Reminder: Homework 5 (Wednesday)

- Due April 3 (Wednesday!)
- Topics: Vector processing, VLIW, Virtual memory, Caching
Reminder: Lab Assignment 5 (Friday)

- Lab Assignment 5
  - Due Friday, April 5
  - Modeling caches and branch prediction at the microarchitectural level (cycle level) in C

- Extra credit: Cache design optimization
  - Size, block size, associativity
  - Replacement and insertion policies
  - Cache indexing policies
  - Anything else you would like

- TAs will go over the baseline simulator in lab sessions
Heads Up: Midterm II Coming

- Originally scheduled for April 10
- Will likely move to the week after
Last Lecture

- More caching
  - Replacement policy
  - Sectored caches
  - Multi-level caching
  - Write policies
  - Virtual memory – cache interaction
    - VIVT, PIPT, VIPT caches
    - Homonyms and synonyms
Today

- Wrap up virtual memory – cache interaction
- Improving cache (and memory hierarchy) performance
- Enabling multiple accesses in parallel
Virtual Memory and Cache Interaction
Review: Homonyms and Synonyms

- **Homonym: Same VA can map to two different PAs**
  - Why?
    - VA is in different processes

- **Synonym: Different VAs can map to the same PA**
  - Why?
    - Different pages can share the same physical frame within or across processes
    - Reasons: shared libraries, shared data, copy-on-write pages within the same process, ...

- Do homonyms and synonyms create problems when we have a cache?
  - Is the cache virtually or physically addressed?
Review: Cache-VM Interaction

CPU

TLB

cache

lower hier.

CPU

cache

lower hier.

CPU

cache

tlb

lower hier.

physical cache  virtual (L1) cache  virtual-physical cache
Review: Virtual-Physical Cache

VIPT code

Where can the same physical address be in the code?
Review: Virtually-Indexed Physically-Tagged

- If $C \leq (\text{page}\_\text{size} \times \text{associativity})$, the cache index bits come only from page offset (same in VA and PA)
- If both cache and TLB are on chip
  - index both arrays concurrently using VA bits
  - check cache tag (physical) against TLB output at the end
Review: Virtually-Indexed Physically-Tagged

- If $C > (\text{page\_size} \times \text{associativity})$, the cache index bits include VPN
  $\Rightarrow$ Synonyms can cause problems
    - The same physical address can exist in two locations

- Solutions?

![Diagram of cache lookup process]

- TLB hit?
- cache hit?
Review: Solutions to the Synonym Problem

- Limit cache size to (page size times associativity)
  - get index from page offset

- On a write to a block, search all possible indices that can contain the same physical block, and update/invalidate
  - Used in Alpha 21264, MIPS R10K

- Restrict page placement in OS
  - make sure index(VA) = index(PA)
  - Called page coloring
  - Used in many SPARC processors
An Exercise

Problem 5 from

- ECE 741 midterm exam Problem 5, Spring 2009
An Exercise (I)

We have a byte-addressable toy computer that has a physical address space of 512 bytes. The computer uses a simple, one-level virtual memory system. The page table is always in physical memory. The page size is specified as 8 bytes and the virtual address space is 2 KB.

Part A.

i. (1 point)
How many bits of each virtual address is the virtual page number?

ii. (1 point)
How many bits of each physical address is the physical frame number?
We would like to add a 128-byte *write-through* cache to enhance the performance of this computer. However, we would like the cache access and address translation to be performed simultaneously. In other words, we would like to index our cache using a virtual address, but do the tag comparison using the physical addresses (virtually-indexed physically-tagged). The cache we would like to add is direct-mapped, and has a block size of 2 bytes. The replacement policy is LRU. Answer the following questions:

iii. (1 point)
How many bits of a virtual address are used to determine which byte in a block is accessed?

iv. (2 points)
How many bits of a virtual address are used to index into the cache? Which bits exactly?

v. (1 point)
How many bits of the virtual page number are used to index into the cache?

vi. (5 points)
What is the size of the tag store in bits? Show your work.
Suppose we have two processes sharing our toy computer. These processes share some portion of the physical memory. Some of the virtual page-physical frame mappings of each process are given below:

<table>
<thead>
<tr>
<th>PROCESS 0</th>
<th>PROCESS 1</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Virtual Page</strong></td>
<td><strong>Physical Frame</strong></td>
</tr>
<tr>
<td>Page 0</td>
<td>Frame 0</td>
</tr>
<tr>
<td>Page 3</td>
<td>Frame 7</td>
</tr>
<tr>
<td>Page 7</td>
<td>Frame 1</td>
</tr>
<tr>
<td>Page 15</td>
<td>Frame 3</td>
</tr>
</tbody>
</table>

vii. (2 points)
Give a complete physical address whose data can exist in two different locations in the cache.

viii. (3 points)
Give the indexes of those two different locations in the cache.
ix. **(5 points)**
We do not want the same physical address stored in two different locations in the 128-byte cache. We can prevent this by increasing the associativity of our virtually-indexed physically-tagged cache. What is the minimum associativity required?

---

x. **(4 points)**
Assume we would like to use a direct-mapped cache. Describe a solution that ensures that the same physical address is never stored in two different locations in the 128-byte cache.
Solutions to the Exercise


- And, more exercises are in past exams and in your homeworks...
Review: Solutions to the Synonym Problem

- Limit cache size to \((\text{page size times associativity})\)
  - get index from page offset

- On a write to a block, search all possible indices that can contain the same physical block, and update/invalidate
  - Used in Alpha 21264, MIPS R10K

- Restrict page placement in OS
  - make sure \(\text{index(VA)} = \text{index(PA)}\)
  - Called page coloring
  - Used in many SPARC processors
Some Questions to Ponder

- At what cache level should we worry about the synonym and homonym problems?

- What levels of the memory hierarchy does the system software’s page mapping algorithms influence?

- What are the potential benefits and downsides of page coloring?
Virtual Memory – DRAM Interaction

- Operating System influences where an address maps to in DRAM

<table>
<thead>
<tr>
<th>Virtual Page number (52 bits)</th>
<th>Page offset (12 bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Physical Frame number (19 bits)</td>
<td>Page offset (12 bits)</td>
</tr>
<tr>
<td>Row (14 bits)</td>
<td>Bank (3 bits)</td>
</tr>
</tbody>
</table>

- Operating system can control which bank/channel/rank a virtual page is mapped to.

- It can perform page coloring to minimize bank conflicts
- Or to minimize inter-application interference
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 =&gt; 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
    - likely-useless data transferred
    - Extra bandwidth/energy consumed
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?

```
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 \( \text{Addr} \), sequentially look up: \( \{0, \text{Addr}[\lg(K/N)-1: 0]\} \), \( \{1, \text{Addr}[\lg(K/N)-1: 0]\} \), ..., \( \{N-1, \text{Addr}[\lg(K/N)-1: 0]\} \),...
Skewed Associative Caches (I)

- Basic 2-way associative cache structure

```
Tag    Index    Byte in Block
```

Same index function for each way
Skewed Associative Caches (II)

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

```
Way 0            Way 1

tag          index         byte in block
f0
```

same index redistributed to different sets

same index same set

=?
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**

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

**Better code**

```plaintext```
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)

- 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

Hit/Miss: H H H H M  H H H H H  M M M M M
Time: stall
Misses=4  Stalls=4
Belady’s OPT replacement

Hit/Miss: H M M M M  H M M M M  H H H H H
Time: stall  Saved cycles
Misses=6  Stalls=2
MLP-Aware replacement
MLP-Aware Cache Replacement

- How do we incorporate MLP into replacement decisions?

  - Required reading for this week