캐쉬 시뮬레이터 보고서

성주용 201614845

전북대학교 컴퓨터공학부

wndyd9557@gmail.com

요 약

캐쉬 구조와 동작과정을 공부하여 프로그램 구조와 자료구조를 계획하는데 4일, 프로그램을 구현하는 과정에서 3일, 테스트 및 예외상황 처리 에서 2일이 걸렸다. 32주소 체계를 가지고 있다고 가정하고 프로그램을 완성하였고 테스트해본 모든 케이스에서 올바른 값을 출력한다.

1. 실습 프로그램의 구성 및 동작 원리

먼저 자료구조에 대해서 설명하겠다.

```
typedef struct {
    int tag;
    int valid;
}Line;
```

구조체 Line에서 tag는 같은 인덱스일 경우 태를 비교하기 위해서 태그 값을 저장한다. valid는 그 라인의 유효성을 판별한다.

```
typedef struct {
    Line* myLine;
    LRU* lru;
}Set;
```

구조체 Set에서 myLine은 associativity크기의 Line구조체 배열을 가리킨다. 이는 N-way associative cache처럼 Line가 N개 있음을 나타낸다. lru는 Set에서 LRU(Least Recently Used)bit를 구하기 위해서 사용된다.

```
class LRU{
private:
    int size;
    int* RU;    //MRUbit는 RU[0], LRUbit는 RU[size-1]
    int associativity;
public:
    LRU() {}
    void init(int associativity){ ... }
    void put(int num){ ... }
    int getLRUbit(){ ... }
    int getSize(){ ... }
};
```

LRU클래스구현은 Standard LRU 알고리즘의 원리를 비슷하게 사용하여 구현하였다. RU배열에는 최근에 사용한 slot를 집어넣는다. 이때 특별한 put함수를 사용한다.

```
void put(int num) {
    int temp;
    int state = 1;
    for (int i = 0; i < size; i++) {        //최근 사용한 slot를 가장 앞으로..
        if (num == RU[i]) {
            state = 0;
            temp = RU[i];
            for (int j = i; j > 0; j--) {
                RU[j] = RU[j - 1];
            }
            RU[0] = num;
            break;
        }
    }
    if (size == associativity) {
        //무조건 겹치지는 않음
    }
    else if(state) {
        for (int i = size; i > 0; i--) {
            RU[i]=RU[i-1];
        }
        RU[0] = num;
        size++;
    }
}
```

put함수는 RU배열에 최근 사용한 slot를 집어 넣는데 원래 있던 값들을 한 간격을 뒤로 미루고 배열의 가장 앞에 값을 저장한다. RU배열 안에는 같은 값이 중복되지 않게 처리하였다. 고로 가장 최근에 사용한 slot은 RU[0]에 위치하고 가장 마지막에 사용한 slot은 RU[size-1]위치 즉 제일 뒷부분에 위치하게 한다. 이 원리를 이용하여 int getLRUbit()에서 LRU비트를 리턴 한다.

```
class Cache{
private:
    int totalbyte;      //cache의 전체 바이트 단위 크기
    int blockbyte;      //cache block 하나의 바이트 단위 크기
    int nSet;           //set의 개수    2의 index승
    int associativity;  //associativity,즉 N-way associative cache에서 N값
    int nTag;           //TIO각각의 비트
    int nIndex;
    int nOffset;
    int hit;            //접근, 히트, 미스 카운트
    int miss;
    int access;
    Set* allSet;        //Set구조체 배열을 가지고 있는 변수
    Set* mySet;         //Set 구조체를 가리키는 포인터 변수, 편한 연산을 위한..
public:
    Cache() {}
    ~Cache(){ ... }
    int init(int TB, int BB, int A){ ... }
    void printTIO(){ ... }
    void printfHitRate(){ ... }
    int Simulator(const char* traceFile){ ... }
};
```

