Implementation of Cache MIPS

성영준 Yeong-Jun Seong

32202231

단국대학교 모바일시스템공학과

목차

1.1 서론

본 프로그램은 MIPS 아키텍처 기반의 프로세서를 소프트웨어로 구현하여 다양한 명령어를 실행하고, 실행 과정에서 캐시의 동작을 시뮬레이션하도록 설계되었다. 프로세서는 기본적인 명령어 실행 파이프라인을 통해 명령어를 가져오고, 디코드하고, 실행하며, 메모리에 접근하고, 결과를 레지스터에 기록하는 과정으로 구성된다.

캐시 시스템은 L1 캐시로 구성되며, 다양한 매핑 방식(직접 매핑, 2-way, 4-way, 8-way, 완전 연관)과 교체 정책(랜덤, FIFO, LRU, 2차 기회 알고리즘)을 지원한다. 이러한 캐시 시스템은 메모리 접근 성능을 향상시키기 위해 설계되었으며, 프로세서가 메모리에 접근할 때 발생하는 캐시 히트와 미스, 교체 상황을 시뮬레이션한다.

이 프로그램을 통해 배운 점은 캐시의 동작 원리와 그 중요성, 그리고 다양한 교체 정책과 매핑 방식이 캐시 성능에 미치는 영향을 이해하는 것이다. 캐시 히트와 미스, 교체 정책의 선택이 프로세서 성능에 어떻게 영향을 미치는지를 관찰함으로써 캐시 설계의 중요성을 인식할 수 있었다. 또한, 파이프라인에서 발생하는 데이터 및 제어 해저드와 이를 해결하기 위한 기법들을 학습함으로써, 보다 효율적인 프로세서 설계에 대한 통찰을 얻을 수 있었다.

2.0 Cache

현대 컴퓨터 시스템에서 CPU와 메모리 간의 속도 차이는 성능의 주요 제약 요인 중 하나이다. CPU는 매우 빠른 속도로 동작하며, 초당 수십억 개의 명령어를 처리할 수 있다. 그러나 메모리 접근 시간은 CPU의 처리 속도에 비해 상대적으로 느리다. 예를 들어, CPU는 나노초(ns) 단위로 동작하는 반면, 주 메모리(DRAM)는 수십 나노초에서 수백 나노초의 지연 시간을 갖는다. 이러한 속도 차이는 CPU가 데이터를 기다리며 유휴 상태로 있는 시간을 증가시켜 전체 시스템 성능을 저하시킨다.

이 문제를 해결하기 위해 캐시 메모리가 도입되었다. 캐시 메모리는 CPU와 주 메모리 사이에 위치한 고속 임시 저장 장치로, 자주 사용되는 데이터를 임시로 저장하여 CPU가 더 빠르게 접근할 수 있도록 한다. 캐시 메모리는 주 메모리보다 훨씬 빠른 속도로 동작하며, 일반적으로 SRAM(Static RAM)으로 구현된다. SRAM은 DRAM에 비해 빠른 접근 시간을 제공하지만, 비용이 높고 밀도가 낮아 용량이 제한적이다.

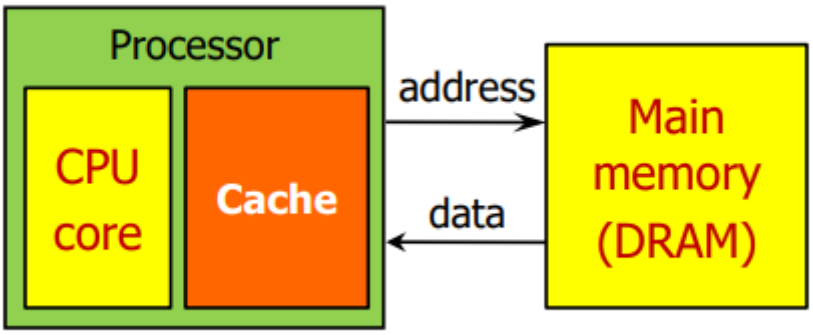
캐시 메모리는 두 가지 주요 원리를 기반으로 설계된다: 시간적 지역성과 공간적 지역성이다. 시간적 지역성은 최근에 접근한 데이터가 곧 다시 접근될 가능성이 높다는 원리이며, 공간적 지역성은 인접한 데이터가 곧 접근될 가능성이 높다는 원리이다. 이러한 원리를 바탕으로 캐시 메모리는 자주 사용되는 데이터와 그 주변 데이터를 저장하여 CPU의 메모리 접근 시간을 최소화한다.

캐시 메모리는 단순히 데이터 접근 속도를 높이는 것 이상의 중요한 역할을 한다. 현대의 멀티코어 프로세서 환경에서 각 코어는 독립적으로 데이터를 처리하지만, 메모리 자원을 공유해야 한다. 이 과정에서 발생할 수 있는 데이터 일관성 문제를 해결하기 위해 캐시 일관성(Coherency) 메커니즘이 필요하다. 캐시 일관성은 여러 코어가 동일한 데이터를 캐시할 때 발생하는 데이터 불일치 문제를 해결하는 데 중요한 역할을 한다.

또한, 캐시 메모리는 다양한 응용 프로그램의 성능을 최적화하는 데 중요한 도구이다. 예를 들어, 데이터베이스 시스템에서는 자주 조회되는 데이터나 인덱스를 캐시에 저장함으로써 쿼리 성능을 크게 향상시킬 수 있다. 그래픽 처리 장치(GPU)에서도 캐시 메모리를 활용하여 대량의 그래픽 데이터를 빠르게 처리할 수 있다.

결론적으로, 캐시 메모리는 현대 컴퓨터 시스템에서 필수적인 구성 요소로, 시스템 성능을 극대화하는 데 중요한 역할을 한다. 본 보고서는 이러한 캐시 메모리의 개념과 중요성을 상세히 설명하고, 효과적인 캐시 설계와 최적화 방법을 제시함으로써, 캐시 메모리를 효율적으로 활용하는 방법을 제시하고자 한다. 캐시 메모리에 대한 깊이 있는 이해는 고성능 컴퓨팅 환경에서의 경쟁력을 강화하는 데 중요한 밑거름이 될 것이다.

2.1 Cache



[1] Cache

캐시 메모리는 CPU와 주 메모리(DRAM) 사이에 위치한 고속 임시 저장 장치이다. 이는 컴퓨터 시스템에서 매우 중요한 역할을 한다. CPU는 매우 빠른 속도로 명령어를 처리하지만, 주 메모리 접근 시간은 상대적으로 느리다. 이로 인해 CPU가 데이터를 기다리며 유휴 상태로 있는 시간이 발생하게 된다. 이러한 지연 시간을 줄이기 위해 캐시 메모리가 도입되었다.

캐시 메모리는 주 메모리보다 훨씬 빠른 속도로 데이터를 읽고 쓸 수 있다. 주 메모리가 DRAM으로 구성되는 반면, 캐시 메모리는 일반적으로 SRAM(Static RAM)으로 구성된다. SRAM은 DRAM에 비해 훨씬 빠른 접근 시간을 제공하지만, 비용이 높고 밀도가 낮아 저장 용량이 제한적이다. 캐시 메모리는 이러한 특성을 활용하여, 최근에 사용된 데이터를 임시로 저장하고, CPU가 필요할 때 빠르게 접근할 수 있도록 한다.

캐시 메모리는 CPU와 주 메모리 간의 데이터 전송을 효율적으로 관리함으로써 전반적인 시스템 성능을 향상시킨다. 예를 들어, CPU가 이전에 사용한 데이터를 다시 필요로 할 때, 이 데이터가 캐시에 저장되어 있다면 주 메모리에 접근할 필요 없이 캐시에서 빠르게 데이터를 읽을 수 있다. 이를 통해 메모리 접근 시 발생하는 지연을 최소화하고, CPU가 보다 원활하게 작업을 수행할 수 있도록 지원한다.

캐시 메모리는 또한 시스템의 전력 효율성을 높이는 데 기여한다. CPU가 주 메모리에 접근하는 횟수를 줄임으로써, 전체 시스템의 전력 소모를 줄일 수 있다. 이는 특히 모바일 기기와 같이 배터리 수명이 중요한 시스템에서 매우 중요한 장점이다.

지역성(Locality)

지역성의 개념은 캐시 메모리 설계의 핵심 원리 중 하나이다. 지역성은 프로그램이 메모리를 접근하는 패턴을 설명하는데, 이를 통해 캐시의 효율성을 극대화할 수 있다. 지역성은 크게 두 가지 주요 형태로 나뉜다: 시간적 지역성과 공간적 지역성이다.

시간적 지역성(Temporal Locality)

시간적 지역성은 최근에 접근한 데이터가 곧 다시 접근될 가능성이 높다는 원리이다. 예를 들어, 프로그램에서 루프 내에서 사용되는 변수는 반복적으로 접근되므로 시간적 지역성이 높다. 루프 내에서 변수의 값은 여러 번 읽히거나 쓰여지게 되므로, 이 변수의 데이터가 캐시에 저장되어 있다면 반복적인 메모리 접근 시간을 줄일 수 있다. 시간적

지역성을 효과적으로 활용하면, 캐시의 히트율(hit rate)을 높일 수 있으며, 이는 전반적인 시스템 성능 향상으로 이어진다.

