Cache

MIPS Simulator

Lab Work #4

32181928 박찬호

Dankook University

Mobile Systems Engineering

2022 Spring

서론

컴퓨터의 발전과 함께 CPU 는 진화해왔다. Lab work 2, 3 에서 실험한 것과 같이 CPU 의 구조를 개선하고, 또한 회로의 집적도를 높이면서 같은 시간 내의 연산 처리량과 연산 속도는 크게 향상되었다.

그러나 폰 노이만 구조를 바탕으로 한 현대의 컴퓨터들에게는 CPU 의 속도 향상이 컴퓨터의 속도 향상으로 온전히 이어지지 않는다. 이는 폰 노이만 구조에서 메모리가 연산 장치 맞은 편의 한 축을 담당하기 때문이다. 주 메모리, 주로 DRAM 은 CPU 에 비해 현저히 속도가 느리고, 다른 저장 장치에 비해 비교적 작은 용량을 가진다. 따라서, CPU 와 메모리의 속도 차이로 발생하는 병목 현상으로 인해 CPU 는 메모리 속도에 따라 연산 처리 속도가 하향되고, 제한된 메모리 크기로 인해 프로그램의 모든 데이터를 메모리에 담아둘 수도 없다. 이러한 상황에서 컴퓨터의 성능을 향상시키기 위한 가장 단순한 방법으로, 메모리의 크기를 늘리고 속도를 빠르게 하는 방법을 생각해볼 수 있으나 이러한 접근은 분명한 기술적 한계를 갖는다.

Locality

그러나 Locality 라는 경향성은 우리에게 다른 방식의 해결책을 제시한다. 프로그램이 어떤 특정 시간 내에 주소 공간 내의 비교적 작은 부분만을 접근하게 되는 이러한 경향성을 principle of locality 라고 하는데 이는 마치, 지금 보고서를 작성하기 위해 교재의 캐시 관련 부분을 반복적으로 들여다보는 것과 비슷하다고 할 수 있다.

Locality 에는 두 종류가 있다.

Temporal Locality : Temporal Locality 는 한 번 참조된 항목은 곧바로 다시 참조되는 경향을 말한다. 예를 들면, for loop 에서 iterator 값을 계속 참조하거나, 어떤 변수에 대하여 증감 연산을 수행하는 등의 상황이 Temporal Locality 에 해당하는 경우라고 할 수 있다.

Spatial Locality : Spatial Locality 는 어떤 항목이 참조될 경우, 그 근처의 다른 항목이 곧바로 참조될 가능성이 높은 경향을 말한다. 예컨대, 배열이나 스택 메모리 등 순차적으로 인접한 공간에 값을 저장하고 참조하는 경우가 Spatial Locality 에 해당된다고 할 수 있다.

메모리 계층 구조

메모리 계층 구조는 유한한 메모리를 여러 계층으로 추상화하여 이러한 Locality 를 잘 이용할 수 있다.

일반적으로 빠른 메모리는 비싼 가격에 작은 용량을 갖고, 저렴한 메모리는 용량이 큰 대신 느린 속도를 가진다. 따라서 속도와 가격 대비 용량으로 구분되는 여러 종류의 메모리를 계층 별로 구성하고, 빠르고 작은 메모리일수록 CPU 에 가깝게 계층을 구성하면 Locality 라는 경향성을 잘 활용하는 구조가 된다. 더 자세히 설명하면, 빈번하게 사용되는 적은 수의 데이터일수록 CPU 와 가까운, 빠르고 작은 메모리에 배치하여 병목 현상을 줄이고, 덜 사용되는 다수의 데이터는 그 빈도에 따라 점차 CPU 로부터 먼, 비교적 느리고 큰 메모리에 배치함으로써, Locality 를 활용할 수 있다.

메모리 계층구조에서 데이터는 인접한 두 계층 사이에서만 복사된다. 따라서, 각 계층에 데이터를 저장하고 참조하는 과정은 세심한 고려를 바탕으로 이뤄져야 한다. 이는 메모리 계층구조의 역할이 성능향상에 있기 때문이다. 각 메모리 계층에 필요한 데이터가 존재할 경우 hit, 존재하지 않을 경우 miss 라고 하는데, miss 일 경우 penalty 는 메모리 구조의 하위 계층에서 데이터를 찾아 다시 현재의 상위 계층에 update 하는 시간을 포함하므로, hit rate 가 높아지도록 세심하게 계층을 구성하는 것이 중요해진다.

