1. 메모리와 저장장치
   1. SRAM
      1. 하나의 Cell(1비트 저장단위) 는 6개의 트랜지스터로 구성된다.
      2. 전원이 공급되는 동안 저장된 값을 무한히 보존
      3. 외부 충격에 내성이 있다.
      4. 빠르고 가격이 비싸다
      5. cpu안에 들어가는 레지스터에 사용.
   2. DRAM
      1. 배터리의 동작원리를 빌려서 만든 램
      2. 하나의 셀은 하나의 캐패시터로 구성된다. 이걸 읽기 위한 하나의 트랜지스터가 필요하다.
      3. 10~100ms마다 refresh해야한다. –누설 전류
      4. 외부 자극에 민감하다.
      5. 주 메모리에 사용된다.
   3. Enhanced DRAM
      1. SDRAM 클럭을 이용하여 메모리 제어
      2. DDR SDRAM 클럭의 상승/하강을 모두 이용하여 메모리 제어
   4. NV MEMORY(비휘발성)
      1. ROM, EPROM, FLASH MEMORY 등
      2. 전원이 공급되지 않아도 데이터가 저장됨
      3. 새로운 값을 넣는데 큰 비용이 들어간다.
      4. 펌웨어(고정된) 프로그램 저장용
      5. DRAM 수준의 접근 속도를 제공하는 비휘발성 메모리 개발중
   5. 버스 구조
      1. 메모리는 I/O 브릿지(사우스 브릿지)에 버스로 연결되어 있다
2. 메모리 계층 구조
   1. 저장 장치의 한계
      1. 시스템 하드웨어간 큰 성능차이 (용량 / 성능 반비례 관계)
      2. 프로세스 사이클의 병목문제
      3. 해결방안 – 메모리 계층 구조
         1. 컴퓨터 프로그램의 특징 : 참조의 지역성(Locality)
         2. 지역성을 활용하여 마치 빠르고 용량이 많은 메모리가 있는 것 처럼
   2. 지역성
      1. 지역성의 원리
         1. 프로그램이 최근 사용한 것과 같은 주소 또는 근처 주소의 데이터와 명령어를 계속 사용하려는 경향이 있다.
         2. 메모리 계층구조가 효과적으로 동작하는 근거
         3. 전체 소프트웨어에 필요한 데이터 양이 많더라도 시간을 작게 쪼개어 보면 지금 현재 동작에 사용되는 메모리는 그렇게 크지 않다.
         4. 그리고 그 사용되는 메모리의 영역은 참조 지역성에 의해 예측될 수 있다.
      2. 시간 지역성
         1. 최근에 참조한 항목이 가까운 미래에 다시 참조하게 되는 특성
         2. 대표적인 사례 : 반복문
      3. 공간 지역성
         1. 일정 시간동안 근접한 주소의 항목들이 함께 참조되는 특성
         2. 대표적인 사례 Array
      4. 프로그램에서 지역성
         1. SUM PROC
            1. sum = 0;
            2. For(int i = 0 ; i < 100; ++i)

sum += arr[i];

//데이터 참조

//공간 지역성(바로 다음 메모리 참조)

//시간 지역성(같은 변수 sum을 반복적으로 참조)

* + - * 1. Return sum
        2. 명령어 참조

명령어를 순차적으로 참조 : 파이프라이닝 적용가능

순환문 내의 명령어가 반복적으로 참조

* + - 1. 2차원 배열의 경우
         1. For(int i =0 ;i < MAX; i++)

For(int j =0 ; j < MAX; j++)