공간적 지역성(Spatial Locality)

공간적 지역성은 인접한 데이터가 곧 접근될 가능성이 높다는 원리이다. 예를 들어, 배열의 요소를 순차적으로 접근하는 경우가 이에 해당된다. 한 배열 요소를 접근한 후, 그와 인접한 요소를 곧바로 접근할 가능성이 높다. 이러한 패턴을 활용하여, 캐시는 특정 데이터뿐만 아니라 그 주변의 데이터를 함께 저장함으로써 이후의 메모리 접근을 최적화할 수 있다. 예를 들어, 배열의 첫 번째 요소를 캐시에 로드할 때, 그 주변 요소들도 함께 로드하여 다음 접근 시 빠르게 데이터를 제공할 수 있다.

지역성의 중요성

캐시 메모리 설계에서 지역성의 개념은 매우 중요하다. 시간적 지역성과 공간적 지역성을 효과적으로 활용함으로써, 캐시는 데이터를 효율적으로 저장하고 관리할 수 있다. 이는 캐시의 히트율을 높이고, 미스율(miss rate)을 줄이며, 결과적으로 전체 시스템의 성능을 향상시키는 데 기여한다.

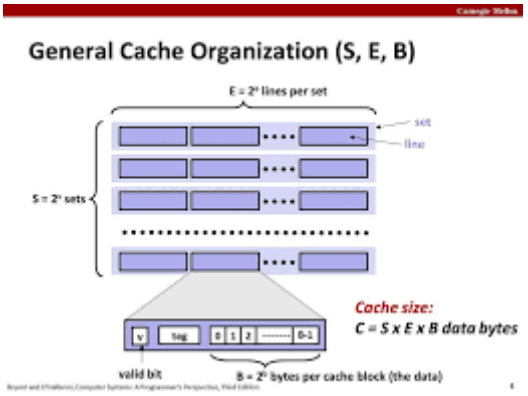
캐시 메모리는 또한 지역성을 활용하여 데이터 일관성을 유지하는 데도 중요한 역할을 한다. 예를 들어, 멀티코어 시스템에서 여러 코어가 동일한 데이터를 공유할 때, 각 코어는 자신의 캐시를 사용하여 데이터를 빠르게 접근할 수 있다. 이 과정에서 데이터 일관성을 유지하기 위해 캐시 일관성 메커니즘이 필요하며, 이는 지역성을 고려한 설계를 통해 효율적으로 구현될 수 있다.

결론적으로, 캐시 메모리는 CPU와 주 메모리 사이의 속도 차이를 줄이고, 시스템 성능을 최적화하는 데 필수적인 요소이다. 캐시 메모리의 정의와 지역성의 개념을 깊이 이해함으로써, 보다 효율적인 캐시 설계와 최적화 방법을 개발할 수 있다. 이는 현대 컴퓨터 시스템에서 높은 성능과 효율성을 달성하는 데 중요한 역할을 한다.

2.2 How the Cache Works

캐시 메모리는 CPU와 주 메모리 사이에서 데이터 접근 속도를 최적화하기 위해 여러 단계의 계층 구조로 구성된다. 각 계층은 속도와 크기에서 차이가 있으며, 이를 통해

데이터 접근 효율을 극대화한다. 캐시 구조의 이해는 시스템 성능 최적화에 중요한 역할을 한다.



[2] 캐시의 구조

캐시 계층 구조

캐시 메모리는 일반적으로 L1, L2, L3의 여러 레벨로 구성되며, 각 레벨은 고유한 특징과 역할을 가지고 있다.

L1 캐시: L1 캐시는 CPU 코어에 가장 가까운 곳에 위치한 캐시로, 매우 빠른 접근 속도를 제공한다. L1 캐시는 CPU의 명령어와 데이터를 처리하는 데 있어서 가장 중요한 역할을 한다. 일반적으로 L1 캐시는 두 부분으로 나뉜다: 명령어 캐시와 데이터 캐시. 명령어 캐시는 실행할 명령어를 저장하고, 데이터 캐시는 데이터를 저장한다. L1 캐시는 크기가 작지만, 그만큼 빠른 접근 시간을 제공하여 CPU가 자주 필요로 하는 데이터를 빠르게 사용할 수 있게 한다. 보통 L1 캐시의 크기는 32KB에서 64KB 정도이다.

L2 캐시: L2 캐시는 L1 캐시보다 크기가 크고 접근 속도는 약간 느리다. L2 캐시는 각 CPU 코어별로 독립적으로 존재하거나, 코어 간에 공유되기도 한다. L2 캐시는 L1 캐시의 미스를 보완하는 역할을 하며, 더 많은 데이터를 저장할 수 있어 자주 사용되는 데이터를 더욱 효과적으로 관리할 수 있다. 일반적으로 L2 캐시의 크기는 256KB에서 512KB 정도이다. L2 캐시는 L1 캐시보다 느리지만, 여전히 주 메모리보다 훨씬 빠른 속도를 제공한다.

L3 캐시: L3 캐시는 여러 CPU 코어가 공유하는 형태로 존재하며, L2 캐시보다 크고 접근 속도는 더 느리다. L3 캐시는 모든 코어가 공유할 수 있는 큰 캐시 공간을 제공하여, 여러 코어 간의 데이터 일관성을 유지하는 데 중요한 역할을 한다. L3 캐시는 시스템 전체의

성능을 최적화하는 데 중요한 역할을 하며, 일반적으로 몇 메가바이트(MB)에서 수십 메가바이트까지 크기를 가진다. L3 캐시는 L1과 L2 캐시의 미스를 보완하며, 주 메모리 접근을 최소화하여 전체 시스템 성능을 향상시킨다.

캐시 라인(Cache Line)

캐시의 기본 저장 단위는 캐시 라인이라고 한다. 캐시 라인은 일반적으로 64바이트 크기로 설정되며, 주 메모리의 특정 블록을 캐시에 저장할 때 사용된다. 캐시 라인은 데이터를 효율적으로 저장하고 관리하기 위한 기본 단위로, 캐시가 데이터를 읽거나 쓸 때 사용된다. 캐시 라인의 크기는 캐시의 효율성과 성능에 큰 영향을 미친다. 너무 작은 캐시 라인은 메모리 접근 횟수를 증가시켜 성능 저하를 초래할 수 있고, 너무 큰 캐시 라인은 캐시의 용량을 비효율적으로 사용할 수 있다.

태그(Tag), 인덱스(Index), 오프셋(Offset)

캐시에서 데이터를 찾기 위한 주소는 태그, 인덱스, 오프셋 세 가지 구성 요소로 나뉜다. 이들 구성 요소는 캐시가 데이터를 효율적으로 관리하고 빠르게 접근할 수 있도록 돕는다.

태그(Tag): 태그는 메모리 블록의 고유 식별자로, 캐시의 특정 라인에 저장된 데이터를 식별하는 데 사용된다. 태그는 메모리 주소의 상위 비트를 사용하여 구성되며, 캐시가 특정 데이터 블록을 저장할 때 이 태그를 사용하여 데이터를 식별한다. 예를 들어, 캐시가 메모리 주소 0x12345678에 있는 데이터를 저장할 때, 이 주소의 상위 비트가 태그로 사용되어 해당 데이터가 캐시의 어느 위치에 저장되어 있는지를 확인할 수 있다.

인덱스(Index): 인덱스는 캐시의 특정 세트를 지정하는 데 사용되며, 데이터가 저장된 캐시 라인을 찾는 데 사용된다. 인덱스는 메모리 주소의 중간 비트를 사용하여 구성되며, 캐시의 각 세트는 여러 개의 캐시 라인을 포함한다. 인덱스는 캐시가 데이터를 저장하거나 검색할 때, 해당 데이터가 어느 세트에 위치하는지를 결정하는 데 중요한 역할을 한다. 예를 들어, 메모리 주소의 중간 비트가 인덱스 값 0x3을 나타내면, 캐시는 세트 3에서 해당 데이터를 검색하게 된다.

오프셋(Offset): 오프셋은 캐시 라인 내에서 데이터를 지정하는 데 사용된다. 오프셋은 메모리 주소의 하위 비트를 사용하여 구성되며, 캐시 라인 내의 특정 위치를 지정한다. 캐시 라인은 여러 바이트로 구성되므로, 오프셋을 사용하여 캐시 라인 내에서 특정

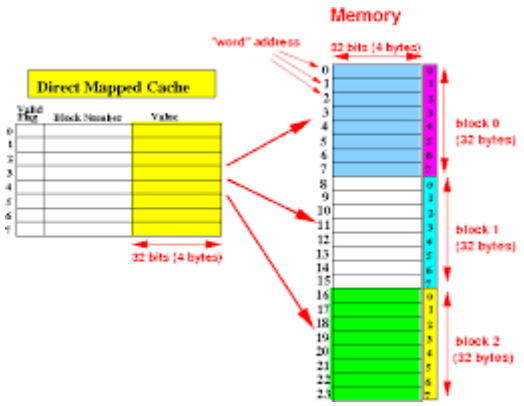
바이트를 선택할 수 있다. 예를 들어, 캐시 라인의 크기가 64바이트이고, 오프셋 값이 0x1F이면, 캐시 라인의 31번째 바이트를 가리킨다.

