
Lecture 4:

Advanced Caching Techniques (2)

Department of Electrical Engineering
Stanford University

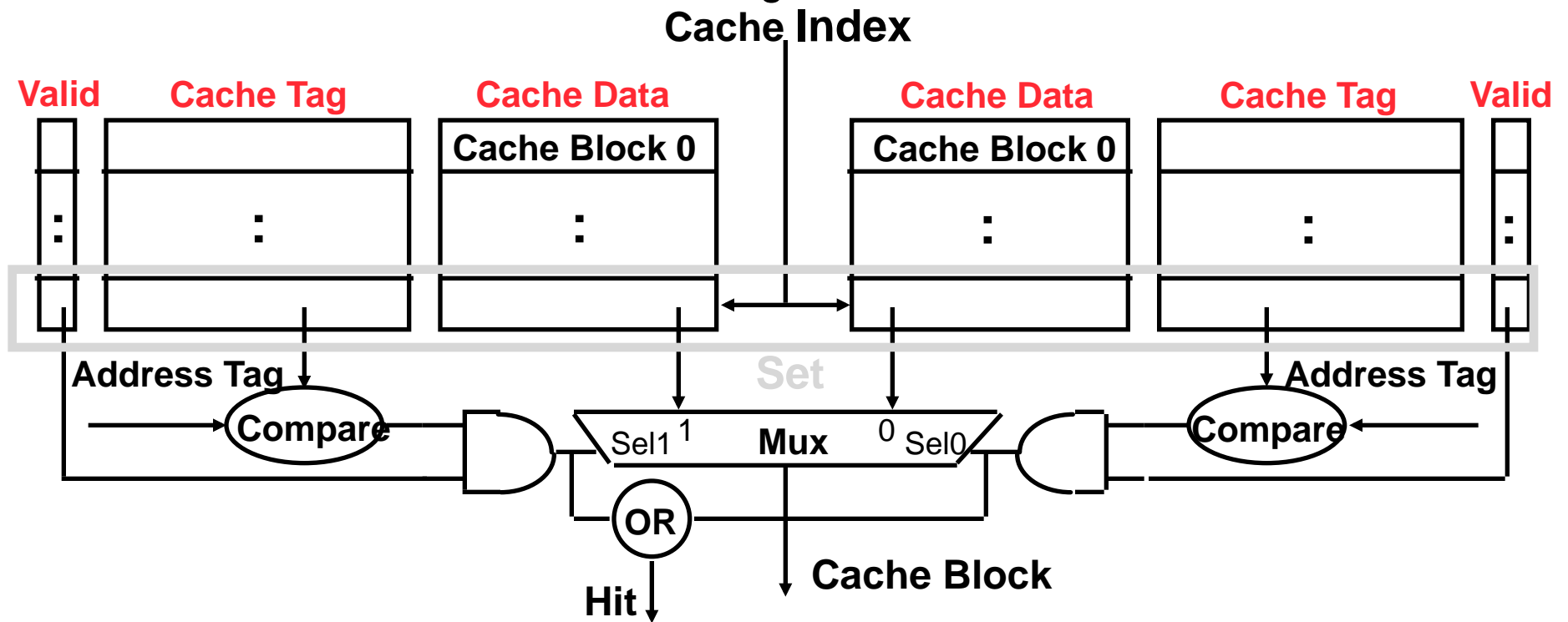
<http://eeclass.stanford.edu/ee282>

Announcements

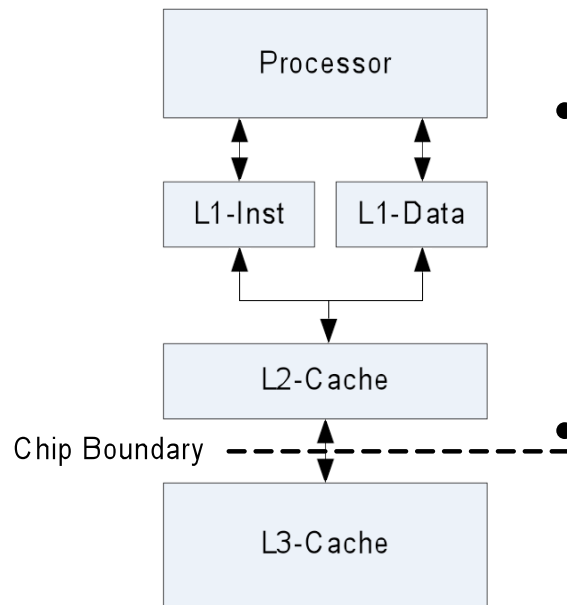
- HW1 is out (handout and online)
 - Due on 10/15 at 5pm
 - No extensions, no exceptions
 - Work in groups of 3

Review: Two-way Set Associative

- Cache index selects a “set”
- The two tags in the set are compared in parallel
- Data is selected based on the tag



Review: Multilevel Caches & Write Policies



- L2 is typically write-back
 - Because off-chip accesses are expensive

- L1 can be write-back or write-through
 - WT: pros/cons?
 - WB: pros/cons?