Sum += a[i][j]; //열++: 좋은 지역성

Sum += a[j][i]; //행++: 나쁜 지역성

열 우선 방식 : Stride-1 방식의 참조 패턴 공간 지역성의 우수

내부 루프의 연속 참조가 대부분 캐시에서 처리되어 속도가 빠름

* + - 1. 프로그램에서 루프의 데이터 크기가 작아서 동일한 변수나 명령어를 짧은 시간안에 반복적으로 사용하는 경우, 시간 지역성이 좋아지고, 루프에서 데이터를 참조할 때 Stride – 1 패턴을 보인다면 공간 지역성이 좋아진다.
  1. 캐시
     1. 메모리 계층 구조에서 아래쪽 저장소에 대해서 위쪽 저장소가 하는 역할
        1. 아래단계의 데이터 중 일부를 더 빠른 상위계층의 저장장치가 저장하는 것
        2. 가능한 상위 계층의 메모리에 접근하여 작업이 진행 될 수 있도록 만드는 것
     2. Hit / Miss
        1. 상위 계층의 메모리(캐시)에 저장한 데이터가 호출된 경우 : HIT
        2. 캐시에 호출된 데이터가 없는 경우 : MISS
           1. CPU 에서 요청된 주소로 캐시를 검사하여 없는 것을 확인 ( 캐시 미스 )
           2. 메모리에서 해당 주소의 데이터를 읽어서 캐시로 가져옴. 이때 캐시에 빈 공간이 없으면 기존 캐시 데이터 중에서 한 개를 메모리에 먼저 저장하고 교체.
           3. 새로 가져온 데이터를 캐시로부터 읽어옴.
           4. 상위 캐시에서 데이터 Replacement

상위캐시에서 시간/ 공간 로컬리티에서 가장 먼 아이템(victim)을 새롭게 호출된 데이터와 교체

* + - * 1. 종류

콜드 미스 : 캐시가 비어있을 때

충돌 미스 :

하위 단계의 블록들은 배치 정책에 따라 캐시에 저장됨.

여러 블록이 동일한 위치에 배치됨으로써 발생하는 캐시 미스

용량 미스 : 상위 캐시가 용량이 부족하기 때문에 어쩔 수 없이 발생하는 캐시 미스

* + 1. 캐시 메모리
       1. SRAM : faster but more expensive than DRAM
       2. 구조 (S, E, B)
          1. 전체 캐시를 S(2^n)개의 SET로 구성된다.
          2. 하나의 세트를 E(2^n)개의 라인으로 구성된다.
          3. 하나의 라인은 적중 판단하는 valid/ tag와 B(2^n)개의 바이트로 구성된 캐시 블락으로 구성된다.

가상 메모리 주소 값에 대한 해시 테이블로 저장. S, E, B는 각각 근접한 주소들의 집합.

Valid :

유효성 판단. (페이지 생각하면 이해가 쉬울 듯),

실제로 해당 메모리 데이터가 캐시에 없을 수 도 있다.

주소만 있고 데이터가 없는 경우 invalid

Tag : 가상 메모리 주소 값의 일부분,

Set값 자체도 주소의 일부분이므로, 같은 Set내부에서 같은 Tag값을 갖는 라인은 없다.

* + - * 1. BLOCK은 연속된 메모리 공간에 있는 바이트들의 집합이다.

캐시의 결과는 한 블럭 전체를 가져온다. I/O작업 속도를 최소화하기 위함.

참조 지역성을 고려하면 매우 유용한 속성

* + - * 1. 캐시 동작

캐시를 불러오는 동작

주소를 tag bits, set index, block offset 나누어서 접근

Set index로 Set를 찾고 tag를 비교하면서 원하는 블록을 찾고 offset으로 정확한 블록 데이터를 가져옴

(VALID && TAG == t.bits) ? HIT : MISS

* + - 1. 캐시 구성 방법
         1. 직접 사상 (Direct Map)

세트당 라인이 1개

구현 쉽고 캐시에 저장된 데이터에 한해서 검색시간이 빠르다.

하지만 캐시 적중률이 적다. 같은 Set비트를 가진 데이터는 동시에 저장될 수 없기 때문에 풀이 적어진다.

충돌 미스이므로 효율이 떨어진다. 특별한 경우가 아니면 실제로 못쓴다.

세트 인덱스 비트의 위치 고찰

주소의 중간 비트를 인덱스로 사용하는 것의 장점

연속된 메모리 참조에 대한 캐시 충돌 미스 회피.

맨 앞 비트가 인덱스이면 인접한 메모리들이 하나의 라인에 들어갈 수 밖에 없다.

중간 비트를 인덱스로 하면 그나마 좀 멀리 떨어진 메모리들이 각각의 라인에 들어가게 된다.

