Reminders

- Homework 5: Due March 26
- Lab 5: Due April 6
  - Branch prediction and caching (high-level simulation)
Cache Performance
Cache Parameters vs. Miss Rate

- Cache size
- Block size
- Associativity
- Replacement policy
- Insertion/Placement policy
## Cache Size

- **Cache size**: total data (not including tag) capacity
  - bigger can exploit temporal locality better
  - not ALWAYS better

- **Too large a cache** adversely affects hit and miss latency
  - smaller is faster => bigger is slower
  - access time may degrade critical path

- **Too small a cache**
  - doesn’t exploit temporal locality well
  - useful data replaced often

- **Working set**: the whole set of data the executing application references
  - Within a time interval
Block Size

- Block size is the data that is associated with an address tag
  - not necessarily the unit of transfer between hierarchies
  - **Sub-blocking**: A block divided into multiple pieces (each with V bit)
    - Can improve “write” performance

- Too small blocks
  - don’t exploit spatial locality well
  - have larger tag overhead

- Too large blocks
  - too few total # of blocks → less temporal locality exploitation
  - waste of cache space and bandwidth/energy if spatial locality is not high
Large Blocks: Critical-Word and Subblocking

- Large cache blocks can take a long time to fill into the cache
  - fill cache line **critical word first**
  - restart cache access before complete fill

- Large cache blocks can waste bus bandwidth
  - divide a block into subblocks
  - associate separate valid bits for each subblock
  - **When is this useful?**

```plaintext
v d subblock v d subblock • • • • v d subblock tag
```
Associativity

- How many blocks can map to the same index (or set)?

- Larger associativity
  - lower miss rate, less variation among programs
  - diminishing returns, higher hit latency

- Smaller associativity
  - lower cost
  - lower hit latency
    - Especially important for L1 caches

- Power of 2 associativity?
Classification of Cache Misses

- Compulsory miss
  - first reference to an address (block) always results in a miss
  - subsequent references should hit unless the cache block is displaced for the reasons below
  - dominates when locality is poor

- Capacity miss
  - cache is too small to hold everything needed
  - defined as the misses that would occur even in a fully-associative cache (with optimal replacement) of the same capacity

- Conflict miss
  - defined as any miss that is neither a compulsory nor a capacity miss
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
Improving Cache “Performance”

- Remember
  - Average memory access time (AMAT)
    \[ \text{AMAT} = (\text{hit-rate} \times \text{hit-latency}) + (\text{miss-rate} \times \text{miss-latency}) \]

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

- Reducing miss latency/cost

- Reducing hit latency
Improving Basic Cache Performance

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

- Reducing miss latency/cost
  - Multi-level caches
  - Critical word first
  - Subblocking/sectoring
  - Better replacement/insertion policies
  - Non-blocking caches (multiple cache misses in parallel)
  - Multiple accesses per cycle
  - Software approaches
Victim Cache: Reducing Conflict Misses


- 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[\text{lg}(K/N)-1: 0]\}, \{1,Addr[\text{lg}(K/N)-1: 0]\}, \ldots, \{N-1,Addr[\text{lg}(K/N)-1: 0]\}
Skewed Associative Caches (I)

- Basic 2-way associative cache structure

Way 0

Same index function for each way

Way 1

Tag  Index  Byte in Block

=?  =?
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

- 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!

```c
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;
}
```
How Do We Make This Cache-Friendly?

- 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?

```c
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;
}
```
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/enhancements to associativity
    - Victim caches, hashing, pseudo-associativity, skewed associativity
  - Better replacement/insertion policies
  - Software approaches

- Reducing miss latency/cost
  - Multi-level caches
  - Critical word first
  - Subblocking/sectoring
  - Better replacement/insertion policies
  - Non-blocking caches (multiple cache misses in parallel)
  - Multiple accesses per cycle
  - Software approaches
Memory Level Parallelism (MLP) means generating and servicing multiple memory accesses in parallel [Glew’ 98].

Several techniques to improve MLP (e.g., out-of-order execution).

MLP varies. Some misses are isolated and some parallel.

How does this affect cache replacement?
Traditional Cache Replacement Policies

- Traditional cache replacement policies try to reduce miss count.

- **Implicit assumption**: Reducing miss count reduces memory-related stall time.

- Misses with varying cost/MLP **breaks** this assumption!

- Eliminating an isolated miss helps performance more than eliminating a parallel miss.

- Eliminating a higher-latency miss could help performance more than eliminating a lower-latency miss.
Misses to blocks P1, P2, P3, P4 can be parallel
Misses to blocks S1, S2, and S3 are isolated

Two replacement algorithms:
1. Minimizes miss count (Belady’s OPT)
2. Reduces isolated miss (MLP-Aware)

For a fully associative cache containing 4 blocks
Fewest Misses ≠ Best Performance

Belady’s OPT replacement

MLP-Aware replacement
MLP-Aware Cache Replacement

- How do we incorporate MLP into replacement decisions?

  - Required reading for this week
Enabling Multiple Outstanding Misses
Handling Multiple Outstanding Accesses

- **Non-blocking or lockup-free** caches

- **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 entry
  - 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 Entry

<table>
<thead>
<tr>
<th>1</th>
<th>27</th>
<th>1</th>
<th>1</th>
<th>3</th>
<th>5</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Valid</strong></td>
<td><strong>Block Address</strong></td>
<td><strong>Issued</strong></td>
<td><strong>Valid</strong></td>
<td><strong>Type</strong></td>
<td><strong>Block Offset</strong></td>
<td><strong>Destination</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><strong>Valid</strong></td>
<td><strong>Type</strong></td>
<td><strong>Block Offset</strong></td>
<td><strong>Destination</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><strong>Valid</strong></td>
<td><strong>Type</strong></td>
<td><strong>Block Offset</strong></td>
<td><strong>Destination</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><strong>Valid</strong></td>
<td><strong>Type</strong></td>
<td><strong>Block Offset</strong></td>
<td><strong>Destination</strong></td>
</tr>
</tbody>
</table>

Load/store 0
Load/store 1
Load/store 2
Load/store 3
MSHR Operation

- On a cache miss:
  - Search MSHRs 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, deallocate 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
Enabling High Bandwidth Caches (and Memories in General)
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 latency, power, area
  - What about read and write to the same location at the same time?
    - Peripheral logic needs to handle this
Peripheral Logic for True Multiporting
Peripheral Logic for True Multiporting
Handling Multiple Accesses per Cycle (I)

- **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 satisfy multiple accesses to the same bank 
  -- Crossbar interconnect in input/output

- **Bank conflicts**
  - Two accesses are to the same bank
  - How can these be reduced? 
    - Hardware? Software?
Interleaving (banking)

- **Problem**: a single monolithic memory array takes long to access and does not enable multiple accesses in parallel

- **Goal**: Reduce the latency of memory array access and enable multiple accesses in parallel

- **Idea**: Divide the array into multiple banks that can be accessed independently (in the same cycle or in consecutive cycles)
  - Each bank is smaller than the entire memory storage
  - Accesses to different banks can be overlapped

- **Issue**: How do you map data to different banks? (i.e., how do you interleave data across banks?)
Further Readings on Caching and MLP
