Project 4: Multi-level Cache Model and Performance Analysis

201811118 이 구

1. Code Flow

구현상의 편의를 위해 valid_bit, dirty_bit, ref_cnt, tag, block_addr로 구성된 Cache_Block 구조체를 만들어서 사용하였다. valid_bit는 Cache가 완전히 비어 있는 초기 상태에서는 0의 값을 가지며, cache block들이 caching 되면 1의 값을 갖는다. L2 Cache에서 특정 cache block이 evict 될 때, L1 Cache에 동일한 cache block이 존재한다면 L1 cache에 존재하는 해당 cache block도 함께 evict 될 수 있도록 하였다. dirty_bit는 cache block에 write 한 경우 1, 그렇지 않은 경우 0의 값을 갖는다. ref_cnt는 reference counter를 의미한다. 큰 값을 가질수록 최근에 참조된 cache block임을 의미한다. 모든 tracefile의 total access가 20,000,001임을 확인하였고, 따라서 별도의 구현 없이 충분히 큰 자료형을 이용하여 access할 때 마다 1씩 증가하는 값을 ref_cnt로 사용하면 정상적으로 동작하는 lru를 구현할 수 있다. tag의 경우, index(set)가 겹칠 때 identify 하기 위한 용도로 사용하였고, block_addr은 L2 Cache에서 evict 되는 cache block이 L1 Cache에도 존재하는지 확인하기 위해 사용하였다.

제공된 tracefile은 64-bit physical address space를 이용하므로, unsigned long long 자료형을 이용하여 연산을 수행하였다. block size가 64(=2^6) byte인 경우, address에서 하위 6개의 bit를 날린 주소를 이용하였다. index의 경우, block address를 cache에 존재하는 set의 갯수로 나눈 나머지 값을 이용하였고, cache에 존재하는 set의 갯수는 cache의 전체 크기, block 하나의 크기, associativity 값을 이용하여 계산할 수 있다. set의 갯수를 알고 있다면, index로 사용해야 하는 bit의 수 역시 알 수 있다. (ex. set의 수가 16(=2^4)인 경우, 4개의 bit를 이용하여 indexing) 따라서, tag로 사용하는 값은 block address에서 indexing에 사용하는 bit를 제외한 나머지 모든 정보를 사용한다.

특정 주소에 access 하고자 할 때, 발생할 수 있는 경우 총 3가지이고, 각각의 경우에 수행한 동작은 아래와 같다.

1) L1 Cache hit
index를 valid bit와 tag를 이용하여 target address와 같은 지 확인한다. 원하는 cache block이 L1 Cache에 존재하는 경우 해당 cache block의 ref_cnt를 update한다.

2) L1 Cache miss & L2 Cache hit
index를 valid bit와 tag를 이용하여 target address와 같은 지 확인한다. 원하는 cache block이 L1 Cache에 존재하지 않는 경우, 해당 cache block이 L2 Cache에 존재하는지 확인한다. L2 Cache에 원하는 cache block이 존재한다면, 이를 L1 Cache에 caching해야 한다. 이때, valid 하지 않은 cache block이 존재한다면 우선적으로 evict하고, 그렇지 않은 경우 replacement policy에 의한 victim을 선정한다. 이때, victim의 dirty bit가 1이라면, dirty eviction, 그렇지 않은 경우 clean eviction으로 체크한다.

3) L1 Cache miss & L2 Cache miss
index를 valid bit와 tag를 이용하여 target address와 같은 지 확인한다. 원하는 cache block이 L1 Cache에 존재하지 않는 경우, 해당 cache block이 L2 Cache에 존재하는지 확인한다. L2 Cache에 원하는 cache block이 존재하지 않는다면, L2 Cache와 L1 Cache 모두에 caching 해야한다. 이 과정은 앞서 설명한 것과 같이, valid하지 않은 cache block을 우선적으로 선택하며, valid하지 않은 cache block이 존재하지 않는 경우 replacement policy에 따라 victim을 선정한다.