Cache 클래스에는 캐쉬의 전체크기, 캐쉬 블록의 크기, 등 캐쉬의 정보를 가지고 있는 변수들이 저장되어 있다. 각 변수의 쓰임은 주석에서 설명하고 있다.

```
int init(int TB, int BB, int A) {                //주어진 값에따라 변수를 초기화 하는 함수
    totalbyte = TB;
    blockbyte = BB;
    associativity = A;
    nSet = (totalbyte/blockbyte)/associativity;

    nOffset = log((double)blockbyte)/log(2.0);          //계산법에 따라서 TIO비트구함
    nIndex = log((double)totalbyte)/log(2.0) - log((double)associativity)/log(2.0) - nOffset;
    nTag = 32 - nOffset - nIndex;
    hit=0;
    miss=0;
    access=0;
```

int init(int TB, int BB, int A)는 Cache 클래스를 초기화 하는 함수이다. 입력받은 cache의 전체 바이트 단위 크기, cache block 하나의 바이트 단위 크기, associativity를 입력 받아서 값을 저장하고 TIObit를 구한다. 시뮬레이터에서 사용될 변수들(hit, miss, access)을 초기화 한다.

```
allSet = (Set*)malloc((nSet)*sizeof(Set));
//set구조체를 nSet만큼(인덱스의개수)저장할 공간 만듬. allSet[nSet]
for(int i=0; i<nSet; i++) {          //각각의 set구조체 초기화
    mySet = allSet+i;
    mySet->myLine = (Line*)malloc((associativity)*sizeof(Line));
    //Line구조체를 associativity만큼(slot의개수)저장할 공간 만듬

    for(int j=0; j<associativity; j++) {          //각Line초기화
        (mySet->myLine+j)->tag =-1;
        (mySet->myLine+j)->valid =0;
    }
    mySet->lru= new LRU( );                       //LRU초기화
    (mySet->lru)->init(associativity);
}
```

Set구조체 배열을 동적으로 할당한다. 반복문을 이용하여 각 Set에 있는 Line구조체 배열을 동적으로 할당하고 초기화 한다. 각 Set에 있는 LRU클래스를 생성하고 초기화 한다.

```
if(log((double)blockbyte)/log(2.0) - (int)(log((double)blockbyte)/log(2.0)) != 0 ){
    printf("Cache block의 크기가 2의 지수승이 아닌 경우 ₩n");
    return 1;
}
if(associativity !=1 && associativity !=2 && associativity != 4 && associativity != 8) {
    printf("Associativity가 1, 2, 4, 8 이외의 값이 입력된 경우 ₩n");
    return 1;
}
if(totalbyte % (blockbyte * associativity) != 0) {
    printf("Cache 전체의 크기가 (cache block 크기)x(associativity)의 배수가 아닐 경우 ₩n");
    return 1;
}
return 0;
```

각 예외 상황에 대해서 에러 메시지를 출력하고 비정상 종료하게 한다. 이때 리턴 1이 비정상종료이다.

```
void printTIO() {                          //TIO비트를 출력하는 함수
    printf("tag: %d bits ₩nindex: %d bits ₩noffset: %d bits₩n", nTag, nIndex, nOffset);
}
```

void printTIO()함수는 init함수에서 구한 TIO값을 출력하는 함수이다.

```
int Simulator(const char* traceFile) {
    //주어진 캐쉬의 형태에 따라서 시뮬레이터를 돌리는 함수
    char none[256];
    int address=0;
    FILE* fin = NULL;
    if((fin = fopen(traceFile, "r"))== NULL ) {          //파일을 읽기모드로 오픈
        printf("존재하지 않은 트레이스 파일 입력했을 경우 ₩n"); //파일이 없을경우 리턴1
        return 1;
    }
    int tag, index, offset;                    //값을 저장할 정수
    int tagMask, indexMask, offsetMask;        //비트마스크

    offsetMask = blockbyte-1;
    //ex 만약 blockbyte == 16이라면 0000 0000 0000 0000 0000 0000 0000 1111
    indexMask = (nSet-1)<<nOffset;
    tagMask = -1<<(nIndex+nOffset);
```

가장 중요한 시뮬레이터 함수이다. 매개변수로 트레이스 파일명을 입력 받는다. 파일열기가 실패했을 경우 에러 메세지를 출력하고 비정상종료(리턴 1)을 한다. 요청된 address에서 태그와 인덱스와 오프셋을 분리하기 위해서 비트마스크를 사용하였다. 사용 원리는 오프셋의 비트가 4비트라면 OOO,,,OOOXXXX와 같이 분리하면 되기 때문에 2^4 = 16에서 1을 빼면 이진수로 000,,,0001111임을 이용해 비트마스크를 구했다. indexMask와 tagMask도 offsetMaks와 같은 원리를 이용한 뒤 <<연산자를 이용하여 구하였다.

```
while(!feof(fin)) {
    if(fscanf(fin, "%s %x %s", none, &address, none) != 3) break;
```

