Project4

-Pipelined + Cache MIPS-

모바일시스템공학과

32212061

변윤성

Left days 0

1. Intro

2. Concepts

2.1 Memory Hierarchy

2.2 Locality

2.2.1 Temporal Locality

2.2.2 Spatial Locality

2.3 Cache Memory

2.4 Cache Mapping Strategies

2.4.1 Direct-Mapped Cache

2.4.2 Fully-Associative Cache

2.4.3 Set-Associative Cache

2.5 Cache Management Policies

2.5.1 Replacement Policy

2.5.2 Write Policy

2.6 Instruction Cache and Data Cache

2.6.1 Instruction Cache (I-Cache)

2.6.2 Data Cache (D-Cache)

3. Implementation

3.1 cache.h

3.1.1 #define

3.1.2 struct

3.2 cache.c

3.2.1 init_cache

3.2.2 Random / FIFO / SCA

3.2.3 update_lru / find_lru_victim

3.2.4 access_d_cache / access_i_cache

4. Result

4.1 캐시 연관도

4.2 캐시 교체 정책

4.3 Mapping

5. Feelings

1. Intro

컴퓨터 구조의 마지막 프로젝트의 주요 목표는 캐시 구조를 이해하고 성능을 분석하며 다양한 캐시 구성에 따른 캐시 히트/미스 비율과 평균 메모리 접근 시간(AMAT)을 비교하는 것이다. 이를 위해 본 보고서에서는 캐시의 동작 원리와 다양한 정책들을 먼저 살펴보고 직접 구현한 캐시 시뮬레이터의 구조와 실행 결과를 바탕으로 성능을 상세히 평가 및 분석하고자 한다.

2. Concepts

2.1 Memory Hierarchy

현대 컴퓨터 시스템의 CPU 처리 속도는 메모리 접근 속도보다 월등히 빠르다. 이 속도 차이는 시스템 전체 성능을 저하시키는 병목 현상을 유발한다. 이 문제를 해결하기 위해 메모리 계층 구조를 사용한다. 메모리 계층은 CPU에 가까울수록 더 빠르고 작은 저장장치를 배치한다. 반대로 멀어질수록 더 느리고 큰 저장장치를 둔다. 상위 계층은 단위 용량당 비용이 비싸고 하위 계층은 저렴하다. 대표적으로 레지스터>캐시>메인 메모리>보조기억장치 순으로 구성된다. 이 구조의 목표는 CPU가 필요로 하는 대부분의 데이터가 상위 계층에 존재하게 만드는 것이다. 이를 통해 전체적인 메모리 접근 시간을 하위 계층이 아닌 상위 계층의 속도에 가깝게 만든다. 본 시뮬레이터는 이러한 계층 구조를 CPU 레지스터, L1 캐시, 메인 메모리의 3단계로 단순화하여 모델링한다.

2.2 Locality

메모리 계층 구조가 효율적으로 동작하는 근본적인 이유는 지역성 원리 때문이다. 프로그램은 메모리에 무작위로 접근하지 않고 특정 패턴을 보인다. 지역성은 프로그램의 과거 접근 패턴이 미래의 접근 패턴을 예측하는 중요한 근거가 된다.

2.2.1 Temporal Locality

시간적 지역성은 최근에 접근했던 메모리 위치는 가까운 미래에 다시 접근될 가능성이 높다는 원리이다. 반복문에서 사용되는 변수나 코드가 대표적인 예이다.

캐시는 이렇게 한 번 접근된 데이터를 저장해둔다. 후속 접근이 메인 메모리까지 가지 않고 캐시에서 빠르게 처리되도록 만든다. 내 시뮬레이터의 캐시 역시 최근에 사용된 데이터와 명령어 블록을 유지하여 이 원리를 활용한다.

2.2.2 Spatial Locality

공간적 지역성은 특정 메모리 위치에 접근했을 때 그 근처의 인접한 메모리 위치들도 곧이어 접근될 가능성이 높다는 원리이다. 배열의 원소를 순차적으로 접근하는 경우가 이에 해당한다. 이 원리를 활용하기 위해 캐시는 데이터 요청 시 해당 데이터 하나만 가져오지 않는다. 그 데이터를 포함하는 일정 크기의 덩어리인 캐시 라인단위로 데이터를 가져온다. 본 시뮬레이터의 cache.h에 정의된 CACHE_LINE_SIZE가 바로 이 공간적 지역성을 활용하는 단위이다. 캐시 미스 시 요청된 주소를 포함하는 64바이트 전체 블록을 메인 메모리에서 가져온다. 이는 향후 인접 주소에 대한 접근이 캐시 히트가 되도록 유도한다.

2.3 Cache Memory

캐시는 CPU와 메인 메모리 사이에 위치하는 작고 빠른 버퍼 메모리이다. 두 장치 간의 속도 차이를 완화하는 것이 캐시의 주된 역할이다. 캐시의 성능은 평균 메모리 접근 시간(AMAT)으로 측정할 수 있으며 공식은 다음과 같다.

AMAT = (Hit Time) + (Miss Rate × Miss Penalty)으로 본 시뮬레이터에서는 캐시 히트 시간을 1 클럭 사이클로 가정한다. 미스 패널티는 1000 클럭 사이클로 가정하여 AMAT를 계산하고 출력한다. 시뮬레이션의 궁극적인 목표는 다양한 캐시 설정을 통해 미스율을 낮추고 AMAT를 최소화하는 것이다.