• Assume there is an L2 replacement, how do you guarantee inclusion if

- L1 is WT?
- L1 is WB?

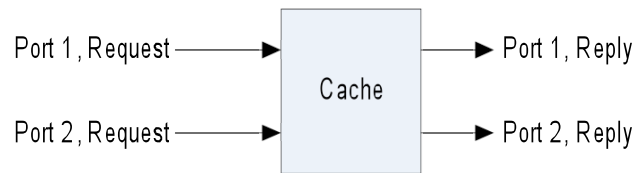
Review of Last Lecture: Advanced Caching Techniques

- Cache basics
 - 3 Cs and AMAT
 - Capacity, associativity, block size
 - Multi-level caches and inclusion
- Techniques for reducing hit time
 - Wide cache interfaces
 - Pseudo-associative caches

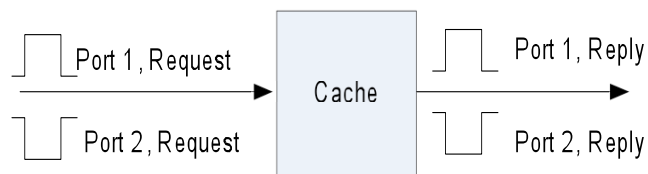
Today's Agenda

- Today's catching up portion:
 - Reducing hit time with multi-ported caches
 - Reducing miss rate: skew-associative caches and victim caches
 - Reducing miss penalty: sub-blocks, critical word first
- More on reducing miss penalty
 - Write buffers
 - Non-blocking caches
- Prefetching
- Software optimizations for memory hierarchies

Review: True Multiporting & Overclocking

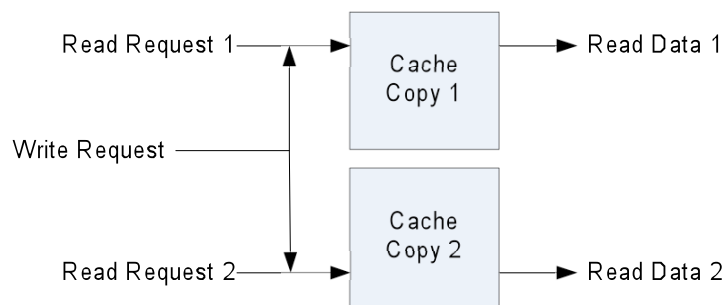


- True multiporting
 - Use 2-ported tag/data storage
 - Problem: large area increase
 - Problem: hit time increase



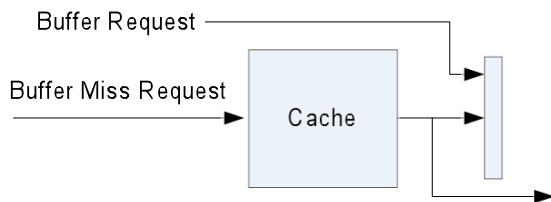
- Overclocking
 - Clock cache twice as fast as processor
 - Possible because caches are regular
 - One access per half cycle

Multiple Cache Copies & Line Buffers

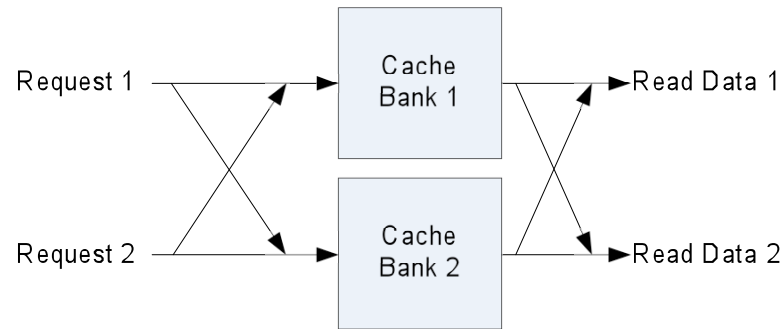


- Multiple cache copies
 - Two loads at the same time
 - Still only one store at a time
 - Twice the area, but same latency