캐시 메모리는 CPU와 주 메모리 간의 속도 차이를 줄이고, 시스템 성능을 최적화하기 위해 설계된 고속 임시 저장 장치이다. 여러 레벨로 구성된 캐시 계층 구조, 효율적인 데이터 저장과 관리를 위한 캐시 라인, 태그, 인덱스, 오프셋 등의 구성 요소는 캐시 메모리가 높은 성능을 제공하는 데 중요한 역할을 한다. 캐시 메모리의 구조를 깊이 이해함으로써, 효율적인 캐시 설계와 최적화 방법을 개발할 수 있으며, 이는 현대 컴퓨터 시스템의 성능을 극대화하는 데 필수적인 요소이다.

2.3 Cache Design Method

캐시 메모리는 시스템 성능을 극대화하기 위해 다양한 설계 방식을 사용한다. 각 설계 방식은 구현의 복잡성과 성능 특성이 다르며, 특정 상황에 적합한 방식을 선택하는 것이 중요하다. 캐시 설계 방식에는 직접 매핑 캐시(Direct-Mapped Cache), 세트 연관 캐시(Set-Associative Cache), 완전 연관 캐시(Fully Associative Cache) 세 가지 주요 방식이 있다.

2.3.1 Direct-Mapped Cache



[3] Direct-Mapped Cache

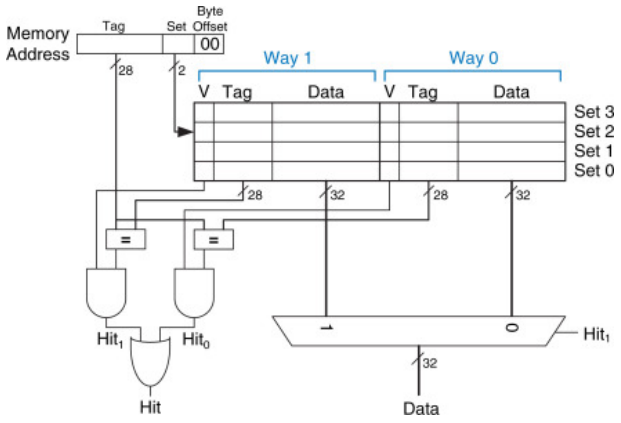
직접 매핑 캐시는 메모리의 각 블록이 단일 캐시 라인에 매핑되는 방식이다. 이 방식은 구현이 간단하고 빠르다. 메모리 주소는 세 부분으로 나뉜다: 태그(Tag),

인덱스(Index), 오프셋(Offset). 인덱스는 캐시 라인을 지정하고, 태그는 메모리 블록을 식별하는 데 사용된다. 오프셋은 캐시 라인 내의 특정 바이트를 가리킨다.

직접 매핑 캐시의 장점은 구조가 단순하여 빠른 검색이 가능하다는 점이다. 캐시 라인을 찾는 과정이 간단하여 데이터 접근 속도가 빠르다. 또한, 하드웨어 구현이 비교적 쉬워 비용이 낮다.

그러나 직접 매핑 캐시는 충돌 미스(Conflict Miss) 발생 가능성이 높다. 동일한 인덱스를 가진 여러 메모리 블록이 동일한 캐시 라인에 매핑되기 때문에, 한 라인에 여러 데이터가 경쟁하는 상황이 자주 발생한다. 이로 인해 특정 패턴의 데이터 접근이 반복될 경우, 캐시 미스가 증가하여 성능 저하를 초래할 수 있다.

2.3.2 Set-Associative Cache



[4] Set-Associative Cache

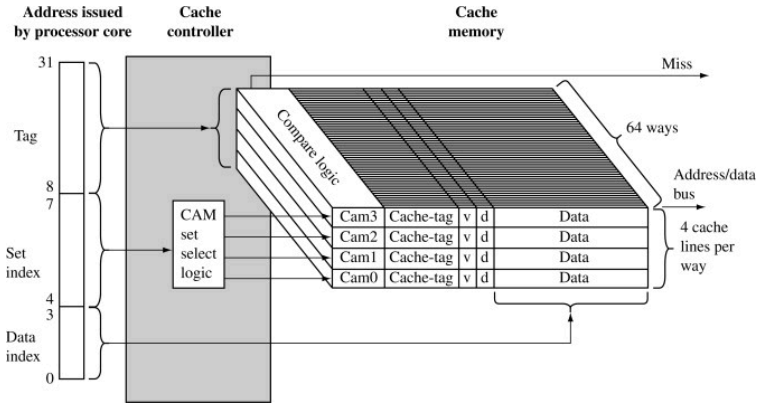
세트 연관 캐시는 각 블록이 여러 캐시 라인 중 하나에 매핑될 수 있는 방식이다. 캐시는 여러 세트로 나누어지고, 각 세트는 여러 라인을 포함한다. 메모리 주소는 세 부분으로 나뉜다: 태그(Tag), 세트 인덱스(Set Index), 오프셋(Offset). 세트 인덱스는 특정 세트를 지정하고, 태그는 세트 내에서 데이터를 식별하는 데 사용된다.

세트 연관 캐시는 충돌 미스를 줄일 수 있는 장점이 있다. 각 세트는 여러 라인을 포함하므로, 동일한 인덱스를 가진 여러 블록이 한 세트 내의 다양한 라인에 저장될 수 있다. 이로 인해 특정 데이터 접근 패턴에서의 충돌 미스 발생 가능성이 낮아진다.

세트 연관 캐시의 구현 복잡성은 중간 정도이다. 각 세트 내에서 데이터를 찾기 위해 태그를 비교해야 하므로, 하드웨어 구현이 직접 매핑 캐시에 비해 복잡하다. 그러나 이 방식은 충돌 미스를 줄여 전체적인 캐시 성능을 향상시킬 수 있다.

세트 연관 캐시는 일반적으로 2-way, 4-way, 8-way 등의 방식으로 구현된다. 예를 들어, 4-way 세트 연관 캐시는 각 세트에 4개의 캐시 라인을 포함하며, 메모리 블록이 4개의 라인 중 하나에 저장될 수 있다. 이러한 방식은 충돌 미스를 효과적으로 줄이면서도 구현 복잡성을 적절히 유지한다.

2.3.3 Fully Associative Cache



[5] Fully Associative Cache

완전 연관 캐시는 메모리의 모든 블록이 캐시의 어떤 라인에나 저장될 수 있는 방식이다. 메모리 주소는 태그(Tag)와 오프셋(Offset)으로 나뉘며, 인덱스는 사용되지 않는다. 태그는 캐시의 모든 라인을 검색하여 데이터를 식별하는 데 사용된다.

완전 연관 캐시의 가장 큰 장점은 충돌 미스가 거의 발생하지 않는다는 점이다. 모든 메모리 블록이 캐시의 모든 라인에 저장될 수 있으므로, 특정 인덱스에 여러 데이터가 매핑되는 충돌 미스 문제가 발생하지 않는다. 이로 인해 캐시 히트율(hit rate)이 높아지고, 전반적인 시스템 성능이 향상된다.

그러나 완전 연관 캐시는 구현이 매우 복잡하고 비용이 높다. 캐시의 모든 라인을 검색해야 하므로 하드웨어가 복잡해지고, 태그 비교를 위한 많은 리소스가 필요하다. 대규모 캐시에서는 이러한 복잡성이 기하급수적으로 증가하므로, 완전 연관 캐시는 작은 크기의 캐시에 주로 사용된다.

캐시 설계 방식은 시스템 성능에 큰 영향을 미치며, 각 방식은 고유한 장점과 단점을 가지고 있다. 직접 매핑 캐시는 구조가 단순하고 빠르지만, 충돌 미스가 발생할 가능성이 높다. 세트 연관 캐시는 충돌 미스를 줄일 수 있지만, 구현 복잡성이 중간 정도이다. 완전 연관 캐시는 충돌 미스가 거의 발생하지 않지만, 구현이 복잡하고 비용이 높다.

시스템 설계자는 특정 응용 프로그램과 사용 환경에 적합한 캐시 설계 방식을 선택하여, 최적의 성능을 달성할 수 있도록 해야 한다. 이를 통해 캐시 메모리의 효율성을 극대화하고, 전반적인 시스템 성능을 향상시킬 수 있다.

2.4 Type of cache miss

캐시 미스는 캐시에 필요한 데이터가 존재하지 않을 때 발생하는 상황으로, 이는 시스템 성능에 부정적인 영향을 미친다. 캐시 미스는 여러 유형으로 분류되며, 각 유형은 다른 원인과 특성을 가지고 있다. 주요 캐시 미스 유형에는 차가운 미스(Cold Miss), 용량 미스(Capacity Miss), 충돌 미스(Conflict Miss), 일관성 미스(Coherency Miss)가 있다.

차가운 미스(Cold Miss)

차가운 미스는 처음 캐시에 접근할 때 발생하는 미스로, 데이터가 아직 캐시에 저장되지 않은 경우에 발생한다. 이 유형의 미스는 시스템이 처음 부팅되거나 특정 데이터에 처음 접근할 때 주로 나타난다. 차가운 미스는 캐시가 아직 데이터에 대한 정보가 없어 발생하는 자연스러운 현상이다.

