Announcements

- **Milestone II**
  - Due November 4, Friday
  - Please talk with us if you are not making good progress

- **CALCM Seminar Wednesday Nov 2 at 4pm**
  - Hamerschlag Hall, D-210
  - Edward Suh, Cornell
  - Hardware-Assisted Run-Time Monitoring for Trustworthy Computing Systems
Review Sets 11 and 12

- **Was Due Today (Oct 31)**
  - Complete reviews even if late

- **Due Next Monday (Nov 7)**
Last Lecture

- Caching basics
Today

- More caching
Review: Handling Writes (Stores)

- Do we allocate a cache block on a write miss?
  - Allocate on write miss: Yes
  - No-allocate on write miss: No

- Allocate on write miss
  + Can consolidate writes instead of writing each of them individually to next level
  -- Requires (?) transfer of the whole cache block
  + Simpler because write misses can be treated the same way as read misses

- No-allocate
  + Conserves cache space if locality of writes is low
Inclusion vs. Exclusion in Multi-Level Caches

- **Inclusive caches**
  - Every block existing in the first level also exists in the next level
  - When fetching a block, place it in all cache levels. Tradeoffs:
    -- Leads to *duplication of data* in the hierarchy: less efficient
    -- Maintaining inclusion takes effort (forced evictions)
    + But makes *cache coherence* in multiprocessors easier
      - Need to track other processors’ accesses only in the highest-level cache

- **Exclusive caches**
  - The blocks contained in cache levels are mutually exclusive
  - When evicting a block, do you write it back to the next level?
    + More efficient utilization of cache space
    + (Potentially) More flexibility in replacement/placement
    -- More blocks/levels to keep track of to ensure cache coherence; takes effort

- **Non-inclusive caches**
  - No guarantees for inclusion or exclusion: simpler design
  - Most Intel processors
Maintaining Inclusion and Exclusion

When does maintaining inclusion take effort?
- L1 block size < L2 block size
- L1 associativity > L2 associativity
- Prefetching into L2
- When a block is evicted from L2, need to evict all corresponding subblocks from L1 → keep 1 bit per subblock in L2 saying “also in L1”
- When a block is inserted, make sure all higher levels also have it

When does maintaining exclusion take effort?
- L1 block size ≠ L2 block size
- Prefetching into any cache level
- When a block is inserted into any level, ensure it is not in any other
Multi-level Caching in a Pipelined Design

- First-level caches (instruction and data)
  - Decisions very much affected by cycle time
  - Small, lower associativity
- Second-level caches
  - Decisions need to balance hit rate and access latency
  - Usually large and highly associative; latency not as important
  - Serial tag and data access

- Serial vs. Parallel access of levels
  - Serial: Second level cache accessed only if first-level misses
  - Second level does not see the same accesses as the first
    - First level acts as a filter. Can you exploit this fact to improve hit rate in the second level cache?
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


- **Idea:** Use a small (fully-associative) buffer 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 the cache
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]\}, \ldots, \{N-1,Addr[\lg(K/N)-1: 0]\}
Skewed Associative Caches (I)

- Basic 2-way associative cache structure

![Diagram of a basic 2-way associative cache structure with ways 0 and 1, showing the same index function for each way and the tag, index, and byte in block fields.](image)
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 $x[i,j]$

<table>
<thead>
<tr>
<th>Poor code</th>
<th>Better code</th>
</tr>
</thead>
<tbody>
<tr>
<td>for $i = 1$, rows</td>
<td>for $j = 1$, columns</td>
</tr>
<tr>
<td>for $j = 1$, columns</td>
<td>for $i = 1$, rows</td>
</tr>
<tr>
<td>sum = sum + $x[i,j]$</td>
<td>sum = sum + $x[i,j]$</td>
</tr>
</tbody>
</table>

- 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!
How Do We Make This Cache-Friendly?

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

```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;
}
```

- **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
Midterm I Results
Midterm I Results

- Solutions will be posted
  - Check the solutions and make sure you know the answers

- Max possible score 185 / 167 (including bonus)

- Average: 98 / 167
- Median: 94 / 167
- Maximum: 169 / 167
- Minimum: 29 / 167