15-740/18-740 Computer Architecture Lecture 13: More Caching

Prof. Onur Mutlu
Carnegie Mellon University

Announcements

- Project Milestone I
  - Due Monday, October 18

- Paper Reviews
  - Jouppi, "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers," ISCA 1990.
  - Qureshi et al., "A Case for MLP-Aware Cache Replacement," ISCA 2006.
  - Due Friday October 22

Last Time

- Handling writes
- Instruction vs. data
- Cache replacement policies
- Cache performance
  - Cache size, associativity, block size
- Enhancements to improve cache performance
  - Critical-word first, subblocking
  - Replacement policy
  - Hybrid replacement policies
  - Cache miss classification
  - Victim caches
  - Hashing
  - Pseudo-associativity

Today

- More enhancements to improve cache performance
- Enabling multiple concurrent accesses
- Enabling high bandwidth caches

- Prefetching

Cache Readings

- Required:
  - Hennessy and Patterson, Appendix C.1-C.3
  - Jouppi, "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers," ISCA 1990.
  - Qureshi et al., "A Case for MLP-Aware Cache Replacement," ISCA 2006.

- Recommended:
  - Seznec, "A Case for Two-way Skewed Associative Caches," ISCA 1993.
  - Chilimbi et al., "Cache-conscious Structure Layout," PLDI 1999.
  - Chilimbi et al., "Cache-conscious Structure Definition," PLDI 1999.

Improving Cache "Performance"

- Reducing miss rate
  - Caveat: reducing miss rate can reduce performance if more costly-to-refetch blocks are evicted
- Reducing miss latency
- Reducing hit latency

Improving Basic Cache Performance

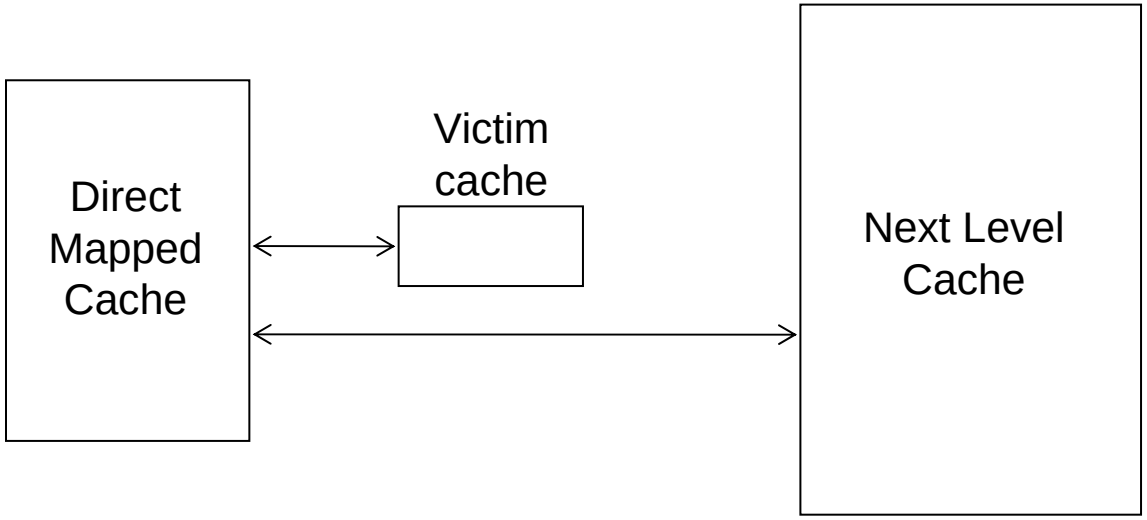
- Reducing miss rate
  - More associativity
  - Alternatives/enhancements to associativity
    - Victim caches, hashing, pseudo-associativity, skewed associativity
  - Software approaches

- Reducing miss latency/cost
  - Multi-level caches
  - Critical word first
  - Subblocking
  - Non-blocking caches
  - Multiple accesses per cycle
  - Software approaches