2.4 Cache Mapping Strategies

캐시 mapping은 메인 메모리의 특정 블록을 캐시의 어느 위치에 저장할지 결정하는 규칙이다. 이 방식에 따라 캐시의 성능과 하드웨어 복잡성이 크게 달라진다.

2.4.1 Direct-Mapped Cache

직접 mapping은 메모리의 각 블록이 캐시 내의 단 하나의 정해진 위치에만 저장될 수 있는 방식이다. 주소의 'index' 필드를 통해 캐시의 특정 라인을 직접 지정

한다. 성능 측면에서 히트 판정이 단 한 번의 비교로 끝나 빠르다. 하드웨어 구현 비용도 저렴하다. 하지만 심각한 단점으로 충돌 미스가 빈번하게 발생할 수 있다. 서로 다른 메모리 블록들이 동일한 캐시 라인 위치를 두고 경쟁하는 경우이다. 이때 캐시의 다른 공간이 비어있음에도 지속적인 미스가 발생해 성능이 급격히 저하될 수 있다. 코드에서 cache.h의 WAYS 값을 1로 설정하면 직접 mapping 방식으로 동작을 시뮬레이션할 수 있다.

2.4.2 Fully-Associative Cache

완전 연관 방식은 메모리의 블록이 캐시 내의 어떤 라인에도 자유롭게 저장될 수 있는 방식이다. 주소에 'index' 필드가 없으며 전체 주소 영역이 'tag'로 사용된다. 성능 측면에서 충돌 미스를 원천적으로 방지한다. 따라서 주어진 캐시 크기에서 가장 높은 히트율을 기대할 수 있다. 그러나 데이터의 위치를 찾기 위해 모든 캐시 라인의 태그를 동시에 비교해야 한다. 이로 인해 비교기 하드웨어가 매우 복잡하고 비싸지며 탐색 속도가 느려진다. 이 때문에 대용량 캐시에는 실용적이지 않다.

2.4.3 Set-Associative Cache

세트 연관 방식은 직접 mapping과 완전 연관 방식의 장점을 절충한 방식이다. 현실적으로 가장 널리 사용된다. 캐시를 여러 개의 세트로 나누고 각 세트는 여러 개의 라인을 가진다. 메모리 블록은 주소의 'index' 필드에 의해 특정 세트로 지정된다. 그 세트 안에서는 어떤 Way에도 자유롭게 저장될 수 있다.

성능 측면에서 직접 mapping 방식에 비해 충돌 미스를 크게 줄여준다. 동시에 완전 연관 방식만큼의 하드웨어 부담을 요구하지 않는다. Way의 수를 늘릴수록 충돌 미스가 줄어 히트율이 높아지는 경향이 있다. 하지만 히트 시간과 하드웨어 비용이 소폭 증가할 수 있다. 본 프로젝트의 핵심 구현 방식이 바로 이 세트 연관 방식이다. cache.h에 정의된 WAYS 값을 통해 연관도를 조절할 수 있다. 이를 통해 연관도가 성능에 미치는 영향을 분석하는 것이 과제의 주요 목표 중 하나이다.

2.5 Cache Management Policies

캐시의 내용을 효율적으로 관리하고 성능을 유지하기 위한 알고리즘이다.

2.5.1 Replacement Policy

교체 정책은 캐시 미스가 발생했을 때 이미 모든 Way가 차 있는 세트에서 어떤 블록을 제거할지 결정하는 규칙이다. 좋은 교체 정책은 미래에 사용될 가능성이 가장 낮은 블록을 예측하여 제거한다. 이를 통해 캐시 히트율을 높인다. 본 시뮬레이터는 LRU(Least Recently Used) 정책을 기본으로 구현했다. 각 캐시 라인에 lru_counter를 두어 가장 오랫동안 참조되지 않은 라인을 찾아 교체한다. 이는 시간적 지역성을 최대한 활용하여 최근에 사용된 데이터를 캐시에 더 오래 유지하려는 전략이다.

2.5.2 Write Policy

쓰기 정책은 CPU가 쓰기 연산을 수행할 때 캐시와 메인 메모리의 데이터를 어떻게 동기화할지 결정하는 규칙이다. 본 시뮬레이터는 Write-Back 정책을 사용한다. 이 방식에서는 CPU의 쓰기 요청이 발생하면 우선 캐시에만 데이터를 기록한다. 그리고 해당 캐시 라인을 'dirty' 상태로 표시한다. 이 더티 상태의 라인이 나중에 다른 데이터에 의해 교체될 때만 메인 메모리에 최종 내용을 기록한다. 동일한 주소에 여러 번 쓰기 작업이 발생하는 경우 불필요한 메인 메모리 접근을 최소화한다. 따라서 Write-Through 방식보다 일반적으로 높은 성능을 보인다.

2.6 Instruction Cache and Data Cache

본 시뮬레이터는 현대 고성능 프로세서의 L1 캐시 구조를 모방했다. 캐시를 명령어용과 데이터용으로 분리한 Split Cache 구조를 채택했다.

2.6.1 Instruction Cache (I-Cache)

명령어 캐시는 프로그램의 실행 코드인 명령어만을 저장하기 위한 전용 캐시이다. 파이프라인의 명령어 인출(IF) 단계에서 사용된다. 프로그램 코드는 실행 중에 거의 변경되지 않으므로 사실상 읽기 전용으로 동작한다. 루프와 같은 구조 덕분에 매우 높은 지역성을 보여 히트율이 극도로 높은 경향이 있다. 본 시뮬레이터의 fetch() 함수는 I-Cache에 접근하며 i_cache 전역 변수가 이를 구현한 것이다.