- Line buffer or L0 cache
 - Store latest line accessed in buffer
 - Can do in parallel
 - An access to a new cache line
 - Multiple accesses that hit in buffer



Multi-banked Caches



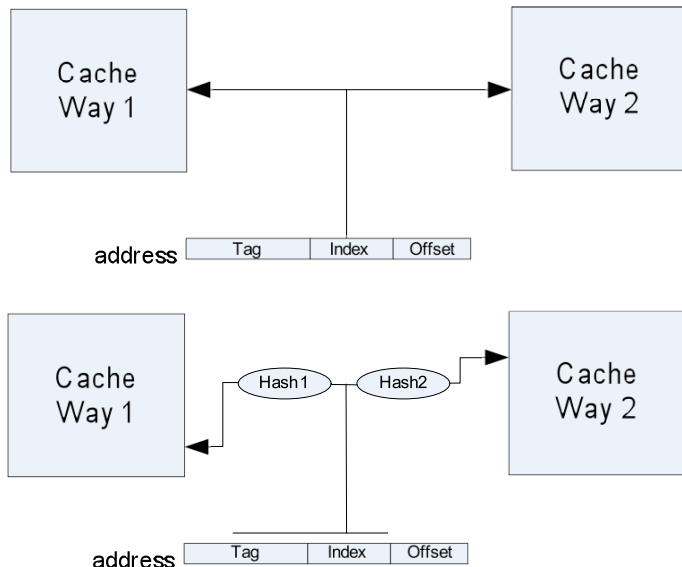
- Partition address space into multiple banks
 - Bank0 caches addresses from partition 0, bank1 from partition 1...
 - Can use least or most significant address bits for partitioning
 - What are the advantages of each approach?
- Benefits: accesses can go in parallel if no conflicts
- Problems: conflicts, distribution network, bank utilization
- Usage:
 - Multi-ported L1, low latency L2

Reducing Miss Rate

- Techniques we have seen so far
 - Larger caches
 - Reduces capacity misses
 - Higher associativity
 - Reduces conflict misses
 - Larger block sizes
 - Reduces cold misses
- Additional techniques
 - Skew associative caches
 - Victim caches

Skew Associative Caches

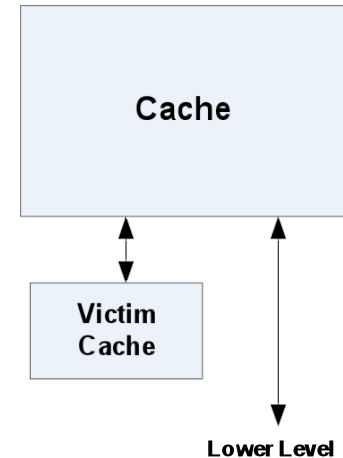
- Idea: reduce conflict misses by using different indices in each cache way
 - N-way cache: conflicts when N+1 blocks have same index bits in address



- Different indices though hashing
 - E.g. XOR index bits with some tag bits
 - E.g. reorder some index bits
- Benefit: indices are randomized
 - Less likely two blocks have same index
 - Conflict misses reduced and cache better utilized
 - May be able to reduce associativity
- Cost: latency of hash function

Victim Cache

- Small FA cache for blocks recently evicted from L1
 - Accessed on a miss in parallel or before the lower level
 - Typical size: 4 to 16 blocks (fast)
- Benefits
 - Captures common conflicts due to low associativity or ineffective replacement policy
 - Avoids lower level access
- Notes
 - Helps the most with small or low-associativity caches
 - Helps more with large blocks



Reducing Miss Penalty

- Techniques we have seen so far
 - Multi-level caches
- Additional techniques
 - Sub-blocks
 - Critical word first
 - Write buffers
 - Non-blocking caches

Sub-blocks

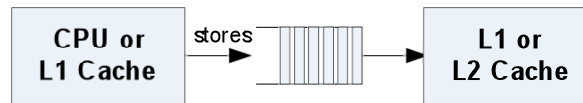
V	Subblock0	V	Subblock1	V	Subblock2	V	Subblock3
----------	------------------	----------	------------------	----------	------------------	----------	------------------

- Idea: break cache line into sub-blocks with separate valid bits
 - But they still share a single tag
- Low miss latency for loads:
 - Fetch required subblock only
- Low latency for stores:
 - Do not fetch the cache line on the miss
 - Write only the sub-block produced, the rest are invalid
 - If there is temporal locality in writes, this can save many refills