차가운 미스를 줄이기 위해 다양한 기술이 사용될 수 있다. 예를 들어, 프리패칭(Pre-fetching) 기법을 사용하여 예상되는 데이터 접근 패턴을 기반으로 데이터를 미리 캐시에 로드할 수 있다. 또한, 효율적인 초기화 전략을 통해 시스템이 시작될 때 중요한 데이터를 미리 로드함으로써 차가운 미스를 줄일 수 있다. 그러나 차가운 미스는 완전히 제거하기 어렵기 때문에, 이를 줄이기 위한 지속적인 노력이 필요하다.

용량 미스(Capacity Miss)

용량 미스는 캐시가 가득 차서 추가 데이터를 저장할 공간이 없을 때 발생하는 미스이다. 캐시의 크기가 제한적이기 때문에, 자주 사용되지 않는 데이터가 캐시에서 제거되고

새로운 데이터가 저장된다. 이 과정에서 용량 미스가 발생할 수 있다. 용량 미스는 캐시 크기와 데이터 접근 패턴에 크게 영향을 받는다.

용량 미스를 줄이기 위해 캐시 크기를 늘리는 것이 가장 직접적인 해결책이다. 그러나 캐시 크기를 늘리는 것은 비용과 전력 소비를 증가시키므로, 현실적인 한계가 있다. 따라서 용량 미스를 최소화하기 위해 다양한 최적화 기술이 사용된다. 예를 들어, 데이터 접근 패턴을 분석하여 자주 사용되는 데이터만 캐시에 저장하는 알고리즘을 적용할 수 있다. 또한, 캐시 교체 정책을 최적화하여 불필요한 데이터 교체를 줄이고, 캐시의 효율성을 높일 수 있다.

충돌 미스(Conflict Miss)

충돌 미스는 동일한 인덱스에 여러 데이터가 매핑될 때 발생하는 미스이다. 특히 직접 매핑 캐시에서 자주 발생하는 문제로, 하나의 캐시 라인에 여러 데이터가 경쟁하게 되어 발생한다. 충돌 미스는 캐시의 인덱스가 한정된 범위 내에서 동작하기 때문에 발생하며, 이는 특정 패턴의 데이터 접근이 반복될 때 더욱 심각해진다.

충돌 미스를 줄이기 위해 다양한 기법이 사용된다. 가장 일반적인 방법은 세트 연관 캐시를 사용하는 것이다. 세트 연관 캐시는 각 인덱스에 여러 캐시 라인을 할당하여, 동일한 인덱스에 여러 데이터가 매핑되는 문제를 줄일 수 있다. 또한, 캐시 교체 정책을 개선하여 충돌 미스를 최소화할 수 있다. 예를 들어, LRU(Least Recently Used) 알고리즘을 사용하여 자주 사용되지 않는 데이터를 교체함으로써 충돌 미스를 줄일 수 있다.

일관성 미스(Coherency Miss)

일관성 미스는 여러 CPU 코어가 동일한 캐시 라인을 공유할 때 발생하는 미스이다. 멀티코어 시스템에서는 각 코어가 독립적으로 데이터를 처리하지만, 공통 데이터를 접근할 때 데이터 일관성을 유지해야 한다. 이 과정에서 데이터가 여러 캐시에 분산되어 있을 경우, 한 캐시에서 데이터가 업데이트되면 다른 캐시의 데이터도 동기화되어야 한다. 이러한 동기화 과정에서 일관성 미스가 발생할 수 있다.

일관성 미스를 줄이기 위해 캐시 일관성 메커니즘이 사용된다. 대표적인 일관성 메커니즘으로는 MSI(MESI) 프로토콜이 있다. MSI 프로토콜은 각 캐시 라인의 상태를 Modified, Shared, Invalid의 세 가지로 관리하여 데이터 일관성을 유지한다. MESI

프로토콜은 Exclusive 상태를 추가하여, 데이터를 보다 효율적으로 관리할 수 있다. 이러한 일관성 메커니즘은 멀티코어 시스템에서 데이터 일관성을 유지하면서 일관성 미스를 최소화하는 데 중요한 역할을 한다.

캐시 미스는 캐시 메모리의 성능을 저하시키는 주요 요인으로, 각 유형의 미스는 다른 원인과 특성을 가지고 있다. 차가운 미스는 데이터가 처음 캐시에 접근할 때 발생하며, 용량 미스는 캐시가 가득 찼을 때 발생한다. 충돌 미스는 동일한 인덱스에 여러 데이터가 매핑될 때 발생하며, 일관성 미스는 여러 CPU 코어가 동일한 캐시 라인을 공유할 때 발생한다.

각 유형의 캐시 미스를 줄이기 위해 다양한 최적화 기술이 사용된다. 프리패칭, 캐시 크기 조정, 세트 연관 캐시, 일관성 메커니즘 등은 캐시 미스를 최소화하고, 시스템 성능을 향상시키는 데 중요한 역할을 한다. 캐시 미스의 원인을 이해하고 적절한 대책을 적용함으로써, 캐시 메모리의 효율성을 극대화하고, 전반적인 시스템 성능을 최적화할 수 있다.

2.5 Cache Replacement Policy

캐시 교체 정책은 캐시 메모리가 가득 찼을 때 어떤 데이터를 교체할지 결정하는 알고리즘을 의미한다. 캐시 교체 정책은 캐시의 효율성과 성능에 큰 영향을 미치며, 다양한 알고리즘이 사용된다. 주요 캐시 교체 정책에는 LRU(Least Recently Used), FIFO(First In First Out), 랜덤 교체(Random Replacement), 두 번째 기회 알고리즘(Second Chance Algorithm, SCA)가 있다. 각 정책은 고유한 장점과 단점을 가지고 있으며, 특정 상황에 적합한 방식을 선택하는 것이 중요하다.

2.5.1 LRU

LRU 교체 정책은 가장 최근에 사용되지 않은 데이터를 교체하는 방식이다. 이 정책은 시간적 지역성을 잘 반영하며, 최근에 사용된 데이터가 다시 사용될 가능성이 높다는 가정에 기반한다. LRU 알고리즘은 캐시 내의 각 데이터가 마지막으로 사용된 시점을 추적하여, 가장 오래 사용되지 않은 데이터를 교체한다.

LRU의 장점은 시간적 지역성을 효과적으로 반영하여 높은 캐시 히트율을 달성할 수 있다는 점이다. 데이터 접근 패턴이 시간적 지역성을 갖는 경우, LRU는 매우 효율적으로 작동한다. 그러나 LRU의 단점은 구현이 복잡하다는 점이다. 각 데이터의 사용 시점을

추적해야 하므로, 추가적인 메모리와 연산이 필요하다. 특히 대규모 캐시에서는 이러한 오버헤드가 성능에 부정적인 영향을 미칠 수 있다.

2.5.2 FIFO

FIFO 교체 정책은 가장 오래된 데이터를 교체하는 방식이다. 이 정책은 데이터가 캐시에 들어온 순서대로 교체를 결정하며, 구현이 매우 간단하다. FIFO 알고리즘은 큐(Queue) 자료구조를 사용하여, 가장 먼저 들어온 데이터를 가장 먼저 제거한다.

FIFO의 장점은 구현의 간단함이다. 데이터의 진입 순서만 추적하면 되므로, 추가적인 연산이 거의 필요 없다. 그러나 FIFO의 단점은 성능이 상대적으로 떨어질 수 있다는 점이다. FIFO는 시간적 지역성을 반영하지 않기 때문에, 자주 사용되는 데이터가 교체될 가능성이 높아 캐시 히트율이 낮아질 수 있다. 특히 데이터 접근 패턴이 자주 변경되는 경우, FIFO는 비효율적으로 작동할 수 있다.

2.5.3 Random Replacement

랜덤 교체 정책은 무작위로 데이터를 교체하는 방식이다. 이 정책은 특정 데이터를 선택하지 않고, 무작위로 선택된 데이터를 교체함으로써 간단하게 구현할 수 있다. 랜덤 교체 알고리즘은 추가적인 메모리나 복잡한 연산이 필요 없으며, 단순한 난수 생성기를 사용하여 교체할 데이터를 결정한다.

랜덤 교체의 장점은 구현의 간단함과 예측 가능성이 낮다는 점이다. 예측 가능성이 낮다는 것은 특정 패턴에 의한 성능 저하를 방지할 수 있음을 의미한다. 그러나 랜덤 교체의 단점은 성능이 일정하지 않다는 점이다. 무작위로 데이터를 교체하기 때문에, 자주 사용되는 데이터가 교체될 가능성이 있으며, 이는 캐시 히트율을 낮출 수 있다.

2.5.4 SCA

두 번째 기회 알고리즘(SCA)은 최근 접근 여부를 고려하여 두 번째 기회를 부여하는 방식이다. 이 정책은 LRU와 유사하지만, 구현이 더 간단하다. SCA는 각 데이터에 접근 비트를 추가하여, 데이터가 최근에 접근되었는지 여부를 기록한다. 교체 시점이 되면, 접근 비트가 0인 데이터를 교체하고, 1인 데이터는 비트를 0으로 설정하고 다음 데이터를 검사한다.