2.6.2 Data Cache (D-Cache)

데이터 캐시는 명령어를 제외한 모든 데이터 즉 변수나 배열 등을 저장하기 위한 전용 캐시이다. 파이프라인의 메모리 접근(MEM) 단계에서 lw, sw 같은 데이터 전송 명령어에 의해 접근된다. 데이터는 프로그램 실행에 따라 수시로 읽고 쓰기가 발생한다. 데이터 접근 패턴은 명령어 접근보다 예측이 어려워 상대적으로 낮은 히트율을 보일 수 있다. 본 시뮬레이터의 memaccess() 함수는 D-Cache에 접근한다. d_cache 전역 변수가 이를 구현하며 Write-Back 정책이 적용된다.

이 두 캐시를 분리하는 가장 중요한 이유는 파이프라인의 구조적 해저드를 방지하기 위함이다. 캐시가 하나라면 IF 단계의 명령어 인출과 MEM 단계의 데이터 접근이 동일한 캐시 자원을 두고 충돌한다. 이는 파이프라인 정지를 유발한다. 캐시를 분리하면 두 작업이 각각 I-Cache와 D-Cache에 병렬적으로 동시에 접근할 수 있다. 따라서 구조적 해저드를 원천적으로 방지하고 파이프라인의 처리 효율을 극대화할 수 있다.

3. Implementation

3.1 cache.h

3.1.1 #define

```
#ifndef CACHE_H
#define CACHE_H

#define POLICY_LRU    0
#define POLICY_RANDOM 1
#define POLICY_FIFO   2
#define POLICY_SCA    3

#define CURRENT_REPLACEMENT_POLICY POLICY_SCA
#define CACHE_LINE_SIZE 64

#define I_CACHE_SIZE (1024 * 4)
#define I_CACHE_WAYS 8
#define I_CACHE_SETS (I_CACHE_SIZE / (CACHE_LINE_SIZE * I_CACHE_WAYS))

#define D_CACHE_SIZE (1024 * 4)
#define D_CACHE_WAYS 8
#define D_CACHE_SETS (D_CACHE_SIZE / (CACHE_LINE_SIZE * D_CACHE_WAYS))
```

과제의 핵심 목표인 성능 분석을 위해 캐시의 구조를 코드 수정 없이 쉽게 변경할 수 있는 방법이 필요했다. 이 문제를 해결하기 위해 #define 전처리기 지시문을 적극적으로 활용했다. CACHE_LINE_SIZE, I_CACHE_SIZE, I_CACHE_WAYS 같은 매크로의 숫자만 바꾸고 재컴파일하면 시뮬레이터가 전혀 다른 구조의 캐시로 동작하도록 설계했다. I_CACHE_SETS와 D_CACHE_SETS는 다른 매크로 값에 따라 자동으로 계산되므로 사용자가 직접 계산할 필요가 없다.

특히 CURRENT_REPLACEMENT_POLICY 매크로는 여러 교체 알고리즘(LRU, FIFO, SCA 등) 중 어떤 것을 사용할지 컴파일 시점에 결정하게 해준다. 이 설계 덕분에 각 정책의 성능을 개별적으로 정확하게 측정하고 비교할 수 있다.

3.1.2 struct

```
typedef struct {
    int hit;          // 1이면 히트, 0이면 미스
    unsigned int data; // 캐시에서 읽은 데이터 (lw 경우)
} CacheResult;

// 캐시 라인을 표현하는 구조체
typedef struct {
    int valid;
    int dirty;
    unsigned int tag;
    int lru_counter;
    int second_chance_bit;
    unsigned char data[CACHE_LINE_SIZE];
} CacheLine;

void init_cache();
CacheResult access_d_cache(unsigned int address, unsigned int write_data, int write_enable);
void update_lru(CacheLine* cache_set, int accessed_way);
int find_lru_victim(CacheLine* cache_set);

CacheResult access_i_cache(unsigned int address);
void update_i_cache_lru(CacheLine* cache_set, int accessed_way);
int find_i_cache_lru_victim(CacheLine* cache_set);

void update_lru(CacheLine* cache_set, int accessed_way);
int find_lru_victim(CacheLine* cache_set);
int find_random_victim();
int find_fifo_victim_d_cache(int set_index);
int find_fifo_victim_i_cache(int set_index);
int find_sca_victim_d_cache(int set_index);
int find_sca_victim_i_cache(int set_index);

#endif
```

파이프라인 구조는 필연적으로 해저드 문제를 야기한다. 본 설계는 이 문제들을 효과적으로 해결하는 데 중점을 두었다. 데이터 해저드는 Forwarding 기법으로 해결한다. 특정 연산 결과를 다음 명령어가 즉시 사용할 수 있도록 파이프라인 내부에서 데이터를 직접 전달한다. 이는 불필요한 스톨을 제거하여 파이프라인의 효율을 유지시킨다.

제어 해저드는 분기 예측으로 대응한다. 분기 명령어 발생 시 파이프라인은 멈추지 않는다. 대신 분기의 결과를 예측하고 다음 명령어를 투기적으로 인출한다. 예측이 성공하면 큰 성능 이득을 얻는다. 예측이 틀렸을 경우에만 파이프라인을 비우고 올바른 위치에서 다시 시작한다. 이 방식은 페널티의 위험을 감수하고 더 높은 평균 성능을 추구하는 전략이다.