How to Reduce Each Miss Type

- Compulsory
  - Caching cannot help
  - Prefetching
- Conflict
  - More associativity
  - Other ways to get more associativity without making the cache associative
    - Victim cache
    - Hashing
    - Software hints?
- Capacity
  - Utilize cache space better: keep blocks that will be referenced
  - Software management: divide working set such that each "phase" fits in cache

Victim Cache: Reducing Conflict Misses



- Jouppi, "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers," ISCA 1990.
- Idea: Use a small fully associative buffer (victim cache) to store evicted blocks
  - + Can avoid ping ponging of cache blocks mapped to the same set (if two cache blocks continuously accessed in nearby time conflict with each other)
  - -- Increases miss latency if accessed serially with L2

Hashing and Pseudo-Associativity

- Hashing: Better "randomizing" index functions

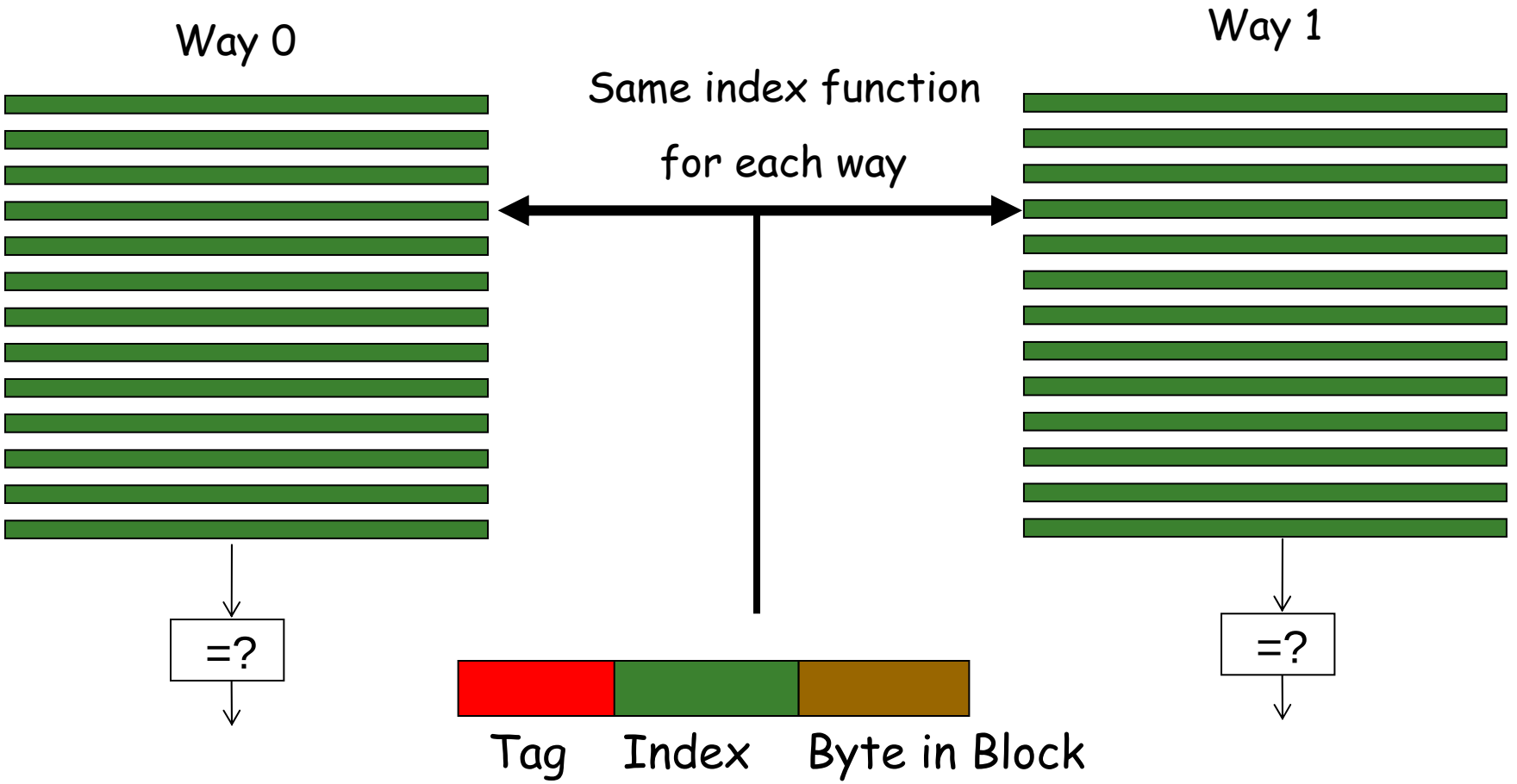
  + can reduce conflict misses
    - by distributing the accessed memory blocks more evenly to sets
  - Example: stride where stride value equals cache size

