 L2에 따로 넣어줘야 하는데, 이미 L1에 있는 block은 L2에도 있기 때문에 또 다른 행동을 할 필요가 없다.

6-B. cache_simulate_in (type = 1)

사실상 read와 write의 차이이기 때문에 6-A와 거의 유사하다.

유일한 차이점은 L1 miss, L2 hit일 때 일어난다. L2 hit이 일어나면, 해당 block의 dirty bit을 true로 바꿔주는 것 뿐이다.

6-C. cache_simulate_in (type = 2)

D-cache와 작동 원리는 거의 유사하다.

D-cache와 차이점이 있다면 dirty bit에 대해 신경을 쓰지 않는다는 점이다.

7. cache_simulate_ex function

Inclusive 대신 Exclusive policy를 따랐다.

우선 L1의 D-cache에 접근하여 hit인지 miss인지 판단한다.

1. L1 D-cache hit일 경우, LRU를 업데이트 하기 위해 해당 주소를 지운 후에 다시 list queue의 맨 뒤로 삽입하는 식으로 업데이트 하면 된다. 이렇게 하면 큐의 맨 앞에는 LRU address가 남게 된다.

2. L1 D-cache miss 일 경우, L2 cache에 이어서 찾는다. 이 때 L1 cache에 block을 넣어주게 되는데, 해당 set이 꽉 차서 탈락되는 block이 발생한 경우 해당 block의 tag와 index를 이용하여 L2 cache의 index와 tag 값에 맞도록 거꾸로 계산해 준 후, 해당 장소로 찾아가 L2 cache에 복사하여 넣어준다.

3. 만약 L2 cache hit 일 경우, L1에만 block이 존재해야 하므로 hit된 해당 블록을 삭제한다.

4. 만약 L2 cache miss가 났을 경우 L1에만 block을 넣어줘야 한다. 2번 과정에서 L1 cache를 업데이트 했으므로 L2 cache에서는 별다른 과정이 요구되지 않는다.

8. output table (trace1.din)

trace1.din을 실행 시켜 나온 output table 이다.

trace1.din <L1 I-cache Miss Ratio (block size = 16B)>					
LRU / 16	1024	2048	4096	8192	16384
Direct	0.1757	0.145	0.1125	0.0769	0.0452
2-way	0.1689	0.1367	0.0989	0.0632	0.0334
4-way	0.1692	0.1345	0.096	0.057	0.0268
8-way	0.1697	0.1356	0.0928	0.0548	0.0249

trace1.din <L2 Inclusive Miss Ratio (block size = 16B)>					
LRU / 16	1024	2048	4096	8192	16384
Direct	0.2018	0.2517	0.3278	0.4133	0.5486
2-way	0.2232	0.2861	0.387	0.5228	0.7197
4-way	0.2294	0.2967	0.399	0.5839	0.8597
8-way	0.2306	0.2962	0.4142	0.5943	0.9311

trace1.din <L1 I-cache Miss Ratio (block size = 64B)>					
LRU / 16	1024	2048	4096	8192	16384
Direct	0.0756	0.0624	0.0498	0.0359	0.0208
2-way	0.0718	0.0593	0.0444	0.0304	0.0166
4-way	0.0712	0.0591	0.0434	0.0279	0.0138
8-way	0.0711	0.0597	0.043	0.0275	0.0124

trace1.din <L2 Inclusive Miss Ratio (block size = 64B)>					
LRU / 16	1024	2048	4096	8192	16384
Direct	0.2019	0.2631	0.3634	0.4294	0.523
2-way	0.2379	0.3216	0.4462	0.5683	0.677
4-way	0.2559	0.3368	0.4628	0.638	0.8136
8-way	0.2586	0.3453	0.4751	0.646	0.9166

trace1.din <L1 D-cache Miss Ratio (block size = 16B)>					
LRU / 16	1024	2048	4096	8192	16384
Direct	0.1991	0.1409	0.0785	0.0501	0.0303
2-way	0.1504	0.0951	0.0608	0.0354	0.0226
4-way	0.131	0.0843	0.0544	0.0313	0.019
8-way	0.1255	0.0794	0.0529	0.029	0.0183

trace1.din <L2 Exclusive Miss Ratio (block size = 16B)>					
LRU / 16	1024	2048	4096	8192	16384
Direct	0.6608	0.6549	0.6428	0.6707	0.7303
2-way	0.642	0.6325	0.6445	0.6867	0.7973
4-way	0.6309	0.6264	0.6422	0.7018	0.8615
8-way	0.6279	0.6221	0.6453	0.7107	0.8979

trace1.din <L1 D-cache Miss Ratio (block size = 64B)>					
LRU / 16	1024	2048	4096	8192	16384
Direct	0.2225	0.1554	0.0816	0.0496	0.025
2-way	0.166	0.1027	0.0591	0.0311	0.0155
4-way	0.1405	0.0883	0.0485	0.0253	0.0115
8-way	0.136	0.0809	0.0451	0.0238	0.0108

trace1.din <L2 Exclusive Miss Ratio (block size = 64B)>					
LRU / 16	1024	2048	4096	8192	16384
Direct	0.815	0.7899	0.7663	0.7768	0.8039
2-way	0.8151	0.7836	0.7591	0.7818	0.8399
4-way	0.7957	0.7998	0.7857	0.7942	0.8834
8-way	0.788	0.7859	0.8266	0.8441	0.9332

