



# Prefetching Techniques



# Reading

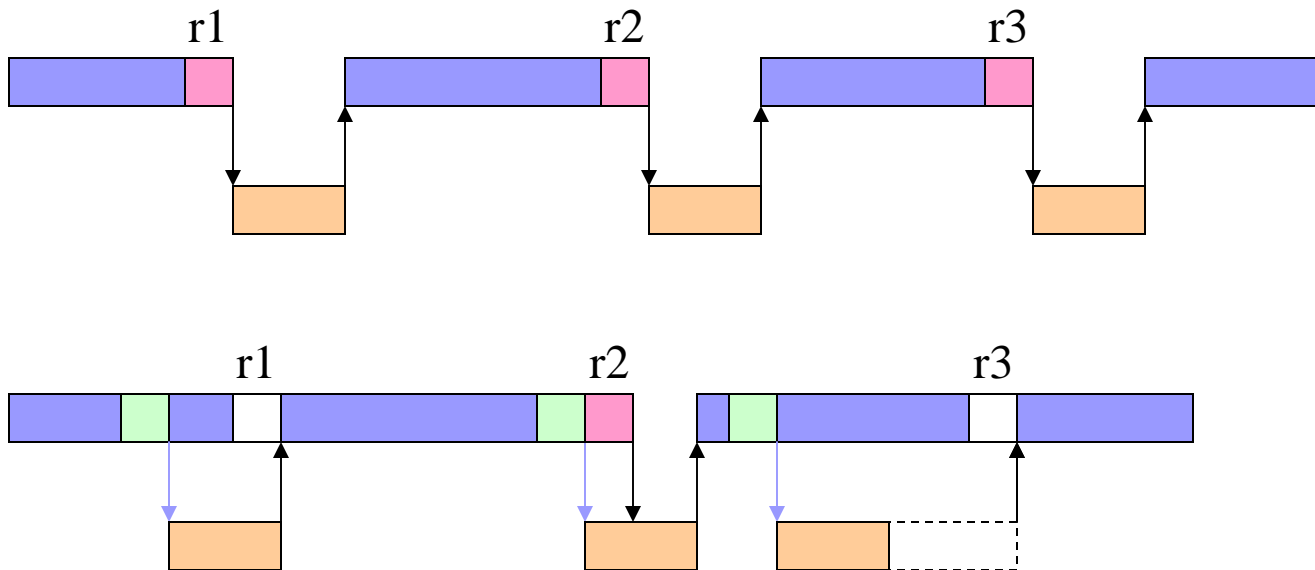
- Data prefetch mechanisms, *Steven P. Vanderwiel, David J. Lilja, ACM Computing Surveys, Vol. 32 , Issue 2 (June 2000)*



# Prefetching

- Predict future cache misses
- Issue a fetch to memory system in advance of the actual memory reference
- Hide memory access latency


# Examples





# Basic Questions

1. *When* to initiate prefetches?
  - Timely
    - Too early → replace other useful data (cache pollution) or be replaced before being used
    - Too late → cannot hide processor stall
2. *Where* to place prefetched data?
  - Cache or dedicated buffer
3. *What* to be prefetched?



# Prefetching Approaches

- Software-based
  - Explicit "fetch" instructions
  - Additional instructions executed
- Hardware-based
  - Special hardware
  - Unnecessary prefetchings (w/o compile-time information)



# Side Effects and Requirements

## ■ Side effects

- Prematurely prefetched blocks → possible "cache pollution"
- Removing processor stall cycles (increase memory request frequency); unnecessary prefetchings → higher demand on memory bandwidth

## ■ Requirements

- Timely
- Useful
- Low overhead



# Software Data Prefetching

- “fetch” instruction
  - Non-blocking memory operation
  - Cannot cause exceptions (e.g. page faults)
- Modest hardware complexity
- Challenge -- *prefetch scheduling*
  - Placement of *fetch* inst relative to the matching *load* or *store* inst
  - Hand-coded by programmer or automated by compiler