파일이 끝을 만날 때 까지 반복문을 돌렸다. 트레이스 파일이 한문장에 I 0x6451321654 4와 같은 구조를 가지고 있으므로 fscanf를 이용해서 파일을 한줄씩 읽으면서 address값을 구별했다. fscanf함수의 반환값이 읽어들인 변수의 개수임을 이용해서 3개를 못 받았을 경우 반복문을 끝낸다. 이는 fscanf가 ₩n₩0을 읽을 때 읽음을 실패하는 것을 예외처리한 것이다. 읽음을 실패할 경우 변수 값이 전과 같은 값이므로 히트레이트 구할 때 문제가 되기 때문이다.

```
offset = address & offsetMask;
index = (address & indexMask)>>nOffset;
tag = (address & tagMask)>>nIndex+nOffset;
```

각 태그와 인덱스와 오프셋 값은 비트 연산자 &와 >>를 이용해서 address와 비트마스크를 비교해서 값을 구했다.

```
access++;                                  //접근++
//printf("@tag : %x,₩tindex : %x,₩t",tag, index);
mySet = allSet + index;                    // mySet = allSet[index]
bool missOrHit = false;                    //false는 miss, true는 hit
for(int j=0; j<associativity; j++) {
    if((mySet->myLine)->valid == 1 && (mySet->myLine+j)->tag == tag) {
        //라인이 채워져 있고 같은태그가 있다면
        missOrHit = true;
        hit++;                             //히트++
        //cout<<"hit,₩tslot : "<< j<<endl;
        (mySet->lru)->put(j);              //LRU계산
        break;
    }
}
```

한번 접근할 때마다 access를 증가시킨다. mySet은 접근할 인덱스의 Set을 가리킨다. mySet에 있는 slot중 유효하고 같은 태그가 있는경우 hit를 증가시킨다. 그리고 LRU클래스에 최근 사용한 slot을 추가한다.

```
if(missOrHit ==false) {                    //미스라면
    miss++;                                //미스++

    int size = (mySet->lru)->getSize();
    //size를 포문에서 invalue를 체크해서 구할수도 있지만,,
    //LRU구조체를 이용하여 구함.
    if(size == associativity) {
        //꽉차있다면 LRU비트에 값을 채워 넣는다.
        int LRUbit = (mySet->lru)->getLRUbit();
        (mySet->myLine+LRUbit)->tag=tag;
        (mySet->myLine+LRUbit)->valid=1;
        (mySet->lru)->put(LRUbit);  //LRU계산
        //cout<<"miss,₩tslot : "<<LRUbit<<endl;
    }else {
        //꽉차지 않았다면 남은 칸에 값을 채워 넣는다.
        (mySet->myLine+size)->tag=tag;
        (mySet->myLine+size)->valid=1;
        (mySet->lru)->put(size);    //LRU계산
        //cout<<"miss,₩tslot : "<<size<<endl;
    }
}
```

Size는 LRU클래스에서 가져왔다. LRU클래스는 slot을 넣을 때 중복을 허용하지 않으므로 size가 사용한 slot의 개수와 같다. 이는 value한 Line의 개수와 같다. 히트가 되지 않았을 경우 모두 miss를 증가시킨다. 만약 mySet이 꽉차있다면 LRU Line에 태그를 저장한다. 그리고 LRU클래스에 사용한 slot를 추가한다.
꽉차지 않았다면 mySet의 사용하지 않은 slot에 있는

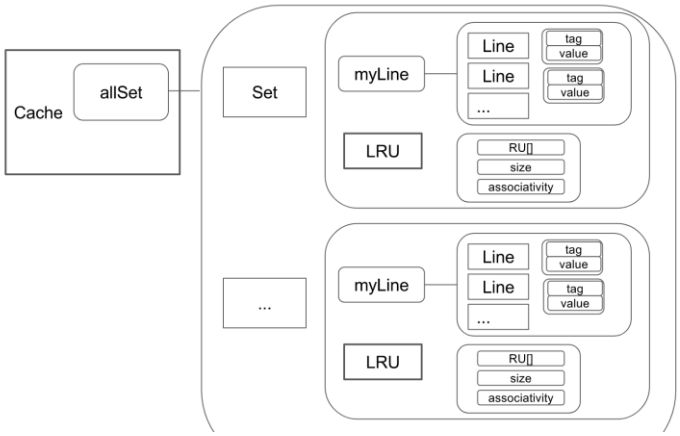
Line에 태그값을 저장하고 유효성을 저장한다. 그리고 LRU클래스에 사용한 slot를 추가한다.

```
            }
        }
    }
    fclose(fin);                //파일을 닫는다.
    return 0;
  }
};
```

파일에서 모든 라인을 읽으면 파일을 닫고 정상종료(리턴 0)을 한다.

```
void printfHitRate() {  //시뮬레이터의 결과 히트레이트를 출력하는 함수
    printf("Result: total access %d, hit %d, hit rate %.2f",
        access, hit, (float)hit/access);
}
```

void printfHitRate()함수는 시뮬레이터에서 구한 access, hit를 가지고 히트레이트를 출력하는 함수이다.