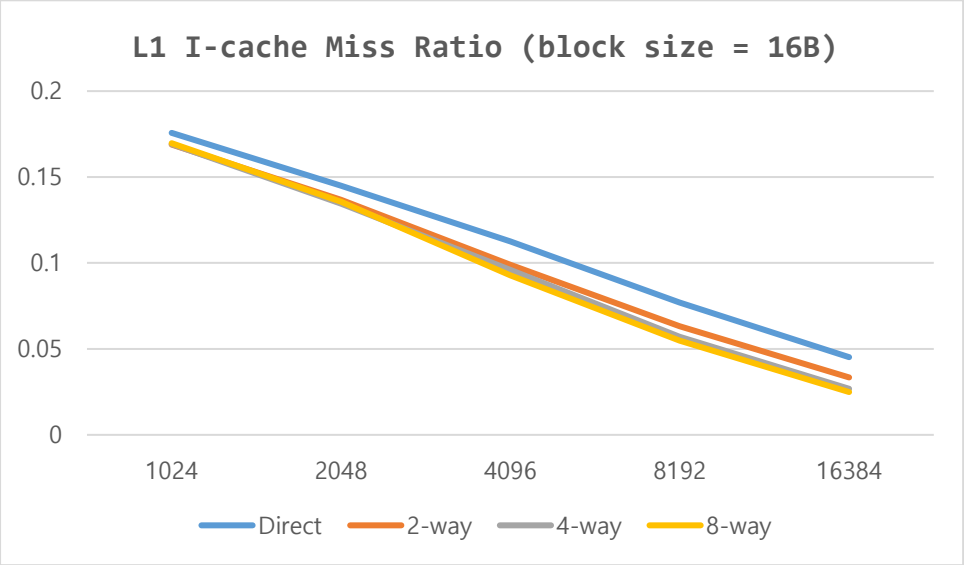
trace1.din <Inclusive Memory Block Writes (block size = 16B)>					
LRU / 16	1024	2048	4096	8192	16384
Direct	4530	4055	3130	1821	958
2-way	4477	3973	3124	1503	546
4-way	4291	3949	3007	1398	337
8-way	4248	3901	2961	1343	188

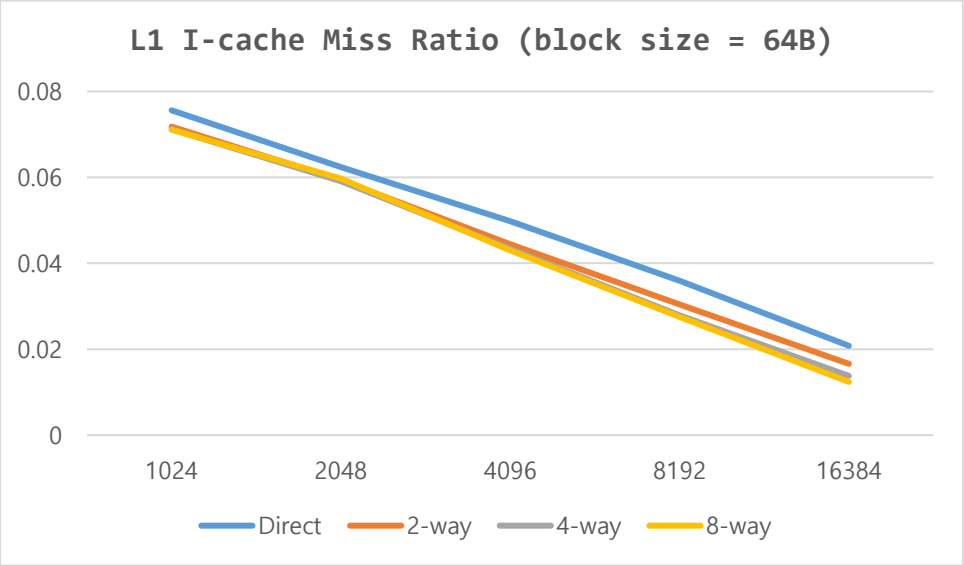
trace1.din <Inclusive Memory Block Writes (block size = 64B)>					
LRU / 16	1024	2048	4096	8192	16384
Direct	2783	2533	1931	1206	572
2-way	2744	2540	1936	976	350
4-way	2720	2504	1868	824	181
8-way	2665	2521	1862	821	96