파이프라인의 효율을 높여도 결국 메모리 접근 속도가 전체 성능을 제한한다. 이

'메모리 장벽' 문제를 해결하기 위해 정교한 캐시 시스템을 구축했다. 캐시는 명령어 캐시(I-Cache)와 데이터 캐시(D-Cache)로 분리했다. 이 분리형 캐시 구조는 구조적 해저드를 원천적으로 방지한다. 명령어 인출(IF)과 데이터 접근(MEM)이 서로 다른 캐시에 동시에 접근할 수 있기 때문이다. 이는 파이프라인 처리량을 극대화하는 핵심 설계이다.

캐시의 내부 구조는 세트 연관 방식을 채택했다. 이는 직접 mapping 방식의 충돌 미스 문제를 완화하고 완전 연관 방식의 과도한 복잡성을 피하는 균형 잡힌 선택이다. cache.h 파일에서 WAYS 값을 쉽게 변경할 수 있도록 설계하여 다양한 연관도에 따른 성능 분석을 용이하게 했다.

캐시 관리 정책으로는 LRU 교체 정책과 Write-Back 쓰기 정책을 사용했다. LRU는 시간적 지역성을 활용하여 가장 오랫동안 쓰이지 않은 데이터를 교체하는 효율적인 방식이다. Write-Back은 쓰기 작업을 우선 캐시에만 반영하고 'dirty' 비트로 관리한다. 메모리 쓰기 횟수를 최소화하여 시스템의 전반적인 성능을 향상시킨다.

캐시 미스는 파이프라인의 즉각적인 정지를 요구하는 중대한 이벤트이다. 미스가 발생하면 fetch 또는 memaccess 함수는 1000 사이클 이상의 막대한 페널티를 계산한다. 이 페널티는 pipeline_stall 전역 변수에 설정된다. main 루프는 이 변수 값을 감지하여 0이 될 때까지 파이프라인의 모든 활동을 멈추고 시간만 흘려보낸다. 이 스톨 메커니즘은 캐시 미스가 전체 성능에 미치는 영향을 현실적으로 시뮬레이션한다.

3.2 cache.c

cache.c는 캐시 시뮬레이터의 핵심 엔진 역할을 수행한다. 이 파일의 함수들은 캐시의 초기 상태를 설정하고 CPU의 메모리 접근 요청을 처리하며 캐시의 내용을 효율적으로 관리하는 모든 실질적인 로직을 담고 있다.

3.2.1 init_cache

```c
void init_cache() {
    // 데이터 캐시 초기화
    for (int i = 0; i < D_CACHE_SETS; i++) {
        for (int j = 0; j < D_CACHE_WAYS; j++) {
            d_cache[i][j].valid = 0;
            d_cache[i][j].dirty = 0;
            d_cache[i][j].tag = 0;
            d_cache[i][j].lru_counter = j; // LRU 카운터를 순차적으로 초기화
        }
    }

    // 명령어 캐시 초기화
    for (int i = 0; i < I_CACHE_SETS; i++) {
        for (int j = 0; j < I_CACHE_WAYS; j++) {
            i_cache[i][j].valid = 0;
            i_cache[i][j].dirty = 0;
            i_cache[i][j].tag = 0;
            i_cache[i][j].lru_counter = j; // LRU 카운터를 순차적으로 초기화
        }
    }
}
```

시뮬레이션을 시작하기 전에 캐시를 예측 가능한 초기 상태로 만드는 것이 중요했다. 이 함수는 모든 캐시 라인을 순회하며 valid 비트를 0으로 초기화한다. 이렇게 구현한 이유는 프로그램 시작 시 발생하는 콜드 미스를 정확히 식별하기 위해서이다. 초기화하지 않으면 유효하지 않은 데이터를 히트로 착각할 수 있기 때문이고 lru_counter도 순차적으로 초기화하여 교체 정책이 일관된 기준에서 시작하도록 만들었다.

3.2.2 Random / FIFO / SCA

```c
// Random 정책
int find_random_victim() {
    return rand() % D_CACHE_WAYS;
}

// 데이터 캐시용 FIFO 교체 정책
int find_fifo_victim_d_cache(int set_index) {
    int victim_way = d_cache_fifo_ptr[set_index];
    d_cache_fifo_ptr[set_index] = (victim_way + 1) % D_CACHE_WAYS;
    return victim_way;
}

// 명령어 캐시용 FIFO 교체 정책
int find_fifo_victim_i_cache(int set_index) {
    int victim_way = i_cache_fifo_ptr[set_index];
    i_cache_fifo_ptr[set_index] = (victim_way + 1) % I_CACHE_WAYS;
    return victim_way;
}
// 데이터 캐시용 SCA 교체 정책
int find_sca_victim_d_cache(int set_index) {
    while (1) { // victim을 찾을 때까지 계속 순회
        int current_way = d_cache_fifo_ptr[set_index];
        if (d_cache[set_index][current_way].second_chance_bit == 0) {
            // 기회가 없는 라인을 찾으면 해당 라인을 victim으로 선택하고 포인터 이동
            d_cache_fifo_ptr[set_index] = (current_way + 1) % D_CACHE_WAYS;
            return current_way;
        }
        else {
            // 기회가 있는 라인이면, 기회를 소진시키고(0으로 만들고) 포인터만 이동
            d_cache[set_index][current_way].second_chance_bit = 0;
            d_cache_fifo_ptr[set_index] = (current_way + 1) % D_CACHE_WAYS;
        }
    }
}

// 명령어 캐시용 SCA 교체 정책
int find_sca_victim_i_cache(int set_index) {
    while (1) {
        int current_way = i_cache_fifo_ptr[set_index];
        if (i_cache[set_index][current_way].second_chance_bit == 0) {
            i_cache_fifo_ptr[set_index] = (current_way + 1) % I_CACHE_WAYS;
            return current_way;
        }
        else {
            i_cache[set_index][current_way].second_chance_bit = 0;
            i_cache_fifo_ptr[set_index] = (current_way + 1) % I_CACHE_WAYS;
        }
    }
}
```

다양한 교체 정책을 실험하기 위해 각 알고리즘을 별도의 함수로 구현했다.