Critical Word First

- Idea: fetch requested word or subblock first
 - And then the rest of the cache block
 - Useful when blocks are large and bandwidth low
 - Not that useful if program has spatial locality
- Why critical word first works: early CPU or L1 restart:
 - Return data to CPU/L1 as soon as requested word/subblock arrives
 - Don't wait for the whole block to arrive in L1 cache

Write Buffers



- Write buffers allow for a large number of optimizations
- Write through caches
 - Stores don't have to wait for lower level latency
 - Stall store only when buffer is full
- Write back caches
 - Fetch new block before writing back evicted block
- CPUs and caches in general
 - Allow younger loads to bypass older stores
 - Beware of dependencies...

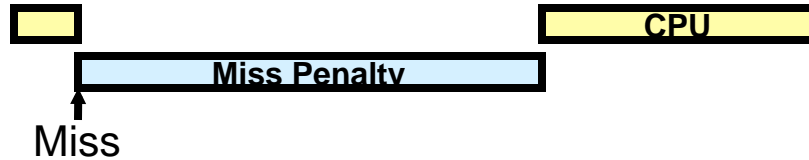
Write Buffer Design

- Size: 2-8 entries are typically sufficient for caches
 - But an entry may store a whole cache line
 - Make sure the write buffer can handle the typical store bursts...
 - Analyze your common programs, consider bandwidth to lower level
- Coalescing write buffers
 - Merge adjacent writes into single entry
 - Especially useful for write-through caches
- Dependency checks
 - Comparators that check load address against pending stores
 - If match there is a dependency so load must stall
 - Optimization: load forwarding
 - If match and store has its data, forward data to load...

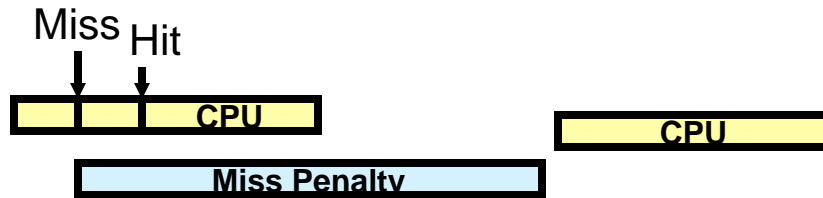
Non-blocking or Lockup Free Caches

- Idea:
 - Allow for hits while serving a miss (hit-under-miss)
 - Allow for more than one outstanding miss (miss-under-miss)
- When does it make sense (for L1, L2, ...)
 - When the processor can handle >1 pending load/store
 - This is the case with superscalar processors
 - When the cache serves >1 processor or other cache
 - When the lower level allows for multiple pending accesses
 - Multi-banked, split transaction busses, pipelining, ...
- What is difficult about non-blocking caches:
 - Handling multiple misses at the time
 - Handling loads to pending misses
 - Handling stores to pending misses