캐시 Cache

이러한 계층 구조의 최상위를 구성하는 것이 바로 Cache 이다. Cache 는 SRAM 이라는 기술을 사용하여 빠르지만 집적도가 낮은 메모리를 구성한다. <컴퓨터 구조 및 설계, Patterson & Hennessy> 에 따르면 2020 년 기준 DRAM 의 접근 속도가 50 – 70ns 정도인데에 반해, SRAM 의 접근 속도는 0.5 – 2.5ns 정도로 50 - 100 배 가까이 빠르다. 그러나 GiB 당 가격은 DRAM 에 비해 100-300 배 가까이 비싸, SRAM 은 집적도가 낮은 메모리로 사용할 수밖에 없다. 그럼에도 계층구조에서 Cache 를 locality 에 맞게 잘 사용하면, register 파일과 main memory 간의 병목현상을 효율적으로 줄일 수 있다. 예컨대, MIPS pipeline 에서 Memory Access 가 일어나는 연산의 경우 메모리 대신 캐시를 사용하면 연산의 성능을 상당히 향상시킬 수 있다.

문제는 Cache 의 크기를 main memory 만큼 크게 만들 수 없다는 것에 있다. 따라서, 더 큰 주소 공간의 데이터를 Cache 에 알맞게 가져오는 과정이 필요한데, 이때 데이터를 할당하는 방식에 따라 Cache 의 종류가 나뉘게 된다

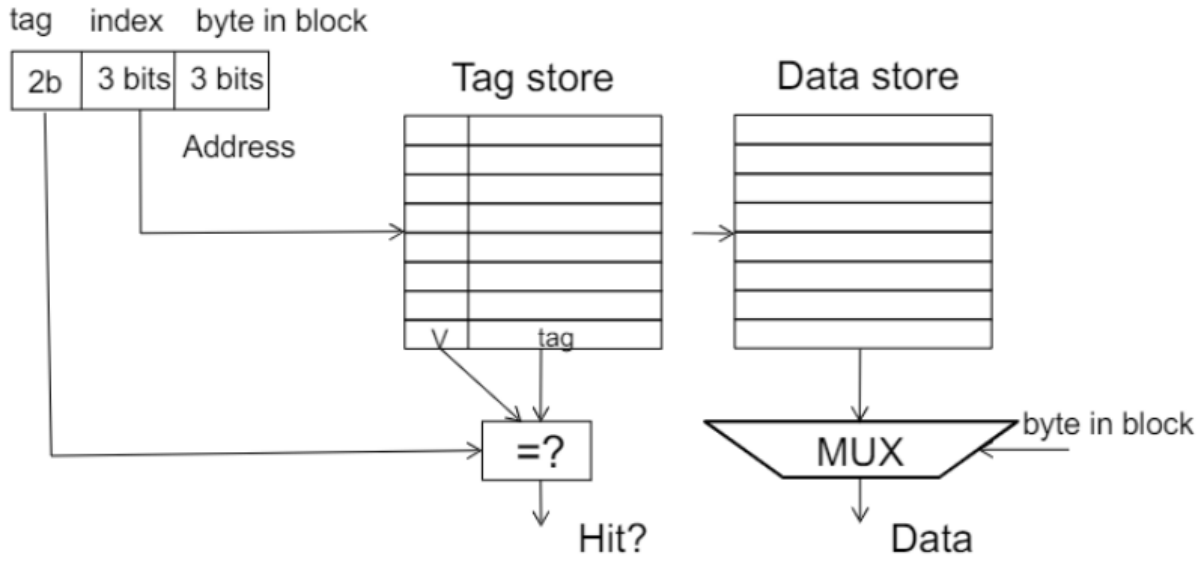
Direct Mapped Cache

첫 번째로 떠올릴 수 있는 방법은 메모리의 주소 공간을 이용하여 Cache 의 공간에 직접 mapping 하는 것이다. 즉, 메모리의 주소 중 일부를 캐시의 공간을 가리키는 주소처럼 사용하는 방식이다. 이때, 메모리의 주소 공간이 더 크기 때문에 메모리에 존재하는 N 개의 주소가 1 개의 Cache 공간으로 직접 mapping 된다.