LRU 정책은 시간적 지역성을 활용하는 가장 대표적인 방법이다. lru_counter 값을 비교하여 가장 오랫동안 참조되지 않은 라인을 찾아 교체하도록 find_lru_victim 함수를 작성했다.

Random 정책은 가장 단순한 비교 기준을 제공한다. 그래서 rand() 함수를 이용

해 무작위로 교체 대상을 선택하는 간단한 함수로 구현했다.

FIFO 정책은 들어온 순서를 기억해야 한다. 나는 이 순서를 관리하기 위해 각 세트별로 fifo_ptr이라는 포인터 변수를 사용했다. find_fifo_victim 함수는 이 포인터가 가리키는 가장 오래된 라인을 반환하고 포인터를 다음 위치로 이동시킨다. 그래서 원형 큐와 같은 동작을 하도록 만들었다.

SCA는 FIFO의 단점을 보완하는 알고리즘이다. FIFO와 마찬가지로 fifo_ptr을 사용해 순차적으로 라인을 검사하도록 구현했다. 검사한 라인에 두 번째 기회(second_chance_bit=1)가 있다면 기회를 소진시키고 넘어가고 기회가 없는 라인(second_chance_bit=0)을 만났을 때 비로소 교체 대상으로 삼는다. 그래서 자주 사용되는 페이지가 쉽게 쫓겨나지 않도록 find_sca_victim 함수를 설계했다.

3.2.3 update_lru / find_lru_victim

```
void update_lru(CacheLine* cache_set, int accessed_way) {
    int prev_lru = cache_set[accessed_way].lru_counter;
    cache_set[accessed_way].lru_counter = 0;

    for (int i = 0; i < D_CACHE_WAYS; i++) {
        if (i == accessed_way) continue;
        if (cache_set[i].lru_counter < prev_lru) {
            cache_set[i].lru_counter++;
        }
    }
}

int find_lru_victim(CacheLine* cache_set) {
    int victim_way = 0;
    int max_lru = -1;
    for (int i = 0; i < D_CACHE_WAYS; i++) {
        // 비어있는 라인이 있으면 최우선으로 사용
        if (!cache_set[i].valid) {
            return i;
        }
        if (cache_set[i].lru_counter > max_lru) {
            max_lru = cache_set[i].lru_counter;
            victim_way = i;
        }
    }
    return victim_way;
}
```

효율적인 교체 정책을 구현하여 캐시 히트율을 높이고자 했다. 시간적 지역성을 잘 활용하는 LRU 정책을 선택했고 이를 두 함수로 나누어 구현했다.

교체 대상을 찾는 find_lru_victim 함수는 먼저 비어있는 라인을 찾도록 설계했는데 유효한 데이터를 내쫓는 것보다 빈 공간을 활용하는 것이 항상 더 효율적이기 때문이다. 빈 라인이 없을 때만 lru_counter를 비교하여 가장 오래된 라인을 찾도록 만들었다.

update_lru 함수는 카운터 기반의 간단한 방식으로 LRU를 구현한다. 캐시에 히트하면 해당 라인의 카운터를 0으로 만들어 가장 최근에 사용했음을 표시하고고 이전에 더 최근에 사용됐던 다른 라인들의 카운터만 1씩 증가시켜 상대적인 나이를 조절한다. 그래서 복잡한 시간 기록 없이도 효율적으로 LRU 정책을 시뮬레이션할 수 있도록 구현했다.

3.2.4 access_d_cache / access_i_cache

```
CacheResult access_d_cache(unsigned int address, unsigned int write_data, int write_enable) {
    CacheResult result = { .hit = 0, .data = 0 };

    int offset_bits = 0;
    if (CACHE_LINE_SIZE > 1) {
        int size = CACHE_LINE_SIZE;
        while (size >>= 1) offset_bits++;
    }

    int index_bits = 0;
    if (D_CACHE_SETS > 1) {
        int sets = D_CACHE_SETS;
        while (sets >>= 1) index_bits++;
    }

    unsigned int offset = address & ((1 << offset_bits) - 1);
    unsigned int index = (address >> offset_bits) & ((1 << index_bits) - 1);
    unsigned int tag = address >> (offset_bits + index_bits);

    CacheLine* target_set = d_cache[index];
    for (int i = 0; i < D_CACHE_WAYS; i++) {
        if (target_set[i].valid && target_set[i].tag == tag) {
            result.hit = 1;
            d_cache_hits++;

            if (write_enable) {
                memcpy(&target_set[i].data[offset], &write_data, 4);
                target_set[i].dirty = 1;
            }
            else {
                memcpy(&result.data, &target_set[i].data[offset], 4);
            }

            update_lru(target_set, i);
            return result;
        }
    }
```

```
CacheResult access_i_cache(unsigned int address) {
    CacheResult result = { .hit = 0, .data = 0 };

    int offset_bits = 0;
    if (CACHE_LINE_SIZE > 1) {
        int size = CACHE_LINE_SIZE;
        while (size >>= 1) offset_bits++;
    }

    int index_bits = 0;
    if (I_CACHE_SETS > 1) {
        int sets = I_CACHE_SETS;
        while (sets >>= 1) index_bits++;
    }

    unsigned int offset = address & ((1 << offset_bits) - 1);
    unsigned int index = (address >> offset_bits) & ((1 << index_bits) - 1);
    unsigned int tag = address >> (offset_bits + index_bits);

    CacheLine* target_set = i_cache[index];
    for (int i = 0; i < I_CACHE_WAYS; i++) {
        if (target_set[i].valid && target_set[i].tag == tag) {
            result.hit = 1;
            i_cache_hits++;

            memcpy(&result.data, &target_set[i].data[offset], 4);

            // I-Cache용 LRU 업데이트 (D-Cache와 동일한 로직 사용)
            update_lru(target_set, i);
            return result;
        }
    }

    result.hit = 0;
    i_cache_misses++;
    return result;
```