  -- More complex to implement: can lengthen critical path

- Pseudo-associativity (Poor Man's associative cache)
  - Serial lookup: On a miss, use a different index function and access cache again
  - Given a direct-mapped array with K cache blocks
    - Implement K/N sets
    - Given address Addr, sequentially look up: {0,Addr[lg(K/N)-1: 0]}, {1,Addr[lg(K/N)-1: 0]}, … , {N-1,Addr[lg(K/N)-1: 0]}

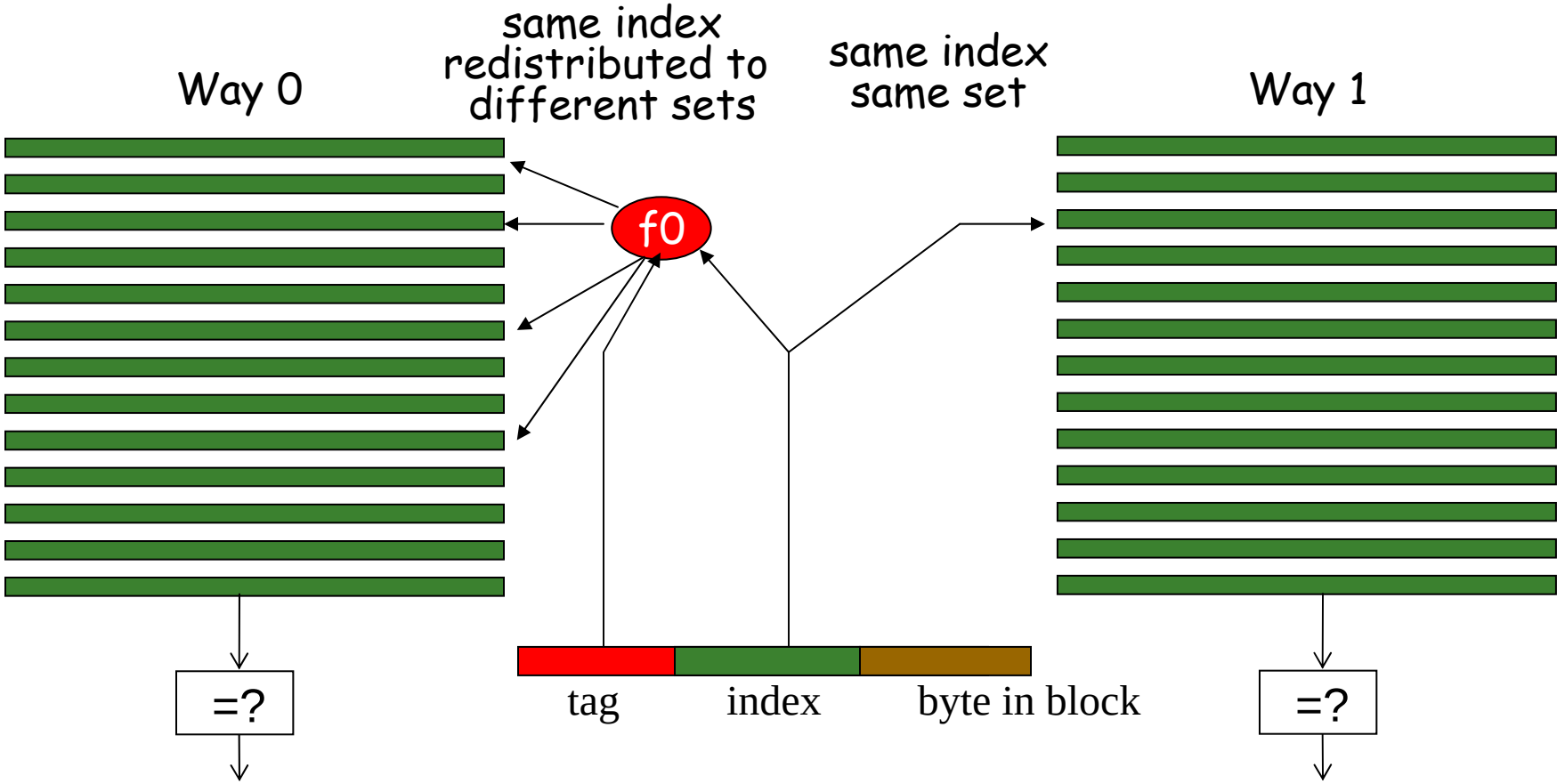
Skewed Associative Caches (I)