* + - * 1. 연관 사상 (Set Associated Map)

세트당 복수개의 라인

충돌 미스 확률은 낮다. Set가 같아도 Tag가 다르면 따로 저장되므로

하지만 태그 검색 비용이 커진다. (두 번 검색)

미스나면 세트에서 가장 안 쓰일 것 같은 라인을 쫓아내고 새로운 애를 저장한다. (페이지 교체 정책과 유사)

* + - * 1. 완전 연관 사상

하나의 세트에 모든 라인이 포함.

충돌 미스는 없지만 검색시간 헬

참조 도메인 자체가 작은 경우 괜찮은 방법이다. (TLB)

TLB : 페이지 테이블을 캐싱해서 가상 메모리 접근 시간을 빠르게

* + - * 1. 일반적으로 사용하는 캐시는 N-Way-Associated Map 캐시
      1. 캐시의 데이터 쓰기 처리
         1. 문제 발생

캐시에 쓰기 했으면, 같은 주소의 캐시의 데이터와 메인 메모리의 데이터가 서로 다름.

데이터 일관성의 문제

* + - * 1. 데이터 쓰기 정책

Write-Through

쓰기 하면 메모리에도 즉시 저장

쓰기 버퍼를 따로 만들어 활용하여 성능 개선

Write-back

Dirty Bit 사용하여 갱신되었는지를 체크

갱신된 캐시가 교체정책에 의해 쫒겨날 때, 메모리도 캐시 내용으로 갱신

쓰기 미스의 처리

Write-allocate

캐시에 읽어와서 해당 라인을 교체

로컬리티를 생각해보면 효율적임

No-Write-allocate

메모리에 직접 저장

캐시 블록 전체를 쓰는 경우에 효율적임

Write through 는 어차피 다 새로 쓰므로 No-Write-allocate가 효율적이다.

Write-back은 교체할 때 새로 쓰므로 Write-allocate가 좋다.

* + - 1. 캐시 성능 평가 기준
         1. HIT RATE / MISS RATE

최고 중요

계층간 속도격차가 크므로

소프트웨어로 개선할 수 있는 부분

* + - * 1. HIT TIME / MISS PENALTY

캐시에서 CPU로 데이터 읽어오는 시간. 캐시 검색 포함

미스났을때 데이터 읽기에 추가되는 시간.

* + - * 1. 일반적인 지표는 MISS RATE

HIT TIME 과 MISS PENALTY의 격차가 심각하므로 MISS RATE가 1, 2 차이나는 게 실제 성능상 큰 차이가 난다.

* + - * 1. 캐시/블락 크기가 크면 miss rate는 감소 hit time 증가
      1. 추가 이슈
         1. 다중 쓰레드 프로그래밍

같은 메모리 위치를 서로 다른 CPU가 각자의 캐시에 복사본을 갖고 있는 경우

한쪽에서 WRITE해서 갱신한다면, 어떤 쓰기 정책을 취하더라도 다른 쪽 캐시의 데이터는 갱신되지 않음.

이런 문제는 시스템 아키텍처 수준에서 엄격한 동기화를 적용하는 것을 통해 해결 가능하다.

스누핑 프로토콜

상위의 컨트롤러가 사용되는 캐시의 주소를 추적, 감시

하나의 캐시가 갱신되면 같은 주소를 갖는 모든 캐시를 INVALDE처리

다시 메모리에서 갱신된 데이터를 가져옴

* + - * 1. 가상 주소 캐시 / 물리 주소 캐시

1. 캐시와 프로그램 성능
   1. 캐쉬 프랜드리 코드
      1. 좋은 지역성
      2. 내부 루프의 캐시 미스가 최소화 되도록 (암달의 법칙)
         1. Common case optimization
         2. 내부 루프가 가장 빈번하게 반복적으로 사용된다.
         3. 따라서 내부 루프의 캐시적중을 높이면, 전체 성능이 올라간다.
      3. 캐시 미스를 최소화하는 방법
         1. 반복문의 Working set을 작게 한다. (시간 지역성)
            1. 단위 시간동안 참조한 메모리의 집합. 그 시간동안 캐쉬 적중률 증가
         2. 같은 주소에 접근하는 Stride를 작게 (공간 지역성)
            1. Stride는 다음스텝에 접근하는 주소와 이전 스템의 주소의 간격
            2. 가능한 연속된 메모리를 사용