파이프라인이 캐시와 상호작용할 명확하고 표준화된 방법이 필요했어서 명령어용 access_i_cache와 데이터용 access_d_cache 두 개의 핵심 함수를 만들었다.

내부에서는 먼저 주소를 태그 인덱스 오프셋으로 분해한다. 이 과정을 cache.h의 매크로 값에 따라 동적으로 계산하도록 구현했다. 그래서 헤더 파일의 캐시 설정만 바꾸면 코드 수정 없이도 다양한 캐시 구조를 실험할 수 있어서 이는 과제의 성능 분석 목표를 달성하기 위한 설계다.

가장 중요한 설계 원칙은 역할 분리였다. 이 접근 함수들은 캐시 내부를 탐색해 히트 또는 미스 여부만 판별하고 그 결과를 보고하도록 만들었다. 실제 미스 페널티를 처리하고 파이프라인을 멈추는 복잡한 작업은 이 함수를 호출한 fetch나 memaccess 함수가 담당하게 했고 각 코드 부분의 책임을 명확히 나누어 디버깅을 더 쉽게 만들었습니다.

특히 access_d_cache는 write_enable 인자를 두어 lw와 sw를 하나의 함수에서 처리하도록 했는데 이는 코드 중복을 줄이고 일관성을 유지하기 위함이다.

4. Result

4.1 캐시 연관도

캐시의 연관도가 성능에 미치는 영향을 분석했다. 이를 위해 교체 정책은 LRU로 고정했다. 캐시의 Way 수를 1, 2, 4, 8로 변경하며 input4.bin 프로그램을 실행했다. 각 실행 결과 중 D-Cache의 핵심 성능 지표는 아래 표와 같다.

Associativity	D-Cache Misses	Conflict/Capacity Misses	Miss Rate (%)	AMAT (cycles)
1-way (DM)	131,528	131,464	1.81%	19.15
2-way	68,886	68,822	0.96%	10.59
4-way	64,112	64,048	0.89%	9.93
8-way	64,112	64,048	0.89%	9.93

실험 결과는 연관도가 캐시 성능에 미치는 영향을 명확히 보여준다. 1-way Direct Mapped 방식은 1.81%의 가장 높은 미스율을 기록했다. 이는 충돌 미스가 빈번하게 발생했기 때문이다. 실제로 충돌 및 용량 미스는 약 13만 건에 달했다.

Way 수를 2개와 4개로 늘리자 성능은 크게 향상되었다. 4-way 방식에서 미스율은 0.89%로 절반 가량 감소했다. AMAT 역시 19.15 사이클에서 9.93 사이클로 크게 단축되었다. 이는 Way가 늘어나면서 같은 세트에 여러 데이터를 저장할 수 있게 되었기 때문이다. 그래서 Direct Mapped에서 발생하던 충돌 미스가 효과적으로 제거되었다.

가장 주목할 점은 4-way와 8-way의 결과가 완전히 동일하다는 것이다. 이는 input4.bin 프로그램의 메모리 접근 패턴에서 발생하는 충돌 미스가 4-way 수준에서 이미 모두 해소되었음을 의미한다. 특정 세트에서 동시에 경쟁하는 데이터 블록의 수가 4개 이하라면 4-way 캐시는 이를 모두 수용할 수 있다. 따라서 Way를 8개로 더 늘려도 추가적인 성능 이득이 발생하지 않는 성능 포화 상태에 도달한 것이다.

결론적으로 이 실험은 연관도를 높이는 것이 항상 성능 향상으로 이어지지 않음을 보여준다. 프로그램의 특성에 따라 특정 수준 이상의 연관도는 더 이상 효과

가 없을 수 있다. input4.bin의 경우 4-way Set-Associative 방식이 충돌 미스 감소와 하드웨어 효율성 측면에서 가장 균형 잡힌 선택임을 데이터가 증명한다.

4.2 캐시 교체 정책

캐시 교체 정책이 성능에 미치는 영향을 분석했다. 이를 위해 캐시 종류는 4-way Set-Associative로 고정했다. LRU, FIFO, Random, SCA 네 가지 정책을 각각 적용하여 input4.bin 프로그램을 실행했다. 각 정책에 따른 D-Cache의 핵심 성능 지표는 아래 표와 같다.

교체 정책	D-Cache Misses	Miss Rate (%)	AMAT (cycles)
LRU	64,112	0.89%	9.93
SCA	64,184	0.89%	9.94
FIFO	66,003	0.92%	10.19
Random	66,812	0.93%	10.30

실험 결과는 교체 정책에 따라 캐시 성능이 의미 있는 차이를 보이는 것을 확인시켜준다. 성능은 LRU ≈ SCA > FIFO > Random 순으로 우수하게 나타났다.