SCA의 장점은 LRU의 시간적 지역성을 반영하면서도 구현이 상대적으로 간단하다는 점이다. 접근 비트만을 사용하여 데이터를 관리하므로, 추가적인 메모리와 연산이 최소화된다. 또한, 자주 사용되는 데이터는 두 번째 기회를 받기 때문에, 캐시 히트율을 높일 수 있다. 그러나 SCA의 단점은 완벽한 LRU 구현에 비해 성능이 다소 낮을 수 있다는 점이다. 접근 비트만으로 데이터를 관리하기 때문에, 정확한 사용 순서를 추적하지 못한다.

캐시 교체 정책은 캐시 메모리의 성능을 좌우하는 중요한 요소이다. LRU는 시간적 지역성을 잘 반영하여 높은 성능을 제공하지만, 구현이 복잡하다. FIFO는 구현이 간단하지만, 성능이 떨어질 수 있다. 랜덤 교체는 예측 가능성이 낮아 특정 패턴의 성능 저하를 방지할 수 있지만, 성능이 일정하지 않다. SCA는 LRU와 유사한 성능을 제공하면서도 구현이 간단하여 효율적이다.

시스템 설계자는 응용 프로그램의 특성과 데이터 접근 패턴을 고려하여 적절한 캐시 교체 정책을 선택해야 한다. 이를 통해 캐시의 효율성을 극대화하고, 전반적인 시스템 성능을 향상시킬 수 있다. 캐시 교체 정책의 선택은 고성능 컴퓨팅 시스템에서 중요한 결정 요소이며, 다양한 상황에 맞는 최적의 알고리즘을 적용함으로써 최상의 성능을 달성할 수 있다.

2.6 Read and write cache

캐시 메모리는 컴퓨터 시스템에서 데이터 접근 속도를 최적화하기 위해 사용되며, 데이터 읽기와 쓰기 과정에서 중요한 역할을 한다. 캐시에서 데이터를 읽고 쓰는 방식은 시스템 성능과 데이터 일관성에 큰 영향을 미친다. 캐시 읽기와 쓰기의 기본 원리를 이해하는 것은 시스템 설계와 최적화에 있어 필수적이다.

읽기(Read)

캐시에서 데이터를 읽는 과정은 다음과 같다:

1. **캐시 히트**: 캐시에서 필요한 데이터를 찾고, 해당 데이터가 캐시에 존재하는 경우 이를 캐시 히트(hit)라고 한다. 캐시 히트 시, CPU는 캐시에서 데이터를 직접 읽어와 빠르게 접근할 수 있다. 이로 인해 메모리 접근 시간이 크게 단축된다.

2. **캐시 미스**: 필요한 데이터가 캐시에 없는 경우를 캐시 미스(miss)라고 한다. 캐시 미스가 발생하면, 주 메모리에서 데이터를 가져와 캐시에 저장한 후 CPU가 데이터를 읽는다. 캐시 미스 시, 다음과 같은 단계가 수행된다:
    - **데이터 검색**: 주 메모리에서 필요한 데이터를 검색한다.
    - **데이터 로드**: 검색된 데이터를 캐시로 로드한다. 이때, 캐시가 가득 차 있으면 기존 데이터를 교체해야 한다.
    - **데이터 제공**: 캐시에 로드된 데이터를 CPU에 제공하여 읽기를 완료한다.

캐시 히트율(hit rate)은 캐시 성능의 중요한 지표 중 하나로, 캐시에서 데이터가 성공적으로 검색되는 비율을 나타낸다. 높은 캐시 히트율은 시스템 성능을 크게 향상시킨다.

쓰기(Write)

캐시에 데이터를 쓰는 방식은 크게 두 가지로 나뉜다: Write-Through와 Write-Back. 각 방식은 데이터 일관성과 쓰기 속도에 따라 장단점이 있다.

2.6.1 Write-Through

Write-Through 방식은 데이터를 캐시와 주 메모리에 동시에 쓰는 방식이다. 이 방식의 주요 단계는 다음과 같다:

1. **데이터 쓰기**: CPU가 데이터를 캐시에 쓰면, 동시에 동일한 데이터를 주 메모리에도 쓴다.
2. **일관성 유지**: 데이터가 항상 캐시와 주 메모리에 동시에 존재하므로, 데이터 일관성이 높다. 캐시와 주 메모리의 데이터가 항상 동일하게 유지된다.

장점

- **데이터 일관성**: 캐시와 주 메모리의 데이터가 항상 동일하므로, 일관성이 높다. 이는 멀티코어 시스템에서 특히 유리하다.
- **간단한 구현**: 데이터 일관성을 유지하기 위한 추가적인 메커니즘이 필요 없다.

단점

- **쓰기 속도 저하**: 데이터를 캐시와 주 메모리에 동시에 써야 하므로, 쓰기 속도가 느릴 수 있다. 이는 주 메모리의 접근 시간이 상대적으로 느리기 때문이다.
- **높은 메모리 트래픽**: 주 메모리에 대한 쓰기 작업이 많아지므로, 메모리 트래픽이 증가할 수 있다.

2.6.2 Write-Back

Write-Back 방식은 데이터를 캐시에만 쓰고, 캐시 라인 교체 시 주 메모리에 쓰는 방식이다. 이 방식의 주요 단계는 다음과 같다:

1. **데이터 쓰기**: CPU가 데이터를 캐시에 쓰면, 주 메모리에는 즉시 쓰지 않고 캐시에만 저장된다.
2. **더티 비트 설정**: 데이터가 수정된 캐시 라인은 더티 비트(dirty bit)가 설정된다. 이는 해당 캐시 라인이 수정되었음을 나타낸다.
3. **캐시 라인 교체**: 캐시 라인이 교체될 때, 더티 비트가 설정된 경우 수정된 데이터를 주 메모리에 쓴다. 그렇지 않은 경우에는 주 메모리에 쓰지 않는다.

장점

- **빠른 쓰기 속도**: 데이터를 캐시에만 쓰므로, 쓰기 속도가 빠르다. 주 메모리에 대한 접근이 줄어들어 전반적인 성능이 향상된다.
- **낮은 메모리 트래픽**: 주 메모리에 대한 쓰기 작업이 줄어들어, 메모리 트래픽이 감소한다.

단점

- **데이터 일관성 문제**: 캐시와 주 메모리의 데이터가 일시적으로 다를 수 있으므로, 일관성 유지에 주의가 필요하다. 멀티코어 시스템에서는 캐시 일관성 메커니즘이 필수적이다.
- **복잡한 구현**: 데이터 일관성을 유지하기 위해 더티 비트와 같은 추가적인 메커니즘이 필요하다.

캐시 일관성(Coherency)

캐시 일관성은 특히 멀티코어 시스템에서 중요한 문제이다. 여러 코어가 동일한 메모리 공간을 캐시에 저장하고 접근할 때, 데이터 일관성을 유지하는 것이 필수적이다. 이를 위해 다양한 캐시 일관성 프로토콜이 사용된다.

MESI 프로토콜

MESI(Modified, Exclusive, Shared, Invalid) 프로토콜은 대표적인 캐시 일관성 유지 메커니즘이다. 각 캐시 라인은 네 가지 상태 중 하나를 가질 수 있다:

1. **Modified**: 캐시에만 존재하며, 주 메모리의 데이터와 다르다.
2. **Exclusive**: 캐시에만 존재하며, 주 메모리의 데이터와 동일하다.
3. **Shared**: 캐시와 주 메모리 모두에 존재하며, 여러 캐시에서 공유될 수 있다.
4. **Invalid**: 캐시에 존재하지 않으며, 주 메모리의 데이터와 일치하지 않는다.

MESI 프로토콜은 캐시 라인의 상태를 관리하여, 데이터 일관성을 유지한다. 이는 멀티코어 시스템에서 데이터 일관성을 효과적으로 유지할 수 있는 방법이다.

캐시 메모리에서의 데이터 읽기와 쓰기는 시스템 성능과 데이터 일관성에 중요한 영향을 미친다. 캐시 히트율을 높이기 위해서는 효율적인 데이터 읽기 메커니즘이 필요하며, 데이터 일관성을 유지하면서 쓰기 속도를 최적화하기 위해서는 적절한 쓰기 정책을 선택하는 것이 중요하다. Write-Through와 Write-Back 방식은 각각 장단점이 있으며, 시스템의 요구사항에 따라 적절한 방식을 선택하여 사용해야 한다. 또한, 멀티코어 시스템에서는 캐시 일관성 메커니즘을 통해 데이터 일관성을 유지하는 것이 필수적이다. 캐시 메모리의 효율적인 관리와 최적화를 통해 전반적인 시스템 성능을 극대화할 수 있다.

2.7 Optimizing Cache Performance

캐시 메모리는 컴퓨터 시스템의 성능을 좌우하는 중요한 요소 중 하나이다. 캐시 성능을 최적화하기 위해 다양한 방법이 사용되며, 이를 통해 캐시 히트율을 높이고, 전반적인 시스템 효율성을 향상시킬 수 있다. 캐시 성능 최적화의 주요 방법에는 캐시 크기 및 라인 크기 조정, 교체 알고리즘 개선, 프리패칭(Pre-fetching)이 있다.