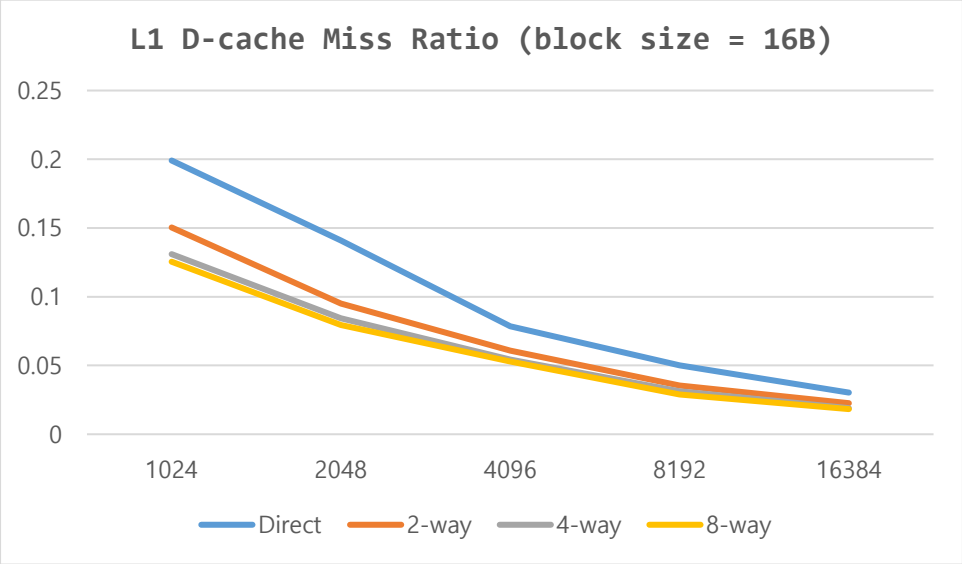
trace1.din <Exclusive Memory Block Writes (block size = 16B)>					
LRU / 16	1024	2048	4096	8192	16384
Direct	7357	5501	3781	2372	1494
2-way	6085	4272	2992	1886	1231
4-way	5763	3759	2704	1711	1110
8-way	5371	3530	2627	1717	1103

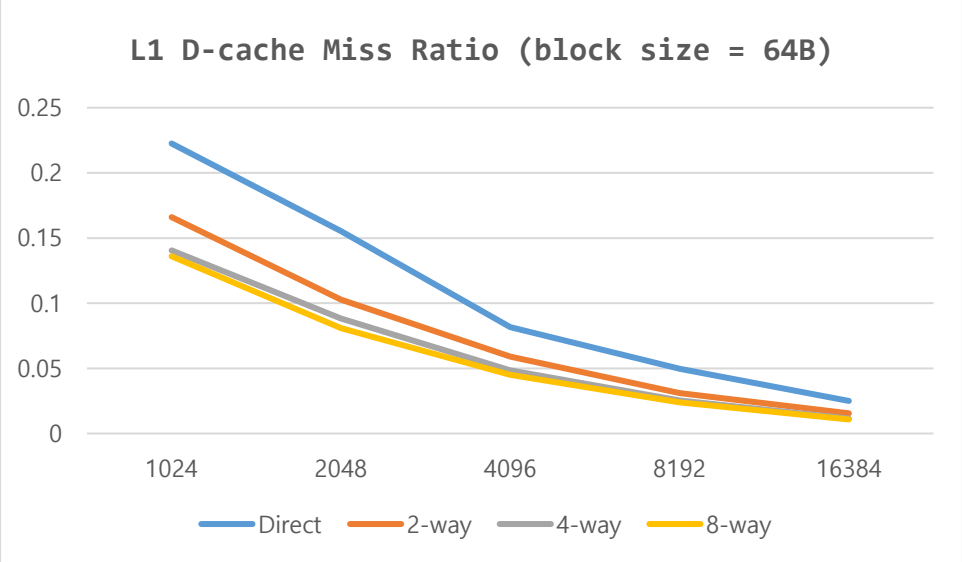
trace1.din <Exclusive Memory Block Writes (block size = 64B)>					
LRU / 16	1024	2048	4096	8192	16384
Direct	5923	4350	2726	1485	849
2-way	4682	2823	1782	925	502
4-way	4115	2216	1217	718	378
8-way	3838	2043	1048	624	366

