Prefetch

데이터에 접근하기 전에 데이터(블록)를 미리 가져온다.

Why? Memory latency와 Compulsory miss(miss rate)를 낮출 수 있다.

Q. Prefetch 과정에서의 misprediction이 correctness에 영향을 주는가?

A. No. 잘못 가져온 데이터는 CPU에서 애초에 사용되지 않기 때문에 정확도에 영향을 주지 않는다.

1. 어떤 데이터(주소)를 Prefetch 할 것인가

   이전의 데이터 접근 패턴을 기반으로 Prefetch algorithm이 예측한다. 필요 없는 데이터를 prefetch하면 memory bandwidth, cache space, energy consumption 등이 낭비된다.

   아래 두 지표로 성능을 평가할 수 있으며 동시에 만족시키기는 어렵다.

   Prefetch accuracy = Used prefetch / Sent prefetch

   Coverage = Used prefetch / Total reference (access)

   Timeliness = On-time prefetch / Used prefetch

   Aggressive : Coverage ↑ / Accuracy ↓ (High P)

   Conservative : Accuracy ↑ / Coverage ↓ (Low P)

2. 언제 Prefetch request를 시작할 것인가

   너무 빠르면 데이터가 사용되기 전에 Cache에서 evict 될 수 있다.

   너무 늦으면 memory latency를 낮출 수 없다.

   Aggressive : 현재 참조하고 있는 메모리부터 좀 더 멀리 떨어진 영역까지 prefetch한다. or accuracy를 낮추는 대신 coverage를 높인다.

3. Where?

   a. Prefetched data를 어디에 배치할 것인가

      i. Cache

         장점. 간단하고 별도의 buffer가 필요하지 않다.

단점. 필요한 데이터를 evict하거나 Cache pollution이 발생할 수 있다.

ii. 별도의 Prefetch buffer

장점. 필요한 데이터를 보호할 수 있고 pollution을 제거한다.

단점. 구조가 더 복잡하다.

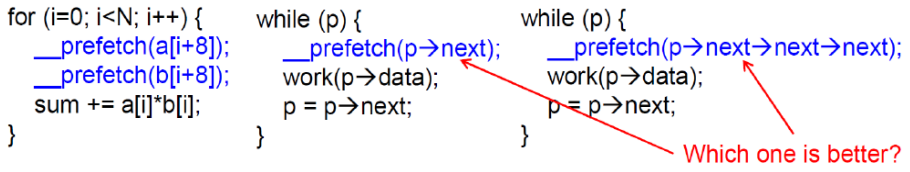
b. Cache의 어떤 level에 Prefetch할 것인가 ex. Memory → L2

c. Cache 내부 중 어느 곳에 데이터를 배치할 것인가 : Insertion policy

d. prefetch가 어떤 데이터 패턴만 확인할 것인가 ex. 모든 데이터 패턴, L1 miss 패턴만 etc.

1. 어떻게 Prefetching을 수행할 것인가

- **Software based prefetching (Compiler Prefetching)**
  - **명시적으로** 데이터를 가져오도록 **개발자가 Prefetch instruction을 코드에 추가한다.**
  - 컴파일러는 test input에 대해 코드를 실행(**Profile**)하여 자주 Cache miss가 발생하는 명령어를 파악하고, 데이터의 위치나 패턴을 분석하여 prefetch instruction 삽입 위치를 결정한다.
  - Binding (hoisting) : prefetching을 수행할 때 데이터가 레지스터에 배치(Bound)한다.
    - 장점) 추가적 prefetch instruction이 필요하지 않다.
    - 단점) Prefetch를 통해 미리 가져온 데이터가 실질적으로 사용되기 전에 다른 프로세서에 의해 수정될 경우 잘못된 값이 될 수 있다. 즉, 프로그램이 정확한가를 검토하기 위해 명시적 동기화와 통신 동작 등이 필요하다.
  - Non-binding : 데이터는 후속 동작에 의해 참조될 때까지 Bound 되지 않는다.
    - How? 레지스터가 아닌 Cache에 배치한다.
    - 장점) 후속 동작이 항상 올바른 값에 접근할 수 있어 더 큰 유연성을 제공한다.
    - 단점) 일반적인 Load와 다르게 ISA의 지원을 받아 처리된다.