# Loop-based Prefetching

- Loops of large array calculations
  - Common in scientific codes
  - Poor cache utilization
  - Predictable array referencing patterns
- *fetch* instructions can be placed inside loop bodies s.t. current iteration prefetches data for a future iteration

# Example: Vector Product

- No prefetching

```
for (i = 0; i < N; i++) {  
    sum += a[i]*b[i];  
}
```

- Assume each cache block holds 4 elements  
→ 2 misses/4 iterations

- Simple prefetching

```
for (i = 0; i < N; i++) {  
    fetch (&a[i+1]);  
    fetch (&b[i+1]);  
    sum += a[i]*b[i];  
}
```

- Problem

- Unnecessary prefetch operations, e.g. a[1], a[2], a[3]

# Example: Vector Product (Cont.)

- Prefetching + loop unrolling

```
for (i = 0; i < N; i+=4) {  
    fetch (&a[i+4]);  
    fetch (&b[i+4]);  
    sum += a[i]*b[i];  
    sum += a[i+1]*b[i+1];  
    sum += a[i+2]*b[i+2];  
    sum += a[i+3]*b[i+3];  
}
```

- Problem

- First and last iterations

- Prefetching + software pipelining

```
fetch (&sum);  
fetch (&a[0]);  
fetch (&b[0]);  
for (i = 0; i < N-4; i+=4) {  
    fetch (&a[i+4]);  
    fetch (&b[i+4]);  
    sum += a[i]*b[i];  
    sum += a[i+1]*b[i+1];  
    sum += a[i+2]*b[i+2];  
    sum += a[i+3]*b[i+3];  
}  
for (i = N-4; i < N; i++)  
    sum = sum + a[i]*b[i];
```

# Example: Vector Product (Cont.)

- Previous assumption: prefetching 1 iteration ahead is sufficient to hide the memory latency
- When loops contain small computational bodies, it may be necessary to initiate prefetches  $\delta$  iterations before the data is referenced

$$\delta = \left\lceil \frac{l}{s} \right\rceil$$

- $\delta$ : prefetch distance,  $l$ : avg memory latency,  $s$  is the estimated cycle time of the shortest possible execution path through one loop iteration

```
fetch (&sum);
for (i = 0; i < 12; i += 4){
    fetch (&a[i]);
    fetch (&b[i]);
}

for (i = 0; i < N-12; i += 4){
    fetch(&a[i+12]);
    fetch(&b[i+12]);
    sum = sum + a[i] *b[i];
    sum = sum + a[i+1]*b[i+1];
    sum = sum + a[i+2]*b[i+2];
    sum = sum + a[i+3]*b[i+3];
}

for (i = N-12; i < N; i++)
    sum = sum + a[i]*b[i];
```



# Limitation of Software-based Prefetching

- Normally restricted to loops with array accesses
- Hard for general applications with irregular access patterns
- Processor execution overhead
- Significant code expansion
- Performed statically



# Hardware Inst. and Data Prefetching

- No need for programmer or compiler intervention
- No changes to existing executables
- Take advantage of run-time information
  
- E.g., Instruction Prefetching
  - Alpha 21064 fetches 2 blocks on a miss
  - Extra block placed in "stream buffer"
  - On miss check stream buffer
- Works with data blocks too:
  - Jouppi [1990] 1 data stream buffer got 25% misses from 4KB cache; 4 streams got 43%
  - Palacharla & Kessler [1994] for scientific programs for 8 streams got 50% to 70% of misses from 2 64KB, 4-way set associative caches
- Prefetching relies on having extra memory bandwidth that can be used without penalty