LRU 정책은 0.89%의 가장 낮은 미스율과 9.93 사이클의 가장 빠른 AMAT를 기록하며 최고의 성능을 보였다. 이는 LRU가 시간적 지역성 원리를 가장 잘 활용하기 때문이다. LRU는 가장 오랫동안 사용되지 않은 데이터를 교체한다. 그래서 최근에 사용되었거나 자주 사용되는 데이터는 캐시에 오래 남아있을 확률이 높다. input4.bin 프로그램의 데이터 접근 패턴이 이러한 특성과 잘 맞아떨어져 높은 히트율로 이어진 것이다.

SCA 정책은 LRU와 거의 대등한 0.89%의 미스율을 보였다. 이는 LRU의 좋은 성능을 효율적으로 근사했음을 의미한다. SCA는 FIFO를 기반으로 동작하지만 '두 번째 기회' 비트를 사용한다. 캐시에 히트한 데이터에 기회를 주어 바로 교체되지 않게 만든다. 이 방식은 자주 사용하는 데이터가 큐의 뒤편에 있다는 이유만으로

교체되는 FIFO의 단점을 보완한다. 그래서 LRU와 유사한 성능을 낼 수 있었다.

반면 FIFO와 Random 정책은 각각 0.92%와 0.93%로 상대적으로 높은 미스율을 기록했다. FIFO는 데이터의 사용 빈도나 최근 사용 여부를 전혀 고려하지 않는다. 단지 캐시에 가장 먼저 들어왔다는 이유만으로 데이터를 교체한다. 만약 자주 사용하는 데이터가 가장 먼저 들어왔다면 불필요한 미스가 발생하게 된다. Random 정책은 아무런 전략 없이 무작위로 교체 대상을 선택한다. 이는 구현이 간단하지만 운이 나쁘면 방금 가져왔거나 자주 사용할 데이터를 바로 교체해버릴 수 있다. 그래서 평균적으로 LRU나 SCA보다 낮은 성능을 보이는 것이 일반적이다.

결론적으로 이 실험은 교체 정책이 캐시 성능에 미치는 영향을 명확하게 보여준다. 프로그램의 지역성을 잘 활용하는 LRU와 SCA 같은 지능적인 정책이 단순한 FIFO나 Random 정책보다 우수한 성능을 나타냈다. 이는 input4.bin 프로그램의 데이터 접근에 시간적 지역성이 뚜렷하게 존재함을 시사한다.

4.3 Mapping

캐시의 세 가지 주요 mapping 방식이 성능에 미치는 영향을 분석했다. 이를 위해 교체 정책은 LRU로 고정했다. Direct Mapped, 4-way Set-Associative, Fully Associative 세 가지 모드에서 input4.bin 프로그램을 각각 실행했다. 각 방식에 따른 D-Cache의 핵심 성능 지표는 아래 표와 같다.

Mapping Strategy	D-Cache Misses	Conflict/Capacity Misses	Miss Rate (%)	AMAT (cycles)
Direct Mapped	131,528	131,464	1.81%	19.15
Set-Associative	64,112	64,048	0.89%	9.93
Fully Associative	64,113	64,049	0.89%	9.93

실험 결과는 캐시 사상 방식의 이론적 장단점을 실제 데이터로 명확히 보여준다.

Direct Mapped 방식의 성능이 가장 낮게 측정되었다. D-Cache 미스율은 1.81%였

고 AMAT는 19.15 사이클을 기록했다. 이는 Direct Mapped 방식이 구조적으로 충돌 미스에 가장 취약하기 때문이다. 서로 다른 데이터가 같은 캐시 인덱스에 할당되면 끊임없이 서로를 교체한다. 실제 데이터에서도 약 13만 건의 압도적인 충돌/용량 미스가 발생한 것을 확인할 수 있다.

Set-Associative(4-way) 방식은 성능이 크게 향상되었다. 미스율은 0.89%로 절반 이하로 감소했고 AMAT 역시 9.93 사이클로 크게 개선되었다. 이는 연관도를 4로 높여 하나의 세트가 4개의 데이터 블록을 동시에 가질 수 있게 되었기 때문이다. 그래서 Direct Mapped에서 빈번했던 충돌 문제가 대부분 해소되었다. 충돌/용량 미스가 6만 4천 건 수준으로 크게 줄어든 것이 이를 증명한다.

Fully Associative 방식의 성능은 Set-Associative(4-way)와 거의 동일하게 측정되었다. 이론적으로 Fully Associative는 충돌 미스가 없어 가장 성능이 좋아야 한다. 하지만 이번 결과는 input4.bin 프로그램이 유발하는 충돌 미스가 이미 4-way 수준에서 대부분 해결되었음을 보여준다. 그 이상의 유연성을 제공해도 더 이상 막을 수 있는 충돌 미스가 거의 없었다. 그래서 성능 향상이 없는 성능 포화현상이 나타난 것이다.

5. Feelings

이번 컴퓨터 구조 설계를 통해 싱글 사이클부터 파이프라인 그리고 캐시까지 직접 구현하며 한 학기를 보냈다. 이론으로만 접했던 컴퓨터의 핵심 동작 원리를 코드로 한 줄씩 만들어가는거는 생각보다 어렵고 힘든 과정이었다. 디버깅을 하며 어디서 멈추고 왜 멈추는지 판단하는데 어려웠고 하나하나 멈춰가며 확인하는데 오래 걸렸다. 코드를 짤 때는 되겠지 하면서 짜지만 디버깅 과정을 제일 오래 걸리면서 찾기도 힘든 지루하지만 찾을 때는 쾌감이 있었다. 나름 한학기 동안 단계별로 올라가며 성장하는 느낌을 받았고 우리가 쉽게 접하고 있는 컴퓨터가 복잡하고 더 어려운거 같다. 많이 힘들고 고된 작업이지만 한 학기의 결과물을 다시 살펴보니 열심히 한거 같아서 뿌듯하다. 의문에서 시작해서 마침표는 못 찍었지만 많이 배운 수업이다.