- Non-faulting : 주소가 invalid하더라도 exception이 발생하지 않는다.

- **Hardware based prefetching**
  - Processor나 Cache Controller 같은 하드웨어는 **Data access의 Regularity Pattern을 모니터링**하고 prefetch address를 자동으로 생성한다.
  - 장점) 시스템 아키텍처에 맞춰 조정 가능하다. 또한 코드 구현에 따라 성능이 달라지지 않으며 instruction execution bandwidth를 사용하지 않는다.
  - 단점) 하드웨어로 복잡한 데이터 패턴을 감지하기 어렵다.

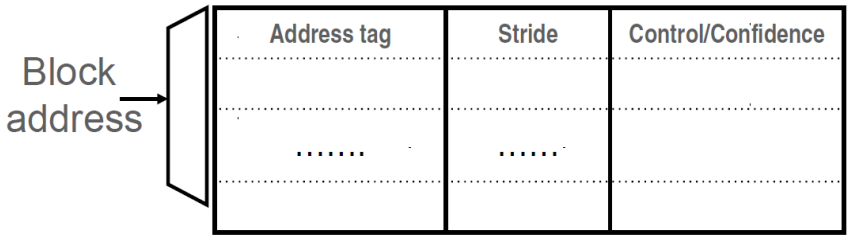
  Next-Line Prefetching (or next sequential)

  - Spatial locality를 가정하고 access 이후 N개의 cache line을 prefetch 한다.
  - 장점) 단순하고 복잡한 패턴을 감지할 필요가 없다. 연속적인 패턴에 대해 효과적이다.
  - 단점) 불규칙한 패턴의 경우 bandwidth가 낭비된다.

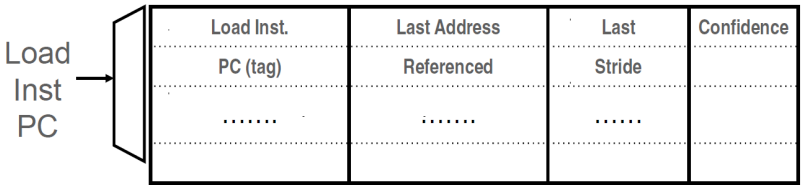
  Stride Prefetching

  - Hardware 기반의 Prefetching으로 메인 메모리에서 **일정한 간격(Stride)만큼 떨어진 데이터**를 Cache로 미리 가져온다.
    - Sequential prefetch : 주소 A에 대해 A + B 블록
    - Stride prefetch : 주소 A에 대해 A + S x B 블록
  - 배열과 같은 메모리에 연속적으로 저장된 데이터에 효과적이며 적절하며 Stride의 크기를 선택하는 것이 중요하다.
  - 장점)
  - 단점)
  - Cache Block Address-based : n개의 address stream에 대해 matching을 수행해야 하기 때문에 PC보다는 비효율적이다.
    - 장점) 여러 명령어 간의 상호작용으로 인해 발생하는 stride를 활용할 수 있다.

- 단점) 명령어보다 데이터 주소를 관리하는 것이 더 어렵다.



- PC (Program Counter) based : Prefetch table에서 PC의 partial bit(last address, tag)에 해당하는 entry를 확인하면, 얼마만큼 stride를 수행할지 기록되어 있다.
  - 아래의 그림에서 명령문(PC) A가 실행되면 Last address와의 간격을 통해 Stride 값을 예측한다.
  - Confidence : 특정 PC entry가 얼마나 참조 되었는지에 따라 값이 증가하고, threshold보다 클 때만 prefetching이 수행된다.
  - 장점) 하나의 PC entry만 확인하면 된다.
  - 단점) 동일한 instruction에 대해서만 prefetching이 수행된다.
  - 해결책) Lookahead PC

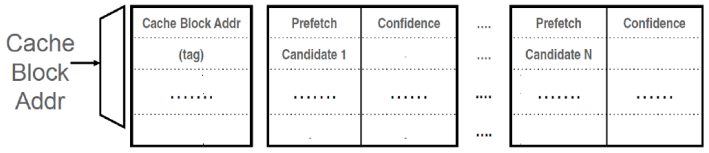


- 기타
  - Correlation-based : address 간의 상관 관계를 기억한다.