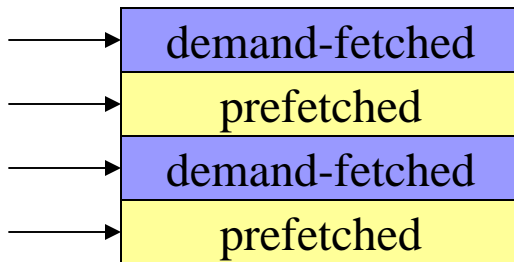


# Sequential Prefetching

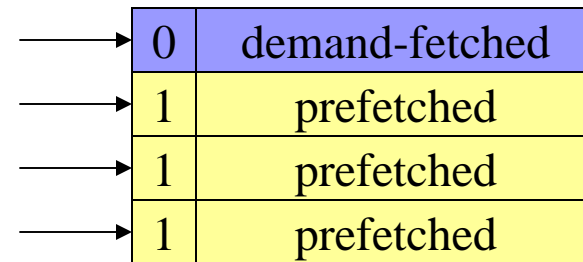
- Take advantage of spatial locality
- *One block lookahead (OBL)* approach
  - Initiate a prefetch for block  $b+1$  when block  $b$  is accessed
  - Prefetch-on-miss
    - Whenever an access for block  $b$  results in a cache miss
  - Tagged prefetch
    - Associates a tag bit with every memory block
    - When a block is demand-fetched or a prefetched block is referenced for the first time.