- Basic 2-way associative cache structure



Skewed Associative Caches (II)

- Skewed associative caches
  - Each bank has a different index function



Skewed Associative Caches (III)

- Idea: Reduce conflict misses by using different index functions for each cache way

- Benefit: indices are randomized
  - Less likely two blocks have same index
    - Reduced conflict misses
  - May be able to reduce associativity

- Cost: additional latency of hash function

Improving Hit Rate via Software (I)

- Restructuring data layout
- Example: If column-major
  - x[i+1,j] follows x[i,j] in memory
  - x[i,j+1] is far away from x[i,j]

Poor code

```
for i = 1, rows
    for j = 1, columns
        sum = sum + x[i,j]
```

Better code

```
for j = 1, columns
    for i = 1, rows
        sum = sum + x[i,j]
```

- This is called loop interchange
- Other optimizations can also increase hit rate
  - Loop fusion, array merging, ...
- What if multiple arrays? Unknown array size at compile time?

More on Data Structure Layout

```
struct Node {
    struct Node* node;
    int key;
    char [256] name;
    char [256] school;
}

while (node) {
    if (node→key == input-key) {
        // access other fields of node
    }
    node = node→next;
}
```

- Pointer based traversal (e.g., of a linked list)
- Assume a huge linked list (1M nodes) and unique keys
- Why does the code on the left have poor cache hit rate?
  - "Other fields" occupy most of the cache line even though rarely accessed!

How Do We Make This Cache-Friendly?

```
struct Node {
    struct Node* node;
    int key;
    struct Node-data* node-data;
}

struct Node-data {
    char [256] name;
    char [256] school;
}

while (node) {
    if (node→key == input-key) {
        // access node→node-data
    }
    node = node→next;
}
```

- Idea: separate frequently-used fields of a data structure and pack them into a separate data structure

- Who should do this?
  - Programmer
  - Compiler
    - Profiling vs. dynamic
  - Hardware?
  - Who can determine what is frequently used?

Improving Hit Rate via Software (II)

- Blocking
  - Divide loops operating on arrays into computation chunks so that each chunk can hold its data in the cache
  - Avoids cache conflicts between different chunks of computation
  - Essentially: Divide the working set so that each piece fits in the cache

- But, there are still self-conflicts in a block
  1. there can be conflicts among different arrays
  2. array sizes may be unknown at compile/programming time

Improving Basic Cache Performance

- Reducing miss rate
  - More associativity
  - Alternatives to associativity
    - Victim caches, hashing, pseudo-associativity, skewed associativity
  - Software approaches