Direct Mapped Cache 에 메모리의 주소를 할당하기 위해서는 Tag, Index, Offset 이 필요하다. 캐시의 각 row 를 Cache line 이라고 하는데, Cache line 의 길이만큼 offset 이 필요하게 되고, Cache line 을 몇 개 구성하느냐에 따라 Index 가 결정된다.

Index 와 Offset 만큼을 주소에서 잘라내고 남은 부분은 Tag 로 활용하여, 동일한 index 를 가지는 N 개의 주소 공간에 대해 hit 또는 miss 를 판별할 수 있다.

이러한 캐시 구조는 아래의 그림으로 이해할 수 있다.



Associative Cache

Direct Mapped Cache 의 단점은 메모리의 주소를 직접 mapping 하여 자주 사용되지 않는 영역이 있을 수 있다는 데에 있다. 즉, index 를 계산한 값이 동일한 cache 공간에 몰리게 되면, 해당 공간에서 반복적으로 miss 와 replacement 다는 데에 있다.

이러한 부분을 해결하기 위해 Associative Cache 방식을 사용할 수 있다. Associative Cache 에서는 직접적으로 cache 와 memory 간에 주소 공간을 mapping 하지 않고, 각 cache entry 에 순차적으로 메모리에서 placement 를 진행한 후 여러 방식을 통해 replacement 를 진행한다.

Fully Associative Cache 는 전체 Cache 의 어느 공간이든 메모리의 새로운 값이 placement 될 수 있다. 그러나 이러한 방식으로 Cache 내에서 데이터를 찾으려면 Cache 의 전체 영역을 다 검색하여 값을 찾아야 한다는 단점이 있다.

이를 보완하기 위해 Set Associative Cache 를 사용하면 전체 Cache Line 을 몇 개의 set 으로 묶고, 메모리의 주소를 특정 set 으로 mapping 시킨다. 이후, 해당 set 내에서는 메모리의 값이 어느 장소에든 저장될 수 있다.

이와 같이 Associative Cache 를 사용하기 위해서는, 어느 공간에든 배치된 값 중 교체될 값을 잘 선택하는 것이 중요하다. 이를 위해 MRU, Victim-Next Victim, LRU 등의 방법이 사용된다,

Least Recently Used & Second Chance Algorithm

캐시에 miss 가 난 경우 어떤 값을 replace 하면 좋을까? 가장 심플한 아이디어는, 가장 오래된 캐시 라인을 교체하는 것이다. 이것이 바로 LRU 로, Associative Cache 와 같이 메모리와 직접 mapping 되지 않은 경우, 어떤 값을 지울지 선택하는 데에 사용된다. 그러나 이는 캐시의 로그를 계속 기록해야 하므로 상당한 오버헤드가 발생한다. 이보다 더 간단한 방법은 캐시 line 에 한 번 더 기회를 주는 것이다. 즉, 두 번 연달아 miss 가 날 경우 replace 가 일어나게 되며, 이는 Associative Cache 뿐 아니라 Directly Mapped Cache 에도 적용할 수 있다.

Cache Read / Write

캐시에서 값을 읽는 과정은 단순하다. 캐시의 값이 valid 할 경우, index 위치의 tag 가 동일하다면 hit, 그렇지 않다면 miss 이다. hit 이라면 캐시로부터 읽고, miss 라면 메모리의 값을 읽는다.

문제는 쓰기를 처리하는 방식이다.

Write Back / Write Through

Write Back 은 메모리 쓰기(sw) 가 발생했을 때 그 값을 캐시에만 쓰고, 나중에 miss 가 날 경우 메모리에 쓰기를 수행하는 방식이다. 이는 조금 더 복잡한 로직을 사용하지만, 빈번하게 메모리에 쓰기를 진행하지 않아 훨씬 나은 성능을 가진다. Write Through 는 반면, 쓰기가 발생할 때마다 메모리와 캐시에 모두 쓰기를 수행한다. 이 경우 로직은 단순하지만 훨씬 많은 비용이 소모되며, Write Through 의 경우 성능 개선을 위해 write buffer 를 사용할 수 있다.