2. Compile & Execution

제출한 압축 파일은 Multi-level_Cache_Model.cpp, Makefile, Report, assets로 구성되어있다. 압축을 해제한 폴더에서 아래의 명령어를 통해 컴파일 및 실행할 수 있다. assets의 경우, 그래프를 그리기 위해 사용한 결과와 그래프의 원본 이미지 파일로 구성되어있다.

```
$ make
$ ./runfile <-c capacity> <-a associativity> <-b block_size> <-lru or -random> <trace file>
```

3. Environment

코드는 Ubuntu 22.04.3 LTS 환경에서 c++로 작성되었으며, g++ 9.5.0을 사용하여 컴파일하였다. 테스트는 Windows Subsystem for Linux (Ubuntu 20.04) 환경에서 g++ 9.4.0을 사용하여 진행하였다.

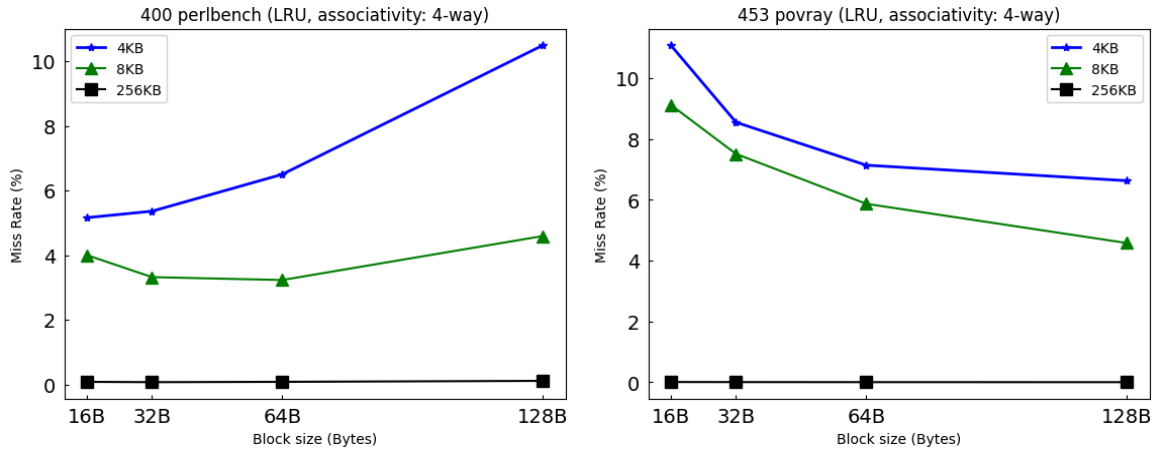
4. Performance Analysis

Cache performance 분석은 크게 block size, associativity, replacement policy에 따른 miss rate의 변화에 대해 진행하였다. Miss rate의 경우, read/write miss rate는 program behavior에 따라 그 비율이 변화하므로, fair한 비교를 위해 L2 miss 전체(L2 read miss + L2 write miss)를 access 횟수(20,000,001)로 나눈 값에 100을 곱한 값을 miss rate로 사용하였다. 주어진 benchmark들 중, cache의 특성을 잘 보여주는 benchmark들에 대한 결과만을 첨부하였다. 그래프에 표시한 모든 cache spec은 L2 Cache에 대한 것으로, L1 Cache의 spec은 지시사항와 같다. (ex. L2 Cache 16-way, 전체 Cache 크기 4KB, block size 64B인 경우, L1 Cache의 spec은 4-way, 전체 Cache 크기 1KB, block size 64B이다.)

4.1. Block Size

Block size에 따른 miss rate의 변화를 확인하기 위해, 두 개의 benchmark(400 perlbench, 453 povray)를 수행하였고, 그 결과는 아래와 같다. 이때, associativity는 4-way, replacement policy로는 LRU를 사용하였다. Legend는 전체 Cache 크기를 의미한다.