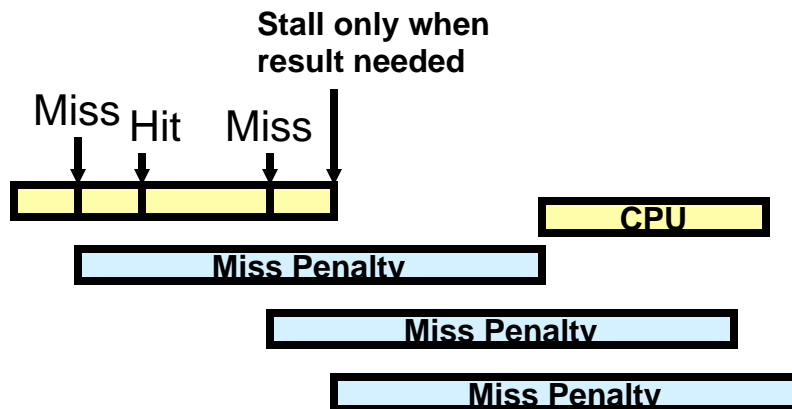
Potential of Non-blocking Caches



Stall CPU on miss



Hit under miss

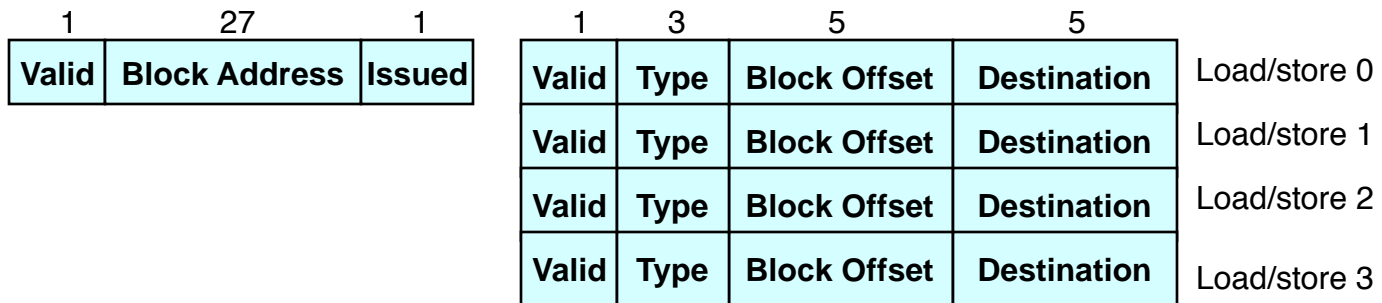


Multiple out-standing misses

Miss Status Handling Register

- Keeps track of
 - Outstanding cache misses
 - Pending load & stores that refer to that cache block
- Fields of an MSHR
 - Valid bit
 - Cache block address
 - Must support associative search
 - Issued bit (1 if already request issued to memory)
 - For each pending load or store
 - Valid bit
 - Type (load/store) and format (byte/halfword/...)
 - Block offset
 - Destination register for load OR store buffer entry for stores

MSHR



Non-block Caches: Operation

- On a cache miss:
 - Search MSHRs for pending access to same cache block
 - If yes, just allocate new load/store entry
 - (if no) Allocate free MSHR
 - Update block address and first load/store entry
 - If no MSHR or load/store entry free, stall
- When one word/sub-block for cache line become available
 - Check which load/stores are waiting for it
 - Forward data to LSU
 - Mark loads/store as invalid
 - Write word in the cache
- When last word for cache line is available
 - Mark MSHR as invalid

Prefetching

- Idea: fetch data into the cache before processors request them
 - Can address cold misses
 - Can be done by the programmer, compiler, or hardware
- Characteristics of ideal prefetching
 - You only prefetch data that are truly needed
 - Avoid bandwidth waste
 - You issue prefetch requests early enough
 - To hide the memory latency
 - You don't issue prefetch requests too early
 - To avoid cache pollution

Software Prefetching

```
for (i=0; i<N; i++) {  
    __prefetch(a[i+8]);  
    __prefetch(b[i+8]);  
    sum += a[i]*b[i];  
}
```

- Two types
 - Binding prefetching
 - Load to registers
 - Takes up registers
 - VM exceptions?
 - Non-binding prefetching
 - Loads to cache
 - Requires ISA support
 - Prefetch loads

- Issues software prefetching
 - Takes up issue slots
 - Not big issue with superscalar
 - Takes up system bandwidth
 - Must have non-blocking caches
 - Prefetch distance depends on specific system implementation
 - Non-portable code
 - Not easy to issue for pointer based structures

Hardware Prefetching

- Same goal with software prefetching but initiated by hardware
 - Can tune to specific system implementation
 - Does not waste instruction issue bandwidth
 - More portable code
- Major design questions
 - Where to place a prefetch engine?
 - L1, L2, ...
 - What to prefetch?
 - Next sequential cache line(s), strided patterns, pointers, ...
 - When to prefetch?
 - On a load, on a miss, when other prefetched data used, ...
 - Where to place prefetched data
 - In the cache or in a special prefetch buffer