    (ex. 주소 A가 참조되면 주소 B를 X 확률로 참조)
    - Markov Prefetching : Cache block address가 주어지면 N개의 후보에 대해 confidence를 비교하여 다음 참조할 주소를 prefetch한다.

      장점) arbitrary access pattern을 처리할 수 있고, linked 구조에 효과적이다.

      단점) 모든 메모리 간의 관계(footprint)를 기억하는 것이 힘들다. (Correlation table) 간 및 bandwidth 소모가 크며 compulsory miss를 해결할 수 없다.



- Dependence-based (Content directed) : Pointer chasing (ex. p = p → next)

  (데이터로부터 다음 주소를 탐색한다.)

  - Locality 기반이 아니라 데이터의 실질적 흐름을 확인해야 하기 때문에 패턴 파악이 어렵다. 일종의 virtual memory prefetcher
- Region-based : Cache miss를 중심으로 주변 영역을 가져온다. (A-N ~ A+N)

  클래스나 구조체의 특정 멤버에 접근하는 경우 다른 멤버까지 참조되는 경우가 잦기 때문에 주변 데이터를 함께 prefetch한다.
- Locality-based : 많은 경우 access가 일정한 stride를 갖지 않기 때문에 spatial locality 원리를 기반으로 주변의 데이터를 함께 가져온다. (region-based와 유사)

  - Pentium 4 (Like) Prefetcher

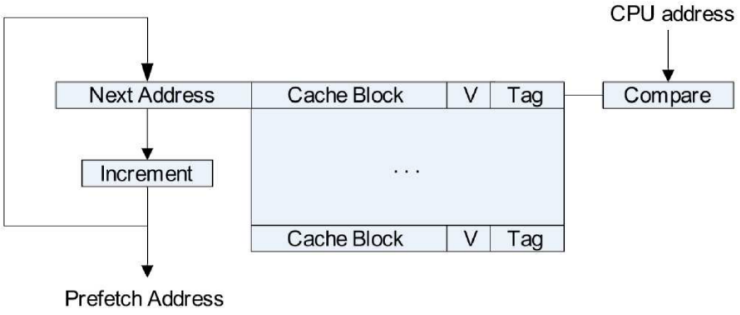
    (stream buffer처럼) 여러 개의 tracking entry를 가지며, 각 entry는 range of address를 가리킨다.

    1. 어느 tracking entry에도 포함되지 않는 L2 miss가 발생하면 tracking entry가 할당된다.
    2. 이후 첫 번째 miss로부터 +/- 16 블록 이내에서 두 번의 L2 miss가 발생하면, high spatial locality가 있다고 region을 설정한다.
    3. 이후 해당 region [A ~ P]에 대해 접근이 발생하면, [P+1 ~ P+N] region 전체에 대해 prefetch한다. → 여러 개의 window 발생
    4. [A+N ~ P+N]에 대해 모니터링을 수행한다.

    access address가 인접한 경우 뿐만이 아니라, 실제로도 효과적인 방법이다.

- Stream Buffers : 여러 stream에 대해 prefetch를 수행한다.
  - 각 entry(Buffer)는 하나의 sequential prefetched cache line(stream)을 처리한다.아래 그림은 단일 buffer에 대한 구조로,

next address가 업데이트 될 때마다 Cache Block이 prefetch 되어 buffer에 저장된다.



    - Load miss가 발생하면 모든 stream buffer의 시작 부분을 확인하여 일치하는 buffer를 탐색한다.
    - 일치하는 buffer가 존재한다면 FIFO에 따라 entry를 가져오고 cache를 업데이트한다.
    -

- Execution-based prefetching
    - 메인 프로그램에 대해 데이터를 prefetch하기 위해 thread가 실행된다.