과제 #2 – Cache Simulator 구현

김수환 (201510743)
전북대학교 컴퓨터공학부
suwhan77@naver.com

요약

Ubuntu 리눅스 기반 C/C++로 작성
메모리 트레이스 파일과 캐시 설정에 기반하여 hit rate를 계산하는 Cache Simulator

1. Cache Simulator

1-1. 실습 프로그램의 구성 및 동작 원리

캐시 설정과 memory trace 파일을 받아서 hit rate를 계산하는 프로그램

주어진 memory trace와 cache 설정 (cache 전체 크기, block 크기, associativity)에 대해 hit rate를 출력한다.
잘못된 캐시 설정이나 trace파일에 대해서는 에러 메시지를 출력한다.

컴파일 방법

```
g++ lab2_201510743_김수환.cc –o simple_cache_sim
```

사용 방법

```
ex) simple_cache_sim swim.trace 1024 16 4
```

위와 같이, 4개의 입력이 필요하다 각 입력들은 다음과 같다

- 메모리 트레이스 파일
- Cache의 전체 바이트 단위 크기
- Cache block 하나의 바이트 단위 크기
- Associativity, 즉 Direct mapped cache의 경우에는 1이며, 4-way set associativity cache는 4가 된다. 일반적으로 N-way associativity cache에서 N 값을 입력한다

핵심 코드 설명

캐시 슬롯 구조체

```
typedef struct CacheSlot_t
{
    bitset<ADDRESS_BITS> tagBits;       // 슬롯의 Tag 비트
    bitset<1> validBit;                 // valid 비트
} CacheSlot;
```

tagBits 를 통해 캐시 슬롯에 들어있는 정보를 식별하고 validBit 를 통해 유효한 정보인지 판단한다.

캐시 인덱스 슬롯 구조체

```
typedef struct CacheIndexSlot_t
{
    bitset<ADDRESS_BITS> indexBits; // 인덱스 슬롯의 Index 비트
    list<int> LRU_list;     // LRU 슬롯을 찾기 위한 리스트
```

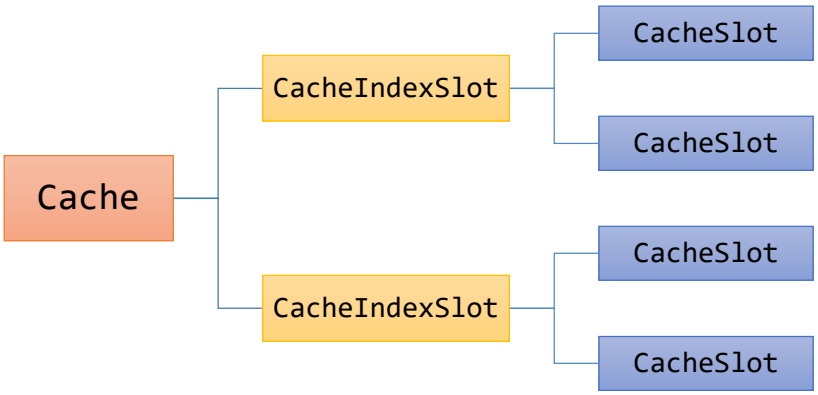
```
    int slotCount;          // 캐시 슬롯 개수
    CacheSlot *slotList;    // 캐시 슬롯 리스트
} CacheIndexSlot;
```

indexBits 를 통해 몇번째 인덱스의 세트인지 식별하고 slotList 에 캐시 슬롯들을 저장하고 있다. LRU_list 에 최근에 사용된 캐시 슬롯을 넣으면서 가장 오래전에 쓰인 슬롯을 찾는다. (리스트의 맨 뒤에 있는 슬롯이 가장 오래전에 쓰인 슬롯)

캐시 시뮬레이터 전용 캐시 구조체

```
typedef struct Cache_t
{
    int size;           // 캐시 크기
    int blockSize;      // 블럭 크기
    int associativity;  // associativity
    int tagBits;        // Tag 비트 수
    int indexBits;      // Index 비트 수
    int offsetBits;     // Offset 비트 수
    int accessCount;    // 캐시 접근 횟수
    int hitCount;       // Hit count
    CacheIndexSlot *indexSlotList;  // 캐시 세트 리스트
} Cache;
```

캐시 설정 값들과 TIO 정보를 가지고있고 캐시의 메모리 접근 횟수와 Hit count 를 저장하고 있다. indexSlotList 에 캐시 세트들을 저장하고있다.