캐시 크기 및 라인 크기 조정

캐시 크기와 라인 크기를 적절히 조정하는 것은 캐시 성능 최적화의 기본적인 방법이다. 캐시 크기는 캐시에 저장할 수 있는 데이터의 총 용량을 의미하며, 라인 크기는 캐시의 기본 저장 단위를 의미한다.

- **캐시 크기**: 캐시의 크기가 너무 작으면, 자주 사용되는 데이터가 캐시에 모두 저장되지 못해 캐시 미스가 증가하게 된다. 반면, 캐시의 크기가 너무 크면, 비용이 증가하고 전력 소모가 늘어나게 된다. 따라서 적절한 캐시 크기를 선택하는 것이 중요하다. 일반적으로, 응용 프로그램의 데이터 접근 패턴을 분석하여 최적의 캐시 크기를 결정한다.
- **라인 크기**: 캐시 라인의 크기가 적절하지 않으면, 성능 저하를 초래할 수 있다. 너무 작은 캐시 라인은 데이터를 효율적으로 저장하지 못해 캐시 미스가 증가할 수 있고, 너무 큰 캐시 라인은 불필요한 데이터를 포함하여 저장 공간을 낭비할 수 있다. 라인 크기는 데이터 접근 패턴을 고려하여 결정해야 한다. 예를 들어, 공간적 지역성이 높은 경우에는 큰 라인 크기가 효과적일 수 있다.

교체 알고리즘 개선

캐시의 교체 알고리즘은 캐시 성능에 큰 영향을 미친다. 효율적인 교체 알고리즘을 통해 캐시 히트율을 높이고, 캐시 미스를 줄일 수 있다. 다양한 교체 알고리즘 중에서 최적의 방식을 선택하는 것이 중요하다.

- LRU (Least Recently Used): 가장 최근에 사용되지 않은 데이터를 교체하는 방식으로, 시간적 지역성을 잘 반영한다. 데이터 접근 패턴이 시간적 지역성을 갖는 경우, LRU는 높은 성능을 제공할 수 있다.
- FIFO (First In First Out): 가장 오래된 데이터를 교체하는 방식으로, 구현이 간단하지만 성능은 상대적으로 떨어질 수 있다. FIFO는 자주 사용되지 않는 데이터가 교체될 가능성이 높아, 캐시 히트율이 낮아질 수 있다.
- **랜덤 교체**(Random Replacement): 무작위로 데이터를 교체하는 방식으로, 간단하지만 예측 가능성이 낮다. 랜덤 교체는 특정 패턴의 성능 저하를 방지할 수 있지만, 일관된 성능을 보장하지는 못한다.
- **두 번째 기회 알고리즘**(Second Chance Algorithm, SCA): 최근 접근 여부를 고려하여 두 번째 기회를 부여하는 방식으로, LRU와 유사하지만 구현이 더 간단하다. 접근 비트를 사용하여 데이터를 관리하므로, 추가적인 메모리와 연산이 최소화된다.

프리패칭(Pre-fetching)

프리패칭은 필요할 것으로 예상되는 데이터를 미리 캐시에 로드하여 캐시 미스를 줄이는 방법이다. 프리패칭의 효과는 예측 정확도에 크게 의존하며, 정확도가 높을수록 캐시 히트율이 증가한다.

- **프리패칭 기법**: 프리패칭 기법에는 여러 가지가 있다. 정적 프리패칭은 프로그램 코드에 의해 결정된 데이터를 미리 로드하는 방식이며, 동적 프리패칭은 실행 중인 프로그램의 데이터 접근 패턴을 분석하여 필요한 데이터를 예측하는 방식이다. 동적 프리패칭은 보다 복잡하지만, 더 높은 정확도를 제공할 수 있다.
- **효과적인 프리패칭**: 효과적인 프리패칭을 위해서는 데이터 접근 패턴을 정확히 예측하는 것이 중요하다. 이를 위해 머신 러닝 알고리즘을 활용한 예측 기법이 사용되기도 한다. 예를 들어, 과거의 데이터 접근 패턴을 학습하여 미래의 접근을 예측할 수 있다.

캐시 성능 최적화는 시스템 성능을 극대화하는 데 중요한 역할을 한다. 적절한 캐시 크기와 라인 크기를 선택하고, 효율적인 교체 알고리즘을 도입하며, 효과적인 프리패칭 기법을 적용함으로써 캐시 히트율을 높일 수 있다. 이러한 최적화 방법을 통해 캐시 미스를 최소화하고, 전반적인 시스템 성능을 향상시킬 수 있다. 캐시 성능 최적화는 지속적인 연구와 개발이 필요한 분야로, 새로운 기술과 알고리즘을 도입하여 보다 높은 성능을 달성하는 것이 중요하다.

3.0 Program

지금부터 캐시의 핵심 원칙들을 반영하여 개발한 프로그램에 대한 설명을 이어가도록 하겠다. 이 프로그램은 MIPS 구조의 단일 사이클 프로세서를 시뮬레이션하며, 캐시 메모리의 역할과 동작을 설명하는 데 초점을 맞추고 있다. 프로그램은 명령어를 메모리에 로드하고, 각 사이클마다 명령어를 가져오고(fetch), 디코드(decode)하며, 실행(execute)하고, 메모리 접근(access memory)을 수행한 뒤, 결과를 레지스터에 기록(write back)하는 단계를 거친다. 이러한 과정을 통해 프로세서의 상태를 갱신하고, 캐시의 동작을 포함한 다양한 메트릭스를 측정한다.

캐시 메모리는 프로세서와 주기억장치 간의 데이터 전송 속도를 향상시키기 위해 사용된다. 프로그램은 L1 캐시를 시뮬레이션하며, 캐시의 매핑 방식과 교체 정책을 사용자가 선택할 수 있도록 설계되어 있다. 매핑 방식에는 직접 매핑(direct-mapped),

2-way, 4-way, 8-way, 완전 연관(fully associative)이 있으며, 교체 정책에는 랜덤(random), FIFO, LRU, 세컨드 찬스 알고리즘(SCA)이 포함된다.

프로세서는 명령어를 실행하는 동안 캐시를 사용하여 데이터를 빠르게 접근할 수 있다. 프로그램의 access_cache 함수는 주어진 주소에 대해 캐시를 검색하여 캐시 히트(cache hit)인지 캐시 미스(cache miss)인지 확인한다. 캐시 히트의 경우, 데이터는 캐시에서 직접 읽거나 쓰게 되며, 캐시 미스의 경우, 데이터를 주기억장치에서 가져와 캐시에 저장한 후 접근하게 된다.

캐시의 초기화는 initialize_cache 함수에서 수행된다. 이 함수는 캐시의 크기, 연관도(associativity), 블록 크기 등을 설정하고, 캐시 라인과 FIFO 큐를 할당한다. 캐시 라인은 각각 유효 비트(valid bit), 더티 비트(dirty bit), 태그(tag), 데이터, 마지막 접근 시간(last access time), FIFO 위치, SCA 비트를 포함한다.

캐시의 검색은 find_cache_line 함수에서 수행된다. 주어진 주소에 대해 해당 세트 인덱스와 태그를 계산하고, 연관도에 따라 해당 세트 내의 각 캐시 라인을 검사하여 일치하는 태그를 찾는다. 일치하는 태그가 있으면 캐시 히트로 간주하며, 그렇지 않으면 캐시 미스로 처리된다.

캐시 미스가 발생하면 select_victim_cache_line 함수를 통해 교체할 캐시 라인을 선택한다. 교체 정책에 따라 랜덤으로 선택하거나, FIFO, LRU, SCA 등의 알고리즘을 사용하여 희생자를 선택한다. 선택된 캐시 라인이 더티 상태인 경우, 해당 데이터를 주기억장치에 쓰는 작업을 수행한 후 새로운 데이터를 캐시로 가져온다.

프로그램은 또한 다양한 메트릭스를 기록하여 성능을 분석한다. 예를 들어, 캐시 히트 수, 캐시 미스 수, 캐시 교체 횟수, 콜드 미스 수, 메모리 접근 시간 등을 측정하고, 최종 실행 결과를 출력한다. 이를 통해 캐시의 성능과 프로세서의 효율성을 평가할 수 있다.

프로그램의 주요 특징 중 하나는 파이프라인 제어 기능이다. 파이프라인은 명령어의 각 단계를 동시에 처리하여 성능을 향상시키는 기법이다. 하지만 데이터 종속성이나 제어 종속성으로 인해 파이프라인 해저드(hazard)가 발생할 수 있다. 이 프로그램은 간단한 데이터 해저드를 감지하고, 필요한 경우 파이프라인을 스톨(stall)하거나 포워딩(forwarding)하여 해결한다.

3.1 Cache Design Method in Program

다양한 캐시 매핑 타입과 교체 정책을 구현하여 시뮬레이션하는 프로그램이다. 각 매핑 타입에 대해 캐시 라인을 선택하고 적중 여부를 확인하는 로직이 포함되어 있다. 주어진 매핑 타입에 따라 캐시를 설정하고, 데이터를 접근하는 방법을 결정한다.