# OBL Approaches

## ■ Prefetch-on-miss



## ■ Tagged prefetch







# Degree of Prefetching

- OBL may not initiate prefetch far enough to avoid processor memory stall
- Prefetch  $K > 1$  subsequent blocks
  - Additional traffic and cache pollution
- Adaptive sequential prefetching
  - Vary the value of  $K$  during program execution
  - High spatial locality  $\rightarrow$  large  $K$  value
  - Prefetch efficiency metric
  - Periodically calculated
  - Ratio of useful prefetches to total prefetches



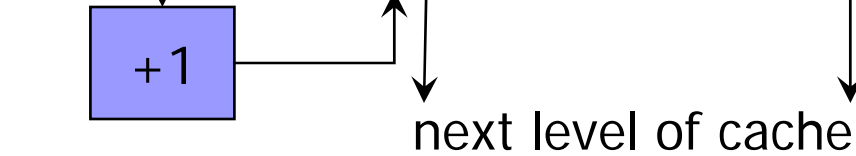
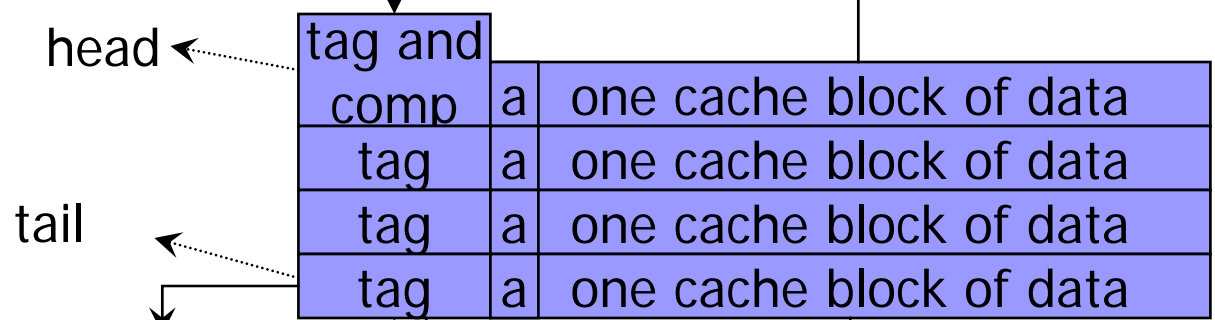
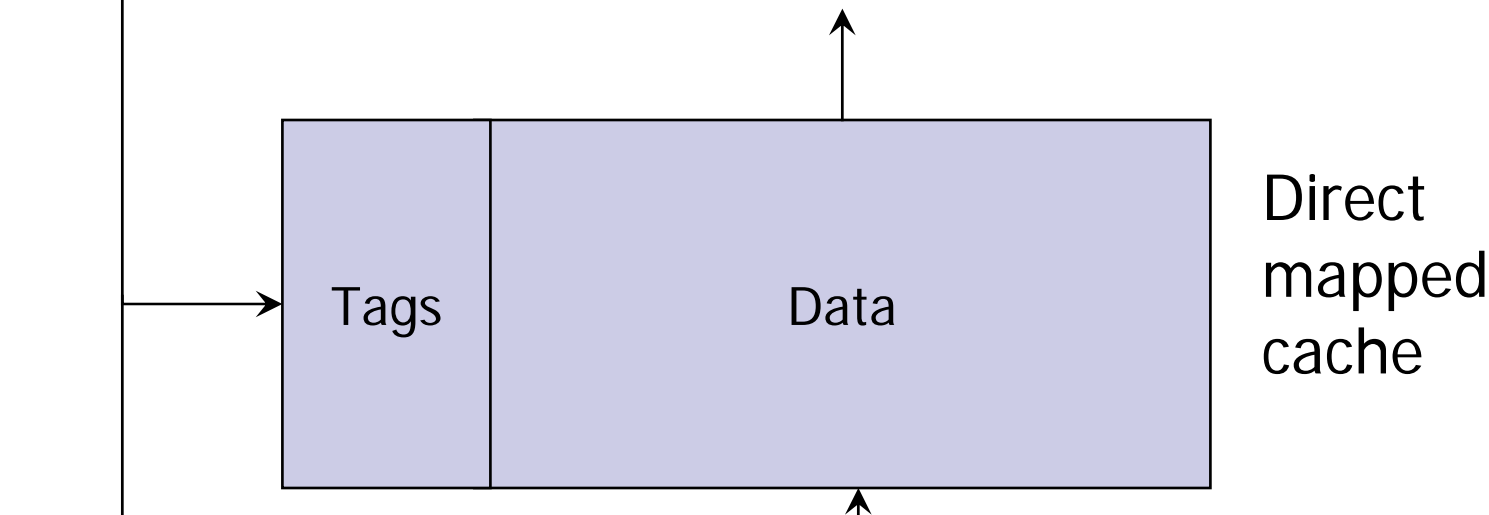
# Stream Buffer