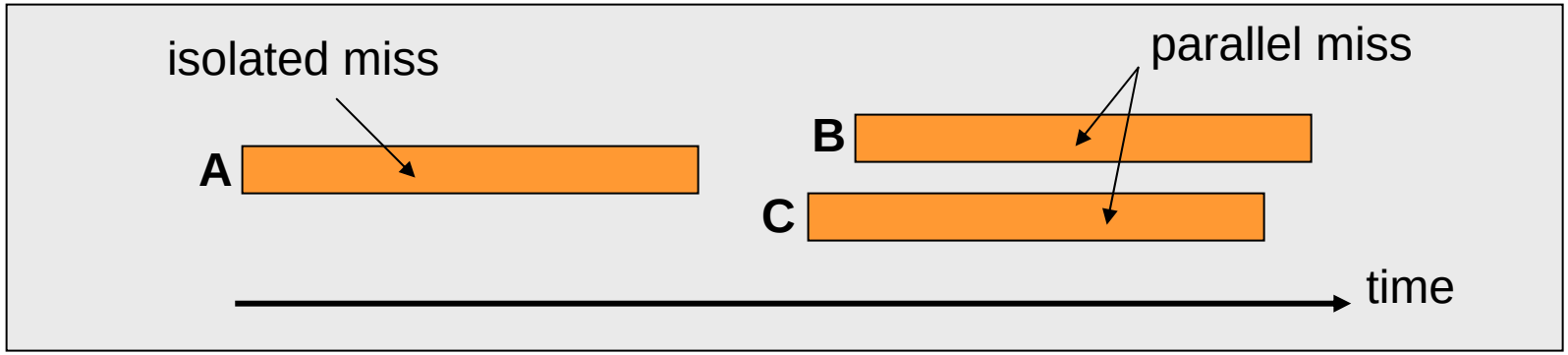
- Reducing miss latency/cost
  - Multi-level caches
  - Critical word first
  - Subblocking
  - Multiple outstanding accesses (Non-blocking caches)
  - Multiple accesses per cycle
  - Software approaches

Handling Multiple Outstanding Accesses

- Non-blocking or lockup-free caches
  - Kroft, "Lockup-Free Instruction Fetch/Prefetch Cache Organization," ISCA 1981.
- Question: If the processor can generate multiple cache accesses, can the later accesses be handled while a previous miss is outstanding?
- Idea: Keep track of the status/data of misses that are being handled in Miss Status Handling Registers (MSHRs)
  - A cache access checks MSHRs to see if a miss to the same block is already *pending.*
    - If pending, a new request is not generated
    - If pending and the needed data available, data forwarded to later load
  - Requires buffering of outstanding miss requests

Non-Blocking Caches (and MLP)

- Enable cache access when there is a pending miss
- Enable multiple misses in parallel
  - Memory-level parallelism (MLP)
    - generating and servicing multiple memory accesses in parallel
  - Why generate multiple misses?