0: Direct-Mapped

- Direct-Mapped Cache는 각 메모리 블록이 캐시의 한 위치에만 매핑될 수 있다. 즉, 특정 메모리 주소는 오직 하나의 캐시 라인에만 저장될 수 있다.
- 가장 단순한 형태의 캐시 매핑 방식이다.
- initialize_cache 함수에서 associativity 값을 1로 설정하여 Direct-Mapped 캐시를 구현한다.

```
int find_cache_line(Cache *cache, uint32_t address, uint32_t *index) {
    uint32_t set_index = (address / cache->block_size) % cache->num_sets;
    uint32_t tag = address / (cache->block_size * cache->num_sets);
    uint32_t base_index = set_index * cache->associativity;

    for (int i = 0; i < cache->associativity; i++) {
        if (cache->lines[base_index + i].valid && cache->lines[base_index + i].tag == tag) {
            *index = base_index + i;
            return 1; // Cache hit
        }
    }
    return 0; // Cache miss
}
```

- Direct-Mapped 캐시는 각 메모리 블록이 캐시의 단일 고정된 위치에만 매핑된다. 즉, 특정 블록은 특정 인덱스의 캐시 라인에만 존재할 수 있다.
- find_cache_line 함수는 주소를 통해 세트 인덱스를 계산하고, 해당 인덱스의 캐시 라인에서 태그를 확인한다.
- select_victim_cache_line 함수는 Direct-Mapped에서는 고정된 위치에 저장되므로 대체 라인 선택이 필요 없다.

2-Way Set-Associative

2-Way Set Associative Cache는 각 메모리 블록이 두 개의 캐시 라인 중 하나에 매핑될 수 있다. 즉, 특정 메모리 주소는 두 개의 캐시 라인 중 어느 하나에 저장될 수 있다. Direct-Mapped에 비해 충돌 확률이 낮아진다.

initialize_cache 함수에서 associativity 값을 2로 설정하여 2-Way Set Associative 캐시를 구현한다.

- 2-Way Set-Associative 캐시는 각 세트에 두 개의 캐시 라인이 존재하여 두 개의 캐시 라인 중 하나에 데이터를 저장할 수 있다.
- find_cache_line 함수는 주소를 통해 세트 인덱스를 계산하고, 해당 세트 내의 두 캐시 라인 중 하나에 데이터가 있는지 확인한다.
- select_victim_cache_line 함수는 교체 정책에 따라 두 개의 캐시 라인 중 하나를 선택하여 대체한다.

4-Way Set-Associative

4-Way Set Associative Cache는 각 메모리 블록이 네 개의 캐시 라인 중 하나에 매핑될 수 있다.
2-Way에 비해 충돌 확률이 더욱 낮아진다.
initialize_cache 함수에서 associativity 값을 4로 설정하여 4-Way Set Associative 캐시를 구현한다.

- 4-Way Set-Associative 캐시는 각 세트에 네 개의 캐시 라인이 존재하여 네 개의 캐시 라인 중 하나에 데이터를 저장할 수 있다.
- find_cache_line와 select_victim_cache_line 함수는 2-Way와 동일한 로직을 따르지만, 캐시 라인 수가 4개로 증가한다.

8-Way Set-Associative

8-Way Set Associative Cache는 각 메모리 블록이 여덟 개의 캐시 라인 중 하나에 매핑될 수 있다.
4-Way에 비해 충돌 확률이 더욱 낮아진다.
initialize_cache 함수에서 associativity 값을 8로 설정하여 8-Way Set Associative 캐시를 구현한다.

- 8-Way Set-Associative 캐시는 각 세트에 여덟 개의 캐시 라인이 존재하여 여덟 개의 캐시 라인 중 하나에 데이터를 저장할 수 있다.
- find_cache_line와 select_victim_cache_line 함수는 4-Way와 동일한 로직을 따르지만, 캐시 라인 수가 8개로 증가한다.

Fully Associative

Fully Associative Cache는 각 메모리 블록이 캐시의 모든 라인 중 어느 곳에나 저장될 수 있다.
가장 유연한 방식이지만, 구현이 복잡하고 비용이 많이 든다.
initialize_cache 함수에서 associativity 값을 cache_size / block_size로 설정하여 Fully Associative 캐시를 구현한다.

- Fully Associative 캐시는 모든 메모리 블록이 캐시의 모든 라인에 매핑될 수 있다.
- find_cache_line 함수는 전체 캐시에서 태그를 확인한다.
- select_victim_cache_line 함수는 전체 캐시에서 교체할 라인을 선택한다.

```
void initialize_cache(Cache *cache, uint32_t size, uint32_t associativity, uint32_t block_size) {
    cache->size = size;
    cache->associativity = associativity;
    cache->block_size = block_size;
    cache->num_sets = size / (block_size * associativity);
    cache->lines = (CacheLine *)malloc(sizeof(CacheLine) * cache->num_sets * associativity);
    cache->fifo_queue = (int *)malloc(sizeof(int) * cache->num_sets * associativity);
    cache->fifo_head = 0;
    cache->fifo_tail = 0;
    for (int i = 0; i < cache->num_sets * associativity; i++) {
        cache->lines[i].valid = 0;
        cache->lines[i].dirty = 0;
        cache->lines[i].last_access_time = 0;
        cache->lines[i].fifo_position = -1;
        cache->lines[i].sca = 0; // Initialize the second chance bit to 0
    }
}
```

캐시 접근 (Cache Access)

- access_cache 함수는 캐시 접근을 담당하며, 적중 여부를 확인하고 데이터를 읽거나 쓴다.
- 캐시 적중 시 데이터가 캐시에서 읽히거나 쓰여지며, 적중 시점이 업데이트된다.
- 캐시 미스 시 희생 라인을 선택하고, 필요시 데이터를 메모리에 쓰고, 새로운 데이터를 캐시에 로드한다.

각 캐시 매핑 방식에 따라 associativity 값을 적절히 설정하여 캐시를 초기화한다. Fully Associative 캐시의 경우, associativity 값을 cache_size / block_size로 설정하여 모든 블록이 모든 캐시 라인에 매핑될 수 있도록 한다.

3.2 Cache Replacement Policy in Program

Select Replacement Policy (0: Random, 1: FIFO, 2: LRU, 3: Second Chance Algorithm): 부분은 캐시 교체 정책을 선택하는 부분이다. 각 교체 정책을 구현하는 코드는 select_victim_cache_line 함수에 있다. 이 함수는 캐시 미스가 발생했을 때 교체할 캐시 라인을 선택하는 역할을 한다.

Random

```
case RANDOM:
    victim_index = base_index + rand() % cache->associativity;
    break;
```

- Random 정책은 캐시 세트 내의 캐시 라인 중 무작위로 하나를 선택하여 교체한다.
- rand() 함수를 사용하여 무작위로 선택한다.

FIFO (First-In-First-Out)

```
case FIFO:
    victim_index = base_index;
    for (int i = 1; i < cache->associativity; i++) {
        if (cache->lines[base_index + i].fifo_position < cache->lines[victim_index].fifo_position) {
            victim_index = base_index + i;
        }
    }
    cache->lines[victim_index].fifo_position = cycle_count;
    break;
```

- FIFO 정책은 먼저 들어온 캐시 라인을 먼저 교체한다.
- 각 캐시 라인은 fifo_position 필드를 가지고 있으며, 이 필드는 캐시 라인이 마지막으로 삽입된 시간을 나타낸다.
- 가장 오래된 fifo_position 값을 가진 캐시 라인을 선택하여 교체한다.
- 새로운 캐시 라인을 삽입할 때, 현재 사이클 수 (cycle_count)를 fifo_position에 저장하여 갱신한다.

LRU (Least Recently Used)

```
case LRU: {
    int lru_index = base_index;
    for (int i = 1; i < cache->associativity; i++) {
        if (cache->lines[base_index + i].last_access_time < cache->lines[lru_index].last_access_time) {
            lru_index = base_index + i;
        }
    }
    victim_index = lru_index;
    break;
}
```

- LRU 정책은 가장 최근에 사용되지 않은 캐시 라인을 교체한다.
- 각 캐시 라인은 last_access_time 필드를 가지고 있으며, 이 필드는 캐시 라인이 마지막으로 접근된 시간을 나타낸다.
- 가장 오래된 last_access_time 값을 가진 캐시 라인을 선택하여 교체한다.

Second Chance Algorithm (SCA)

```
case SCA: {
    int sca_index = base_index;
    while (cache->lines[sca_index].sca) {
        cache->lines[sca_index].sca = 0;
        sca_index = (sca_index + 1) % (cache->num_sets * cache->associativity);
    }
    victim_index = sca_index;
    break;
}
```

- SCA 정책은 FIFO와 비슷하지만, 두 번째 기회를 준다.
- 각 캐시 라인은 sca (Second Chance Algorithm) 비트를 가지고 있다.
- sca 비트가 설정된 캐시 라인은 한 번의 교체 기회를 얻고, sca 비트가 0으로 초기화된다.
- sca 비트가 0인 캐시 라인을 선택하여 교체한다.

3.3 Read and write cache in Program

Write-Through와 Write-Back 정책은 캐시의 데이터 쓰기 정책을 결정하는 중요한 방법이다. 이 두 가지 방법의 구현은 access_cache 함수에서 이루어진다.