9. Graphs

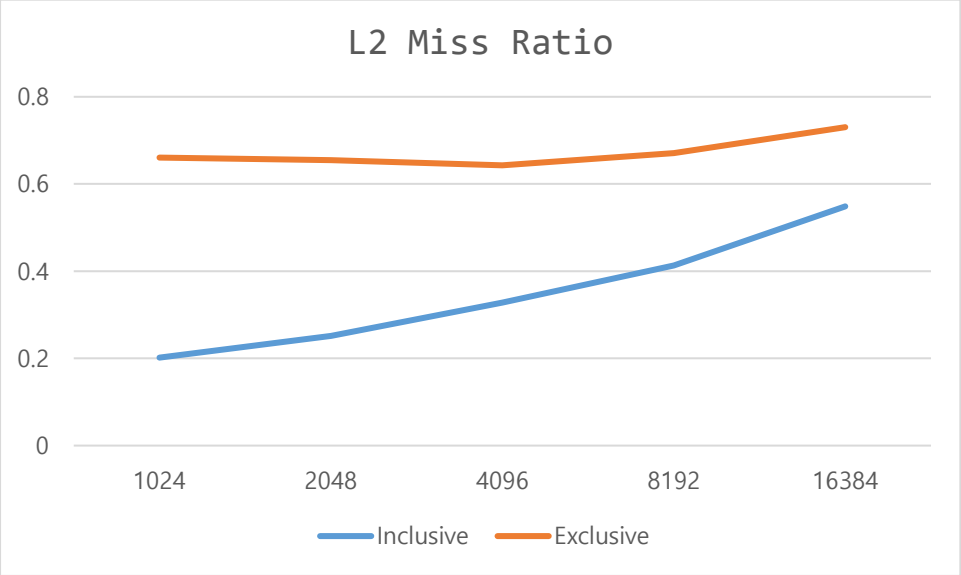

L1 I-cache Miss Ratio (block size = 16B)
0.2
0.15
0.1
0.05
0
1024
2048
4096
8192
16384
Direct
2-way
4-way
8-way

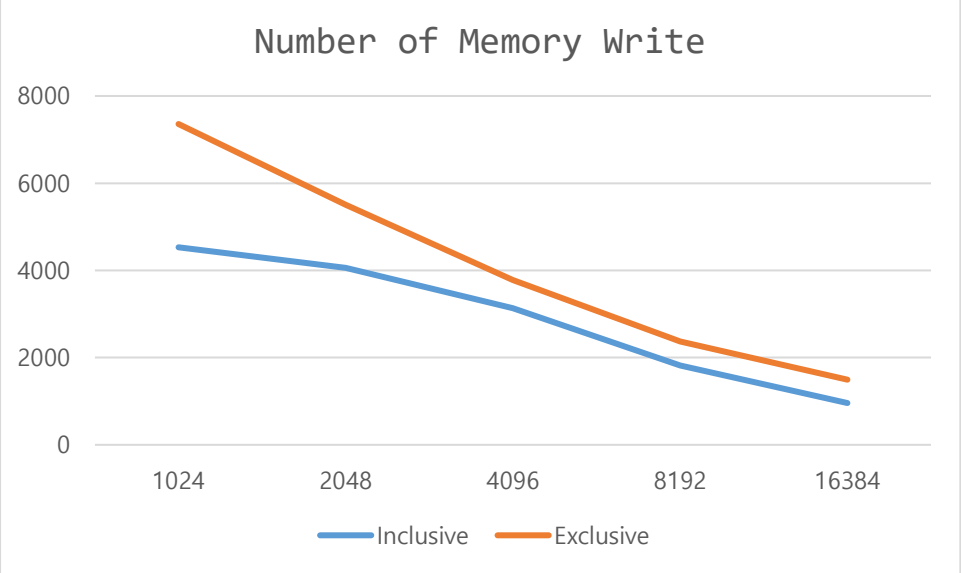

L1 I-cache Miss Ratio (block size = 64B)
0.08
0.06
0.04
0.02
0
1024
2048
4096
8192
16384
Direct
2-way
4-way
8-way





I-cache 에서나 D-cache에서나 associativity가 늘어날수록 miss ratio가 감소하는 양상을 띤다. 또한, cache size가 늘어날수록 또한 miss ratio가 계속 감소하고 있는 일관된 양상을 볼 수 있다. Block size는 64B 일 때가 16B 일 때보다 대체로 miss ratio가 낮음을 알 수 있다.





Direct mapped 일 때를 기준으로 뽑은 차트이다.

Exclusive 일 때보다 Inclusive policy를 따를 때 miss ratio가 낮음을 확인할 수 있었다. 또한, Memory Write의 경우에도 Inclusive policy를 따를 때가 Exclusive policy를 따를 때보다 memory write 횟수 또한 적음을 확인할 수 있었다.