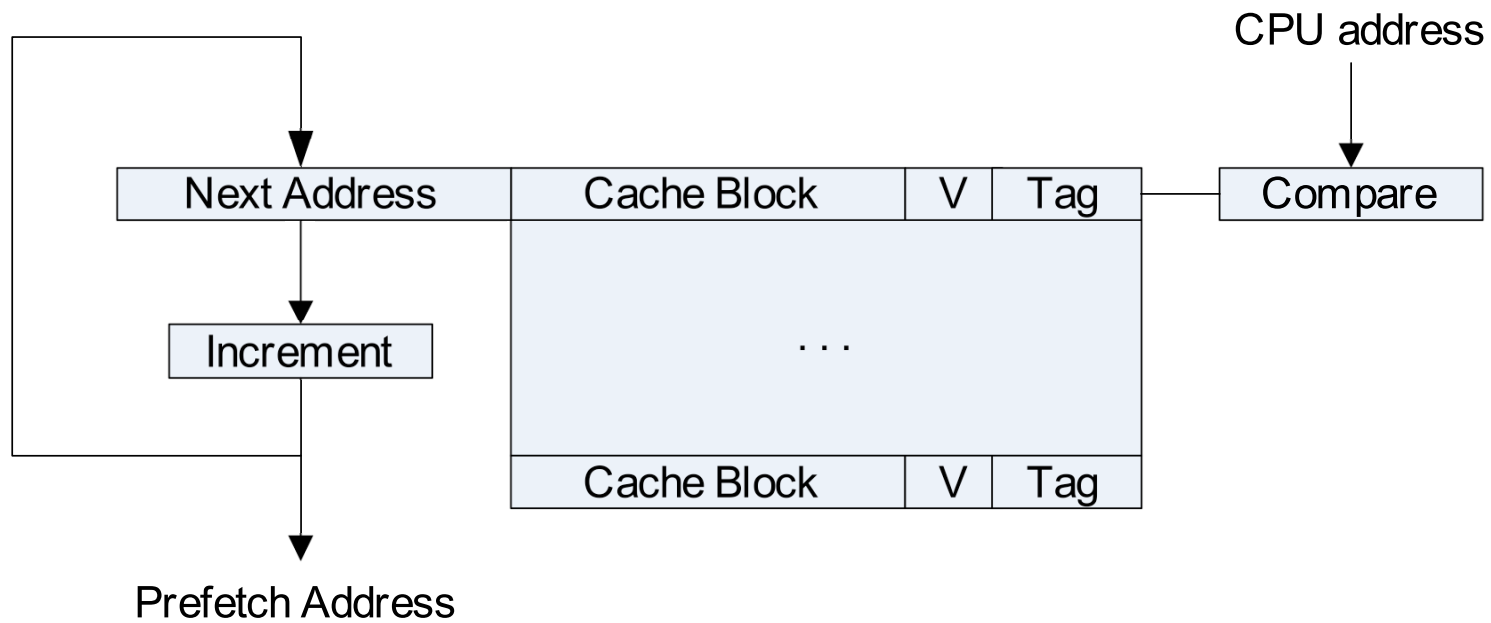
Simple Sequential Prefetching

- On a cache miss, fetch two sequential memory blocks
 - Exploits spatial locality in both instructions & data
 - Exploits high bandwidth for sequential accesses
- Operation if extra block fetched in special prefetch buffer
 - On misses, search in the prefetch buffer first
 - If block found there, move to cache
 - Prefetch buffer can be organized similarly to a victim cache
- Extend to fetching N sequential memory blocks
 - Pick N large enough to hide the memory latency

Stream Prefetching or Stream Buffers

- Sequential prefetching problem:
 - Performance slows down once every N cache lines
- Stream prefetching is a continuous version of sequential prefetching
 - Stream buffer can fit N cache lines
 - On a miss, start fetching N sequential cache lines
 - On a stream buffer hit:
 - Move cache line to cache, start fetching line (N+1)
- In other words, stream buffer tries to stay N cache lines ahead
- Design issues
 - When is a stream buffer released
 - When we miss both in the cache and the stream buffer
 - Can use multiple stream buffers to capture multiple streams
 - E.g. a program operating on 2 arrays

Stream Buffer Design



Strided Prefetching

- Idea: detect and prefetch strided accesses
 - for (i=0; i<N; i++) $A[i+1024]$
- Stride detected using a PC-based table
 - For each PC, remember the stride
 - Stride detection
 - Remember the last address used for this PC
 - Compare to currently used address for this PC
 - Track confidence using a two bit saturating counter
 - Increment when stride correct, decrement when incorrect
- How to use the PC-based table
 - When stream prefetching is initialized, direct to fetch strided
 - Everything else remains the same

PC	Stride	Last Addr	Conf
0x08ab0	8	0xf024	10
0x03fa8	1024	0xf0ab2	11

Other Ideas in Prefetching

- Prefetch engines for pointer-based data structures
 - Predict if fetched data contain a pointer & follow it
 - Works for linked-lists, graphs, etc
 - Must be very careful:
 - What is a pointer?
 - How far to prefetch?
- Correlating prefetchers
 - Learn about address correlation (ABC always accessed in order)
 - When A is accessed, immediately fetch B & C
 - Can use a PC-based table or a Markov prefetcher
- Pre-execution or run-ahead
 - Distill the part of the program that generates addresses
 - Run this program on other processor/thread to generate prefetches