실행 방법

첨부된 파일을 Visual Studio 로 실행하면 실행 가능하다. 테스트 환경은 Windows 10 64bit & Visual Studio 2019 이다.

본론

소스 코드 설명

소스 코드는 파이프라인의 코드에서 캐시를 구현하기 위해 변경된 부분을 중점적으로 설명한다.

먼저 CacheLine 은 아래와 같이 구성하였다

```
typedef struct {
    unsigned int tag : 19;
    unsigned int sca : 1;
    unsigned int valid : 1;
    unsigned int dirty : 1;
    int data[16];
} CacheLine;
```

Int 형으로 16 개의 line 을 가지므로, cache line 은 64Byte 이다.

이때 캐시의 line 개수는 128 개로 총 cache 의 크기는 2^13 Byte = 8KB 이다.

따라서 tag 의 비트 수는 = 32 – 6 – 7 = 19 가 된다.

```
CacheLine inst_cache[128];

CacheLine data_cache[128];

for (int i = 0; i < 128; i++) {
        inst_cache[i].dirty = 0;
        inst_cache[i].sca = 0;
        inst_cache[i].valid = 0;
        inst_cache[i].tag = -1;

        data_cache[i].dirty = 0;
        data_cache[i].sca = 0;
        data_cache[i].valid = 0;
        data_cache[i].tag = -1;
}
```

본 구현에서는 Instruction Cache 와 Data Cache 를 별도로 구현하였다.

캐시는 아래와 같이 실행된다

```
    int tag = (pc & cache_mask[0]) >> cache_shift_length[0];
    int index = (pc & cache_mask[1]) >> cache_shift_length[1];
    int offset = pc & cache_mask[2];
```

먼저 캐시 실행에 앞서 tag, index, offset을 구한다.

이후 아래의 코드대로 실행되는데 이를 pseudocode로 다시 풀어쓰면 다음과 같다.

```
    if (!cache[index].valid) {
        instruction = read_instruction(pc, (char*)memory);

        for (int offset = 0; offset < 64; offset += 4) {
            int cache_pc = (tag << cache_shift_length[0]) | (index <<
cache_shift_length[1]) | offset;
            cache[index].data[offset >> 2] = read_instruction(cache_pc, memory);
        }
        cache[index].tag = tag;
        cache[index].valid = 1;
    } else {
        bool hit = (cache[index].tag == tag);

        if (hit) {
            instruction = cache[index].data[offset >> 2];
            cache[index].sca = 1;
            inst_hit++;
        } else {
            instruction = read_instruction(pc, (char*)memory);

            if (cache[index].sca == 0) {
                for (int offset = 0; offset < 64; offset += 4) {
                    int cache_pc = (tag << cache_shift_length[0]) | (index <<
cache_shift_length[1]) | offset;
                    cache[index].data[offset >> 2] = read_instruction(cache_pc, memory);
                }
                cache[index].tag = tag;
            } else {
                cache[index].sca = 0;
            }
        }
    }
```

1 if cache line is invalid (not initialized)