- $K$  prefetched blocks  $\rightarrow$  FIFO stream buffer
- As each buffer entry is referenced
  - Move it to cache
  - Prefetch a new block to stream buffer
- Avoid cache pollution

# Stream Buffer Diagram

from processor

to processor



*Shown with a single stream buffer (way); multiple ways and filter may be used*

*Source: Jouppi ICS'90*



# Prefetching with Arbitrary Strides

- Employ special logic to monitor the processor's address referencing pattern
- Detect constant stride array references originating from looping structures
- Compare successive addresses used by load or store instructions

# Basic Idea

- Assume a memory instruction,  $m_i$ , references addresses  $a_1$ ,  $a_2$  and  $a_3$  during three successive loop iterations
- Prefetching for  $m_i$  will be initiated if

$$a_1 - a_2 = \Delta \neq 0$$

$\Delta$ : assumed stride of a series of array accesses

- The first prefetch address (prediction for  $a_3$ )

$$A_3 = a_2 + \Delta$$

- Prefetching continues in this way until

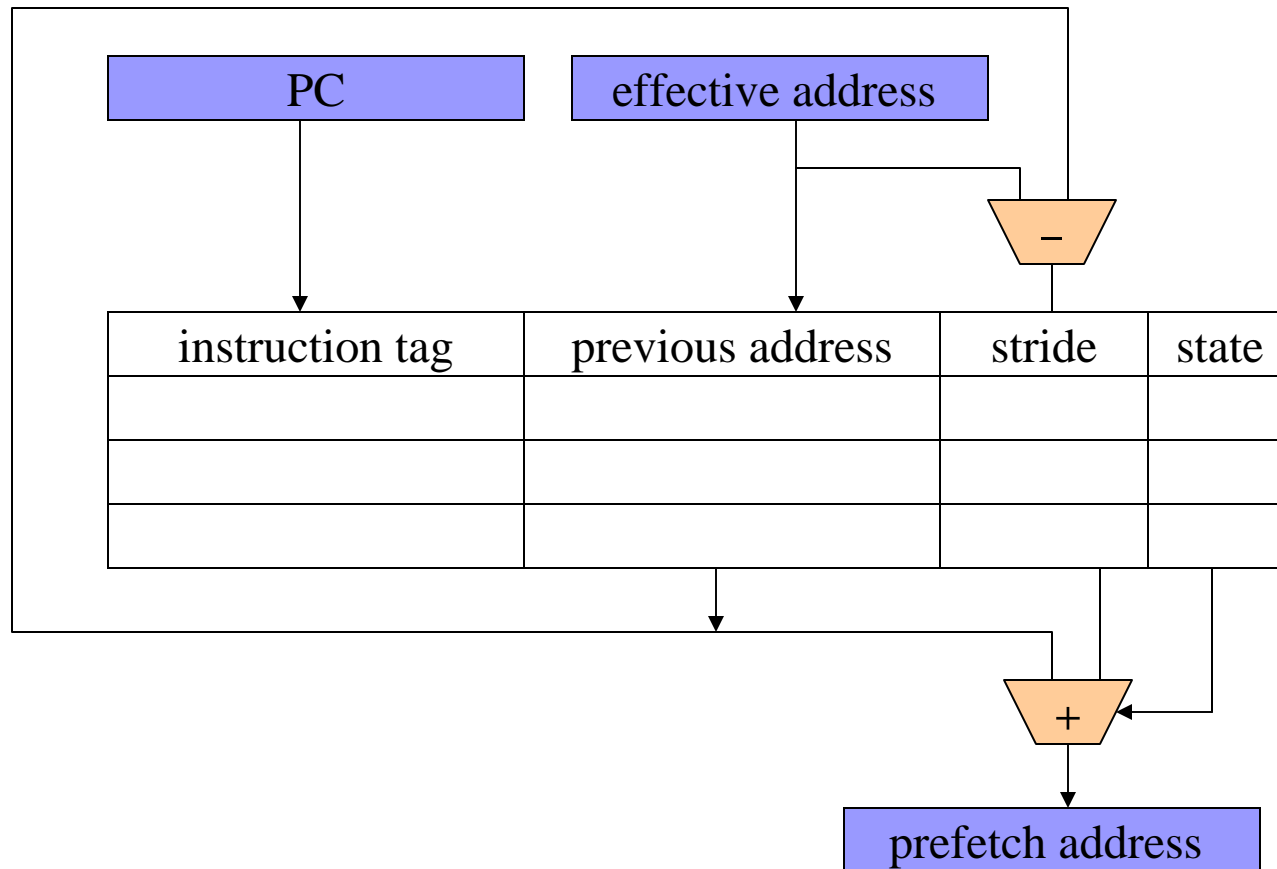
$$A_n \neq a_n$$



# Reference Prediction Table (RPT)

- Hold information for the most recently used memory instructions to predict their access pattern
  - Address of the memory instruction
  - Previous address accessed by the instruction
  - Stride value
  - State field

# Organization of RPT



# Example


```
float a[100][100], b[100][100], c[100][100];
```

```
...
```

```
    for (i = 0; i < 100; i++)  
        for (j = 0; j < 100; j++)  
            for (k = 0; k < 100; k++)  
                a[i][j] += b[i][k] * c[k][j];
```

instruction tag	previous address	stride	state
ld b[i][k]	50008	4	steady
ld c[k][j]	90800	800	steady
ld a[i][j]	10000	0	steady





# Software vs. Hardware Prefetching

## ■ Software

- Compile-time analysis, schedule fetch instructions within user program

## ■ Hardware

- Run-time analysis w/o any compiler or user support

## ■ Integration

- e.g. compiler calculates degree of prefetching ( $K$ ) for a particular reference stream and pass it on to the prefetch hardware.