결과물

1-way/ 2-way/ 4-way/ 8-way

```
----------------------------------------
Cache Statistics:
I-Cache: Hits = 24386896, Misses = 1596 (Cold: 0, Conflict: 0), Total = 24388492
D-Cache: Hits = 7116606, Misses = 68886 (Cold: 64, Conflict: 68822), Total = 7185492
I-Cache Miss Rate: 0.01%, AMAT: 1.07 cycles
D-Cache Miss Rate: 0.96%, AMAT: 10.59 cycles
****************************************
----------------------------------------
Cache Statistics:
I-Cache: Hits = 24386896, Misses = 1596 (Cold: 0, Conflict: 0), Total = 24388492
D-Cache: Hits = 7116606, Misses = 64112 (Cold: 64, Conflict: 64048), Total = 7180718
I-Cache Miss Rate: 0.01%, AMAT: 1.07 cycles
D-Cache Miss Rate: 0.89%, AMAT: 9.93 cycles
****************************************
----------------------------------------
Cache Statistics:
I-Cache: Hits = 24386896, Misses = 1596 (Cold: 0, Conflict: 0), Total = 24388492
D-Cache: Hits = 7116606, Misses = 64112 (Cold: 64, Conflict: 64048), Total = 7180718
I-Cache Miss Rate: 0.01%, AMAT: 1.07 cycles
D-Cache Miss Rate: 0.89%, AMAT: 9.93 cycles
****************************************
----------------------------------------
Cache Statistics:
I-Cache: Hits = 24386896, Misses = 1596 (Cold: 0, Conflict: 0), Total = 24388492
D-Cache: Hits = 7116606, Misses = 131528 (Cold: 64, Conflict: 131464), Total = 7248134
I-Cache Miss Rate: 0.01%, AMAT: 1.07 cycles
D-Cache Miss Rate: 1.81%, AMAT: 19.15 cycles
****************************************
```

LRU/ FIFO / RANDOM / SCA

```
----------------------------------------
Cache Statistics:
I-Cache: Hits = 24386896, Misses = 1596 (Cold: 64, Conflict: 1532), Total = 24388492
D-Cache: Hits = 7116606, Misses = 66003 (Cold: 64, Conflict: 65939), Total = 7182609
I-Cache Miss Rate: 0.01%, AMAT: 1.07 cycles
D-Cache Miss Rate: 0.92%, AMAT: 10.19 cycles
****************************************
----------------------------------------
Cache Statistics:
I-Cache: Hits = 24386896, Misses = 1596 (Cold: 64, Conflict: 1532), Total = 24388492
D-Cache: Hits = 7116606, Misses = 66812 (Cold: 64, Conflict: 66748), Total = 7183418
I-Cache Miss Rate: 0.01%, AMAT: 1.07 cycles
D-Cache Miss Rate: 0.93%, AMAT: 10.30 cycles
****************************************
----------------------------------------
Cache Statistics:
I-Cache: Hits = 24386896, Misses = 1596 (Cold: 64, Conflict: 1532), Total = 24388492
D-Cache: Hits = 7116606, Misses = 64184 (Cold: 64, Conflict: 64120), Total = 7180790
I-Cache Miss Rate: 0.01%, AMAT: 1.07 cycles
D-Cache Miss Rate: 0.89%, AMAT: 9.94 cycles
****************************************
----------------------------------------
Cache Statistics:
I-Cache: Hits = 24386896, Misses = 1596 (Cold: 0, Conflict: 0), Total = 24388492
D-Cache: Hits = 7116606, Misses = 64112 (Cold: 64, Conflict: 64048), Total = 7180718
I-Cache Miss Rate: 0.01%, AMAT: 1.07 cycles
D-Cache Miss Rate: 0.89%, AMAT: 9.93 cycles
****************************************
```

Set-Associative/ Direct Mapped / Fully-Associative

```
----------------------------------------
Cache Statistics:
I-Cache: Hits = 24386896, Misses = 1596 (Cold: 64, Conflict: 1532), Total = 24388492
D-Cache: Hits = 7116606, Misses = 131528 (Cold: 64, Conflict: 131464), Total = 7248134
I-Cache Miss Rate: 0.01%, AMAT: 1.07 cycles
D-Cache Miss Rate: 1.81%, AMAT: 19.15 cycles
****************************************
----------------------------------------
Cache Statistics:
I-Cache: Hits = 24386896, Misses = 1596 (Cold: 64, Conflict: 1532), Total = 24388492
D-Cache: Hits = 7116606, Misses = 64113 (Cold: 64, Conflict: 64049), Total = 7180719
I-Cache Miss Rate: 0.01%, AMAT: 1.07 cycles
D-Cache Miss Rate: 0.89%, AMAT: 9.93 cycles
****************************************
----------------------------------------
Cache Statistics:
I-Cache: Hits = 24386896, Misses = 1596 (Cold: 64, Conflict: 1532), Total = 24388492
D-Cache: Hits = 7116606, Misses = 64112 (Cold: 64, Conflict: 64048), Total = 7180718
I-Cache Miss Rate: 0.01%, AMAT: 1.07 cycles
D-Cache Miss Rate: 0.89%, AMAT: 9.93 cycles
****************************************
```