[그림 1: 전체 구조]

위의 그림 1처럼 클래스(Cache, LRU)와 구조체(Set, Line)가 서로 연관되고 사용하기 편하게 객체 지향적 구조를 설계하였다.

```
int main(int argc, const char* argv[]) {

    if(argc != 5) {
        printf("프로그램의 입력 인자가 4개가 아닐 경우₩n");
        printf("사용법. simple_cache_sim <trace file> <cache'
        return 1;
    }
    Cache* cache = new Cache();
    const char* traceFile = argv[1];

    if(cache->init(atoi(argv[2]),atoi(argv[3]),atoi(argv[4]))
        //입력된 값의 초기화가 정상일 경우
        cache->printTIO();
        if(cache->Simulator(traceFile) == 0) {
            //시뮬레이터가 정상 종료됐을 경우
            cache->printHitRate();
        }else
            printf("시뮬레이터의 비정상 종료");
    }else
        printf("올바르지 않은 캐쉬의 초기화₩n");

    return 0;
}
```

메인 함수에서는 입력 받은 인자의 개수에 따른 예외처리를 한다. Cache클래스를 생성하고 init함수를 이용해서 초기화를 하는데 입력 받은 인자가 잘못될 경우 예외 처리를 하였다. 정상일 경우 Simulator함수를 호출해서 정상 종료일 경우 printHitRate함수를 호출해서 히트레이트를 출력하고 종료한다.

2. 결과

```
>simple_cache_sim.exe gcc.trace
프로그램의 입력 인자가 4개가 아닐 경우
사용법. simple_cache_sim <trace file> <cache's byte size> <cache
block's byte size> <number of associativity>

>simple_cache_sim.exe gcc.trace 1024 15 2
Cache block의 크기가 2의 지수승이 아닌 경우
올바르지 않은 캐쉬의 초기화

>simple_cache_sim.exe gcc.trace 1024 16 3
Associativity가 1, 2, 4, 8 이외의 값이 입력된 경우
올바르지 않은 캐쉬의 초기화

>simple_cache_sim.exe gcc.trace 1024 256 8
Cache 전체의 크기가 (cache block 크기)x(associativity)의 배수가 아닐 경우
올바르지 않은 캐쉬의 초기화
```

[그림 2: 예외 처리]

각 예외 상황에 모두 에러 메세지를 출력한다.

```
>simple_cache_sim.exe gcc.trace 1024 16 2
tag: 23 bits
index: 5 bits
offset: 4 bits
Result: total access 515683, hit 468811, hit rate 0.91

>simple_cache_sim.exe gcc.trace 1024 16 4
tag: 24 bits
index: 4 bits
offset: 4 bits
Result: total access 515683, hit 481364, hit rate 0.93

simple_cache_sim.exe gcc.trace 1024 16 8
tag: 25 bits
index: 3 bits
offset: 4 bits
Result: total access 515683, hit 482373, hit rate 0.94

>simple_cache_sim.exe gzip.trace 1024 16 2
tag: 23 bits
index: 5 bits
offset: 4 bits
Result: total access 481044, hit 321006, hit rate 0.67
```

[그림 3: 출력 결과]

모든 케이스에 대해서 올바른 히트레이스를 출력한다.

3. 결론

코드를 코딩하고 디버깅에는 많은 시간이 걸리지 않았다. 알고리즘도 나름 간단했다. 하지만 캐쉬의 구조를 공부하고 이해한 뒤 구조를 계획하는데 정말 많은 시간이 들었다. 어떻게 하면 좀더 캐쉬의 구조와 같게 할 수 있을까? 어떻게 해야 좀 더 객체지향적으로 만들 수 있을까?라는 생각을 많이 하고 코드를 코딩하는 중간에도 구조를 수정하면서 완성했다. 과제를 하면서 캐쉬의 구조와 효율성에 대해서 많은 생각을 하게 되었고 수업시간에 이론적으로만 배워 어렵게 느껴졌던 캐쉬를 완벽하게 이해할 수 있었던 좋은 경험이였다.