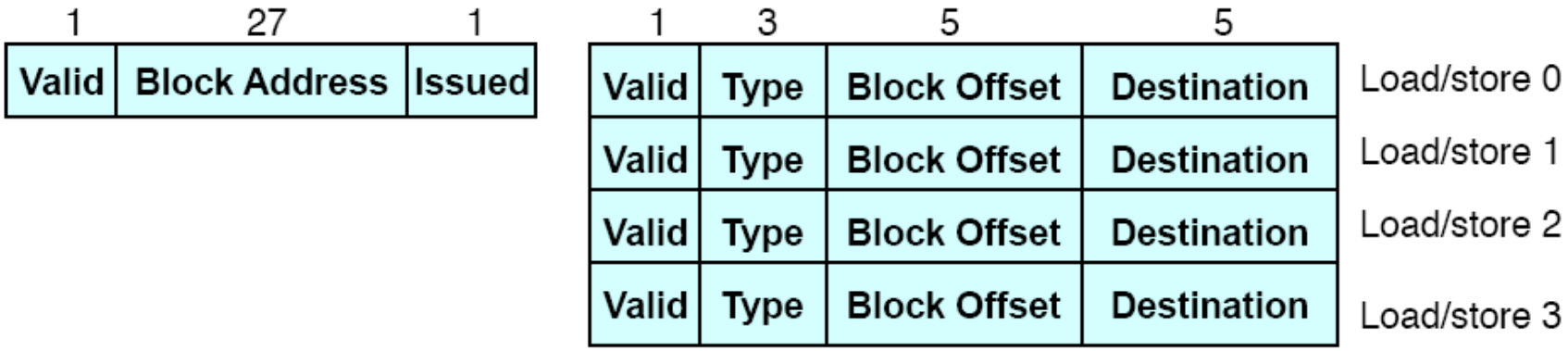


    - Enables latency tolerance: overlaps latency of different misses
  - How to generate multiple misses?
    - Out-of-order execution, multithreading, runahead, prefetching

Miss Status Handling Register

- Also called “miss buffer”
- Keeps track of
  - Outstanding cache misses
  - Pending load/store accesses that refer to the missing cache block
- Fields of a single MSHR
  - Valid bit
  - Cache block address (to match incoming accesses)
  - Control/status bits (prefetch, issued to memory, which subblocks have arrived, etc)
  - Data for each subblock
  - For each pending load/store
    - Valid, type, data size, byte in block, destination register or store buffer entry address

Miss Status Handling Register



MSHR Operation

- On a cache miss:
  - Search MSHR for a pending access to the same block
    - Found: Allocate a load/store entry in the same MSHR entry
    - Not found: Allocate a new MSHR
    - No free entry: stall

- When a subblock returns from the next level in memory
  - Check which loads/stores waiting for it
    - Forward data to the load/store unit
    - Deallocate load/store entry in the MSHR entry
  - Write subblock in cache or MSHR
  - If last subblock, dellaocate MSHR (after writing the block in cache)

Non-Blocking Cache Implementation

- When to access the MSHRs?
  - In parallel with the cache?
  - After cache access is complete?

- MSHRs need not be on the critical path of hit requests
  - Which one below is the common case?
    - Cache miss, MSHR hit
    - Cache hit

Improving Basic Cache Performance

- Reducing miss rate
  - More associativity
  - Alternatives/enhancements to associativity
    - Victim caches, hashing, pseudo-associativity, skewed associativity
  - Software approaches

- Reducing miss latency/cost
  - Multi-level caches
  - Critical word first
  - Subblocking
  - Non-blocking caches
  - Multiple accesses per cycle
  - Software approaches

Reducing Miss Cost/Latency via Software

- Enabling more memory-level parallelism
  - Restructuring code
  - Taking advantage of stall-on-use policy in hardware

- Inserting prefetch instructions

Enabling High Bandwidth Caches

Multiple Instructions per Cycle

- Can generate multiple cache accesses per cycle
- How do we ensure the cache can handle multiple accesses in the same clock cycle?

- Solutions:
  - true multi-porting
  - virtual multi-porting (time sharing a port)
  - multiple cache copies
  - banking (interleaving)

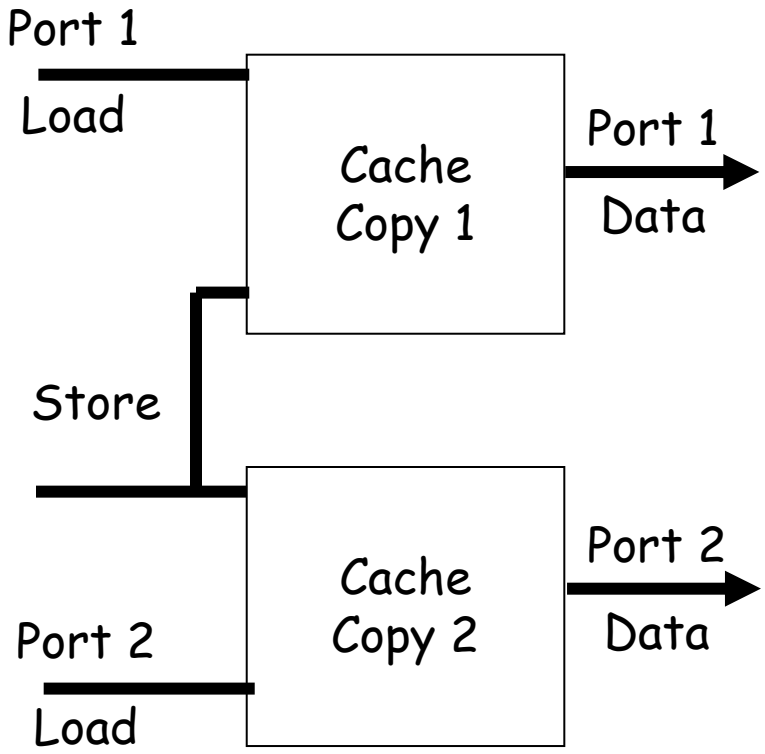
Handling Multiple Accesses per Cycle (I)