<- 캐시 구조 그림

main 함수 실행 흐름 설명

1. argv 에서 입력 인자들을 받아오고 입력받은 이름의 trace 파일을 연다 (예외처리 포함)
2. 새로운 Cache 구조체를 생성하고 입력받은 캐시 설정으로 초기화 한다: **initialize()**
3. 캐시의 Tag, Index, Offset 비트 수를 출력한다
4. 트레이스 파일에서 한 줄씩 읽어와 읽어온 메모리 주소로 캐시에 접근: **access()**
5. 트레이스 파일 읽기가 끝나면 시뮬레이션 결과(AccessCount, HitCount, HitRate)를 출력

initialize 함수 실행 흐름 설명

1. 입력받은 캐시 설정에 기반하여 Tag, Index, Offset 비트 수를 계산한다
2. 세트 개수 만큼 CacheIndexSlot 과 세트 당 슬롯 개수만큼 CacheSlot 을 만들고 초기화한다

access 함수 실행 흐름 설명

1. 캐시의 accessCount 를 1 회 증가

2. 입력받은 주소값에서 Tag 비트와 Index 비트를 추출
3. 캐시에서 입력받은 주소의 Index 비트에 해당하는 CacheIndexSlot 을 가져옴
4. 해당 CacheIndexSlot 의 CacheSlot 들 에서 Tag 비트가 일치하는 슬롯을 찾기 시작
   5.1. CacheSlot 의 validBit 가 0 이면 **Compulsory Miss** 이므로 validBit 를 1 로 변경하고 새로운 Tag 비트를 설정, LRU_list 맨 앞에 해당 슬롯 번호를 추가, access 함수 종료
   5.2. CacheSlot 의 Tag 비트와 입력받은 주소의 Tag 비트가 일치하면 **Hit** 이므로 Cache 의 hitCount 를 1 만큼 증가, LRU_list 에서 해당 슬롯을 맨 앞으로 이동, access 함수 종료
   5.3. 5.1 번과 5.2 번에 해당 하지 않는 경우에는 다음 CacheSlot 과 5.1 번으로 이동, 더 이상 CacheSlot 이 없으면 6 번으로 이동
6. Tag 비트가 일치하는 CacheSlot 이 없으므로 **Conflict Miss** 발생, LRU_list 에서 맨 뒤에 있는 슬롯 번호가 가장 오래 전에 쓰인 슬롯의 번호이므로 해당 슬롯의 Tag 비트를 입력받은 Tag 비트로 덮어 쓰고 해당 슬롯의 번호를 LRU_list 의 맨 앞으로 이동

1-2. 결과

트레이스 파일: twolf.trace, 캐시 크기: 1024, 블록 크기: 16, associativity: 4 일때 실행 결과

```
supernova@ubuntu:~/CA$ simple_cache_sim twolf.trace 1024 16 4
tag: 24 bits
index: 4 bits
offset: 4 bits
Result: total access 482824, hit 460174, hit rate 0.95
supernova@ubuntu:~/CA$
```

트레이스 파일: twolf.trace, 캐시 크기: 8192, 블록 크기: 4, associativity: 8 일때 실행 결과

```
supernova@ubuntu:~/CA$ simple_cache_sim twolf.trace 8192 4 8
tag: 22 bits
index: 8 bits
offset: 2 bits
Result: total access 482824, hit 476489, hit rate 0.99
supernova@ubuntu:~/CA$
```

트레이스 파일: swim.trace, 캐시 크기: 4096, 블록 크기: 32, associativity: 4 일때 실행 결과

```
supernova@ubuntu:~/CA$ simple_cache_sim swim.trace 4096 32 4
tag: 22 bits
index: 5 bits
offset: 5 bits
Result: total access 303193, hit 292760, hit rate 0.97
supernova@ubuntu:~/CA$
```

1-3. 결론

유효한 메모리 트레이스 파일과 캐시 설정들을 입력하면 정상적으로 Cache simulator의 hit rate결과가 출력된다.