- **Write-Through**: 데이터가 캐시에 쓰일 때, 해당 데이터가 즉시 메모리에도 쓰이는 방식이다. 데이터 일관성을 유지하기 쉽지만, 메모리 쓰기 작업이 자주 발생하여 성능이 떨어질 수 있다.
- **Write-Back**: 데이터가 캐시에 쓰일 때, 메모리에 즉시 쓰이지 않고 캐시 라인이 나중에 교체될 때만 메모리에 쓰이는 방식이다. 메모리 쓰기 작업을 줄여 성능이 향상될 수 있지만, 데이터 일관성을 유지하기 위해 추가적인 관리가 필요하다.

```c
void access_cache(Cache *cache, uint32_t address, uint8_t *data, int write, uint32_t cycle_count) {
    uint32_t index;
    uint32_t set_index = (address / cache->block_size) % cache->num_sets;

    if (find_cache_line(cache, address, &index)) {
        cachehit++;
        cache->lines[index].last_access_time = cycle_count;
        cache->lines[index].sca = 1; // Set the second chance bit on access
        if (write) {
            memcpy(cache->lines[index].data + (address % cache->block_size), data, sizeof(uint32_t));
            cache->lines[index].dirty = 1;
        } else {
            memcpy(data, cache->lines[index].data + (address % cache->block_size), sizeof(uint32_t));
```

...

```c
        cache->lines[victim_index].valid = 1;
        cache->lines[victim_index].dirty = 0;
        cache->lines[victim_index].tag = address / (cache->block_size * cache->num_sets);
        cache->lines[victim_index].last_access_time = cycle_count;
        memcpy(cache->lines[victim_index].data, &data_memory[address - (address % cache->block_size)], cache->block_size);
        if (write) {
            memcpy(cache->lines[victim_index].data + (address % cache->block_size), data, sizeof(uint32_t));
            cache->lines[victim_index].dirty = 1;
        } else {
            memcpy(data, cache->lines[victim_index].data + (address % cache->block_size), sizeof(uint32_t));
        }

        // Update FIFO queue for FIFO policy
        if (replacement_policy == FIFO) {
            cache->fifo_queue[cache->fifo_tail] = victim_index;
            cache->fifo_tail = (cache->fifo_tail + 1) % (cache->num_sets * cache->associativity);
        }
    }
}
```

1. **캐시 히트 시 처리**
    - find_cache_line 함수로 캐시 히트를 확인한다.
    - 히트된 경우, last_access_time을 갱신하고 sca 비트를 설정한다.
    - 쓰기 작업인 경우, 데이터를 캐시 라인에 쓰고 dirty 비트를 설정하여 Write-Back 정책을 적용한다.
2. **캐시 미스 시 처리**
    - 캐시 미스가 발생한 경우, select_victim_cache_line 함수로 대체할 캐시 라인을 선택한다.

- 선택된 캐시 라인이 유효하고 dirty 비트가 설정된 경우, Write-Back 정책에 따라 해당 캐시 라인의 데이터를 메모리에 쓴다.
- 새로운 데이터를 캐시 라인에 로드하고, 필요한 경우 Write-Through 정책에 따라 메모리에도 쓴다.
- fifo_queue를 업데이트하여 FIFO 정책을 관리한다.

위와 같이, access_cache 함수는 Write-Through와 Write-Back 정책을 구현하여 캐시 데이터의 일관성과 성능을 관리한다.

4.0 실행결과

```
const char* filename = "C:\\Users\\user\\Desktop\\Single cycle\\simple3.bin";
```

다음에 입력받을 바이너리 파일을 받은 후 실행하면

```
Select Cache Mapping Type (0: Direct-Mapped, 1: 2-Way, 2: 4-Way, 3: 8-Way, 4: Fully Associative): 0
Select Replacement Policy (0: Random, 1: FIFO, 2: LRU, 3: Second Chance Algorithm):
```

다음과 같이 Cache Mapping Type 과 Replacement Policy를 선택할 수 있다. 선택 후 실행을 하면 (Direct-Mapped, Random 기준)

```
Cycle: 8
Return Value (R[2]) : 0
Number of instructions: 8
Number of memory access instructions: 2
Number of Register ops: 5
Number of branch instruction: 0
Number of jump instruction: 1
Predict correct : 0  , mis predict : 0 ,total predict: 0
mismem cache access: 1, hitmem cache access: 1
cache Accurate : 50
cache confilct miss: 0
cold miss: 0
AMAT: 1010 ns
```

simple 결과

```
Cycle: 10
Return Value (R[2]) : 100
Number of instructions: 10
Number of memory access instructions: 4
Number of Register ops: 7
Number of branch instruction: 0
Number of jump instruction: 1
Predict correct : 0  , mis predict : 0 ,total predict: 0
mismem cache access: 1, hitmem cache access: 3
cache Accurate : 75
cache confilct miss: 0
cold miss: 0
AMAT: 1030 ns
```

simple2 결과

```
Cycle: 1330
Return Value (R[2]) : 5050
Number of instructions: 1330
Number of memory access instructions: 613
Number of Register ops: 715
Number of branch instruction: 101
Number of jump instruction: 2
Predict correct : 0  , mis predict : 0 ,total predict: 0
mismem cache access: 1, hitmem cache access: 612
cache Accurate : 99
cache confilct miss: 0
cold miss: 0
AMAT: 7120 ns
```

simple3 결과

```
Cycle: 243
Return Value (R[2]) : 55
Number of instructions: 243
Number of memory access instructions: 100
Number of Register ops: 161
Number of branch instruction: 9
Number of jump instruction: 22
Predict correct : 0  , mis predict : 0 ,total predict: 0
mismem cache access: 10, hitmem cache access: 90
cache Accurate : 90
cache confilct miss: 0
cold miss: 0
AMAT: 10900 ns
```

simple4 결과

5.0 결론

이번 프로그램을 구현하는 과정에서 가장 어려웠던 점은 캐시의 다양한 매핑 방식과 교체 정책을 효과적으로 구현하는 것이었다. 각기 다른 매핑 방식과 교체 정책이 캐시의 성능에 미치는 영향을 이해하고, 이를 코드로 정확하게 반영하는 과정에서 많은 고민과 시행착오가 필요했다. 특히, LRU와 2차 기회 알고리즘과 같은 복잡한 교체 정책을 구현하는 데 있어 적절한 자료 구조와 알고리즘을 선택하는 것이 어려웠다.

이 프로그램을 개발하면서 특히 신경 쓴 부분은 파이프라인의 데이터 및 제어 해저드 해결과 캐시 메모리의 정확한 시뮬레이션이었다. 데이터 해저드와 제어 해저드를 감지하고 이를 해결하기 위해 스톨, 포워딩, 플러싱 등의 기법을 구현하는 데 집중했다. 또한, 캐시의 히트와 미스, 교체 상황을 정확하게 반영하여 시뮬레이션의 신뢰성을 높이는 데 많은 노력을 기울였다.

이 프로그램을 개발하면서 깨닫고 배운 점은 캐시 메모리와 파이프라인의 중요성과 그 복잡성이다. 캐시 메모리는 프로세서의 성능을 극대화하는 데 중요한 역할을 하며, 다양한 매핑 방식과 교체 정책이 존재한다는 것을 이해하게 되었다. 또한, 파이프라인에서 발생하는 해저드를 적절히 해결하지 않으면 성능 저하가 발생할 수 있음을 깨달았다. 이를 통해 보다 효율적이고 성능이 높은 프로세서를 설계하기 위해서는 캐시와 파이프라인의 세부적인 동작을 깊이 이해하고, 이를 효과적으로 구현하는 것이 중요하다는 점을 배울 수 있었다.

컴퓨터 구조 강의를 수강하며 수행한 각 과제는 컴퓨터 시스템의 다양한 구성 요소와 그 상호 작용을 깊이 있게 이해하는 데 큰 도움이 되었다. 자신만의 ISA 설계, 싱글 사이클 프로세서 구현, 파이프라인 마이크로아키텍처, 그리고 캐시 설계 등을 통해 하드웨어 설계의 복잡성과 중요성을 실감할 수 있었다. 또한, 각 과제를 통해 문제 해결 능력, 논리적 사고력, 설계 능력을 배양할 수 있었으며, 이를 통해 컴퓨터 구조에 대한 실질적인 이해와 설계 역량을 크게 향상시킬 수 있었다. 이 과정을 통해 컴퓨터 공학자로서 한 단계 성장할 수 있었으며, 앞으로의 학습에 있어 큰 도움을 줄것이라고 확신한다.

[참고문헌]

[1]letskuku. "[컴퓨터구조] 캐시 메모리." Velog.
https://velog.io/@letskuku/%EC%BB%B4%ED%93%A8%ED%84%B0%EA%B5%AC%EC%A1%B0-%EC%BA%90%EC%8B%9C-%EB%A9%94%EB%AA%A8%EB%A6%AC

[2]koyo. "CS - 캐시 메모리." KOYO.KR. https://koyo.kr/post/csapp-cache-memory/

[3]Cheung, Richard. "Direct Mapped Cache." Emory University.
https://www.cs.emory.edu/~cheung/Courses/355/Syllabus/8-cache/dm.html

[4]"Set Associative Cache." ScienceDirect.
https://www.sciencedirect.com/topics/computer-science/set-associative-cache

[5]"Fully Associative Cache." ScienceDirect.
https://www.sciencedirect.com/topics/computer-science/fully-associative-cache