- True multiporting
  - Each memory cell has multiple read or write ports
  - + Truly concurrent accesses (no conflicts regardless of address)
  - -- Expensive in terms of area, power, and delay
  - What about read and write to the same location at the same time?
    - Peripheral logic needs to handle this

- Virtual multiporting
  - Time-share a single port
  - Each access needs to be (significantly) shorter than clock cycle
  - Used in Alpha 21264
  - Is this scalable?

Handling Multiple Accesses per Cycle (II)

- Multiple cache copies
  - Stores update both caches
  - Loads proceed in parallel
- Used in Alpha 21164
- Scalability?
  - Store operations form a bottleneck
  - Area proportional to “ports”



Handling Multiple Accesses per Cycle (III)

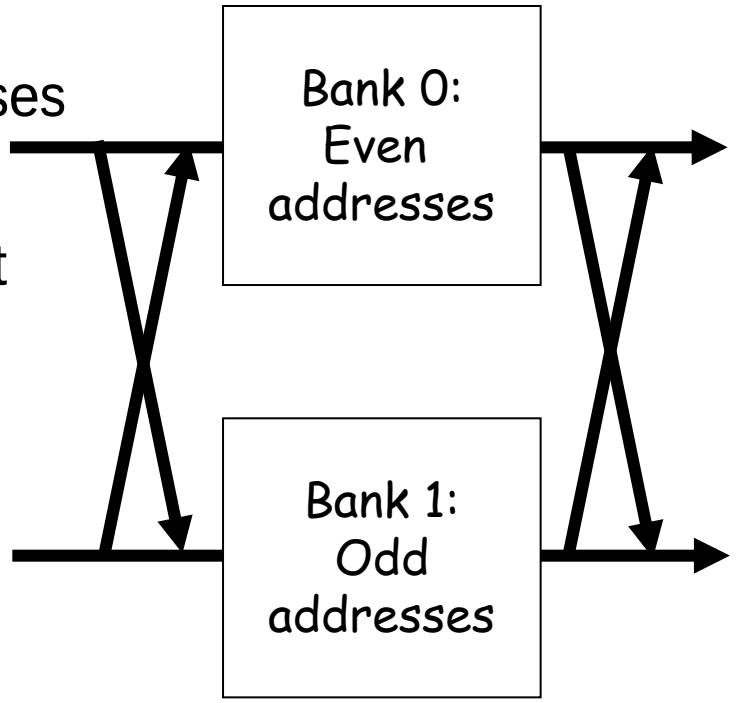
- Banking (Interleaving)
  - Bits in address determines which bank an address maps to
    - Address space partitioned into separate banks
    - Which bits to use for “bank address”?

  + No increase in data store area

  -- Cannot always satisfy multiple accesses

  Why?

  -- Crossbar interconnect in input/output



- Bank conflicts
  - Two accesses are to the same bank
  - How can these be reduced?
    - Hardware? Software?

Evaluation of Design Options

- Which alternative is better?
  - true multi-porting
  - virtual multi-porting (time sharing a port)
  - multiple cache copies
  - banking (interleaving)
  - How do we answer this question?

- Simulation
  - See Juan et al.'s evaluation of above options: "Data caches for superscalar processors," ICS 1997.
  - What are the shortcomings of their evaluation?
  - Can one do better with sole simulation?