    1 get from memory

    2 place cache line from memory

    3 set tag of the index in cache

    4 change cache line to valid

2 else

    1 if cache is hit

        1 get from cache

        2 set sca of index to 1

    2 else

        1 get from memory

        2 if sca is 0

            1 place cache line from memory

            2 set tag of the index in cache

        3 else

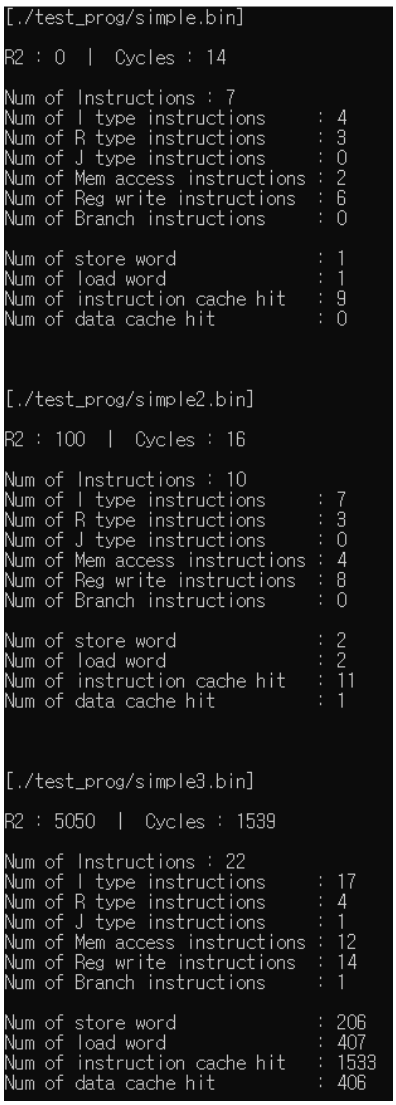
            1 set sca of index to 0

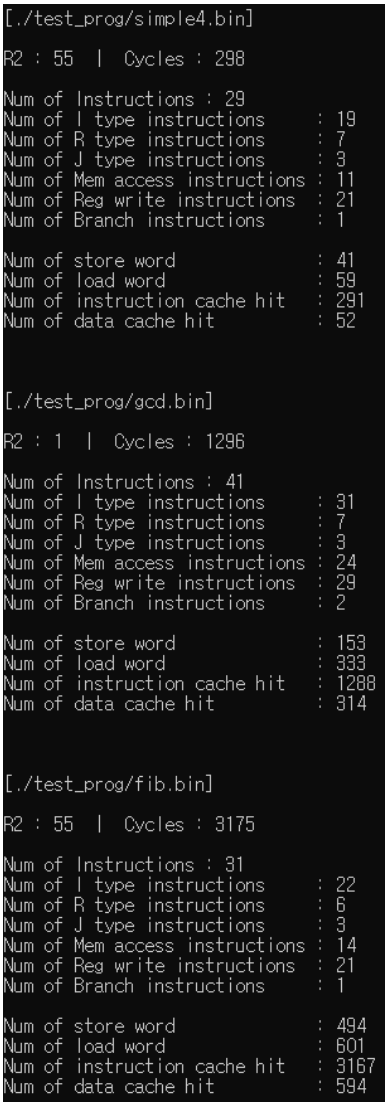
이로써 전체 코드에 대한 설명은 완료되었다.

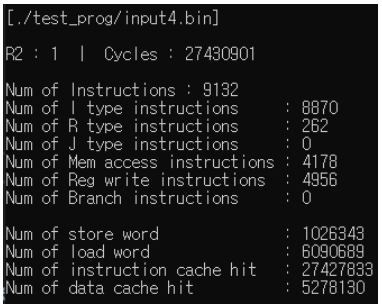
결론

결과 분석

본 프로그램의 결과를 소개하고 분석하겠다.







결과는 위와 같이 출력 되었는데, 이를 캐시가 없을 때의 결과와 비교하도록 하겠다.

비교 방식은 다음과 같다. Instruction Memory 와 Data Memory 에 Access 하는 명령어의 총 실행횟수를 memory access 라고 하고, 이 중 Cache Hit 이 발생한 횟수를 구한다. 이를 통해 hit rate 와 실행 시간 등을 구하여 최종 결과를 비교한다. 이때 메모리는 1000 만큼의 시간, 캐시는 1 만큼의 시간이 필요하다고 가정한다.

INPUT FILE	SAME RESULT	MEMORY ACCESS	CACHE HIT	HIT RATE (%)	ONLY MEMORY TIME	CACHE USING TIME
SIMPLE	Yes	16	9	56	16000	7009
SIMPLE2	Yes	20	12	60	20000	8012
SIMPLE3	Yes	2152	1939	90	2152000	214939
SIMPLE4	Yes	398	343	86	398000	55343
GCD	Yes	1782	1602	90	1782000	181602
FIB	Yes	4270	3761	88	4270000	512761
INPUT4	Yes	34547933	32705963	95	34547933000	1874675963

이를 통해 캐시를 통해 상당한 성능 개선을 이루었음을 알 수 있다.

감사합니다