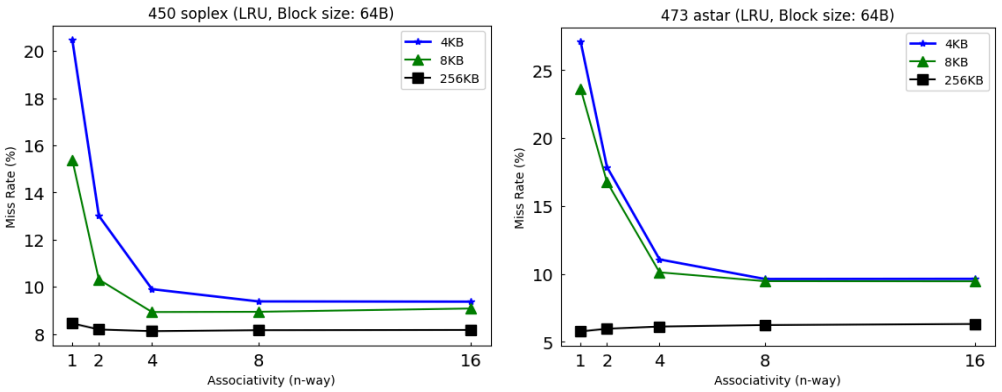
Block size는 spatial locality를 얼마나 활용할 것인지를 결정하는 요소이다. Block size가 커질 수록, spatial locality를 잘 활용할 수 있게 되고, cache에 들어가는 전체 block의 수가 작아지므로 tag strorage의 크기가 작아진다는 장점이 있다. 하지만, cache에 들어갈 수 있는 전체 block의 수가 줄어드므로 temporal locality를 잘 활용하기 어렵다. 또, next level cache에서 block을 load하는데 필요한 시간 역시 함께 증가한다. 이러한 이유로, program behavior에 따라 가장 낮은 miss rate를 보이는 block size가 달라지는 것을 확인할 수 있다.



4.2. Associativity

Associativity에 따른 miss rate의 변화를 확인하기 위해, 두 개의 benchmark(450 soplex, 473 astar)를 수행하였고, 그 결과는 아래와 같다. 이때, block size는 64B, replacement policy로는 LRU를 사용하였다. Legend는 전체 Cache 크기를 의미한다.

Associativity는 miss rate를 줄이기 위해 사용하는 개념으로, direct mapped cache에서는 하나의 index에 하나의 cache block들만이 들어갈 수 있었지만, n-way associative cache에서는 하나의 index에 n개의 cache block이 들어갈 수 있다. 이로 인해, 큰 associativity를 갖는 cache에서는 낮은 miss rate를 보이며, 이는 아래의 그래프에서도 확인할 수 있다. 하지만, associativity가 커질수록, 같은 set 안에서 다시 원하는 cache block을 찾는데 필요한 시간(access latency)이 길어지게 된다. 따라서, 낮은 miss rate를 보인다고 해서 높은 associativity를 갖는 cache가 높은 성능을 보인다는 것을 보장할 수는 없다.



4.3. Replacement Policy

Replacement policy에 따른 miss rate의 변화를 확인하기 위해, 하나의 benchmark(483 xalancbmk)를 수행하였고, 그 결과를 두 가지 방식(block size에 따른 miss rate, associativity에 른 miss rate)으로 시각화하였다. Replacement policy란 cache miss가 발생해서 새로운 cache block을 가져올 때, 어떤 cache block을 내보낼 지 결정하는 방식을 의미한다. LRU(Least Recently Used)는 최근에 사용되지 않은 cache block을 victim으로 선정하여 evict하는 방식이고, Random은 random하게 victim을 선정하여 evict하는 방식이다. 앞서 4.1에서 이야기 하였듯이, 프로그램은 temporal locality와 spatial locality를 갖는다. 따라서, 최근에 사용된 데이터는 다시 접근해야 할 가능성이 높으므로 LRU 방식을 사용하는 것이 reasonable 하므로, Random에 비해 낮은 miss rate를 보여야 한다. 또, associativity가 1인 cache는 하나의 index에 하나의 cache block만을 가지며, 따라서 replacement policy와 관계 없이 항상 같은 결과를 보여야 한다. 이를 아래 그래프에서 확인할 수 있다.