Prefetching and Writing to Caches
Today’s Lecture

- We’ll also see how to mitigate cache misses through pre-fetching.
- Writing to caches
  - Minimize accesses to main memory
  - keeping memory consistent & write-allocation.
How do we avoid stalls from cache misses?

Hit: Stall 1 cycle
Miss: Stall many (~16-200) cycles
Non-blocking caches remember cache misses and the dependencies they affect

- This cache design stalls only when
  - Data is needed
  - Or too many misses outstanding

- Programmer can exploit by “hoist”ing loads up from their uses, but...
  - Uses up a register
  - For potentially many cycles (~100 to memory)
  - Might be guessing what will be accessed.
Software prefetching lets the programmer predict cache misses and fill cache in advance

- Code prefetch as a load without a destination register
  - e.g., on SPIM, `lw $0, 64($a0) # write to the zero register`
- Prefetches are hints to the processor:
  - “I think I might use cache block containing this address”
  - Hardware will try to move the block into the cache.
  - But, hardware can ignore (if busy)

```c
for (int i = 0; i < LARGE; i++) {
    prefetch A[i + 16];   // prefetch 16 iterations ahead.
    computation A[i];
}
```
Prefetching may improve performance by removing or reducing stalls from cache misses

Without Pre-fetching

int array[SIZE];
int A = 0;

for (int i = 0 ; i < 200000 ; ++ i) {
    for (int j = 0 ; j < SIZE ; ++ j) {
        A += array[j];
    }
}

With Pre-fetching

Intel Core 2 Duo
Hardware can prefetch by tracking patterns in cache hits and misses

- Inner loop has simple access streams
  - A, A+4, A+8, A+12, ...

- Example: If we get a pair of sequential misses (blocks X, X+1), predict a stream
  - Fetch the next two blocks (X+2, X+3)
  - Fetch more blocks if predictions come true

- Can learn strides as well (X, X+16, X+32, ...) and (X, X-1, X-2, ...)
Use the program counter to track and prefetch multiple streams of addresses

```c
for (int i=0 ; i < LARGE ; i ++) {
    C[i] = A[i] + B[i];
}
```

- 3 streams might confuse naïve prefetcher.
- A, B, and C accessed by different instructions

```
PC   LW   LW
A, B, C, A+4, B+4, C+4
```
Software prefetching is primarily used for non-strided memory access patterns

- Example: linked data structures:
  - Lists
  - Arrays of pointers

```c
struct_t *A[SIZE];
for (i = 0; i < SIZE; i++) {
    process(A[i]);
}
```
Four important questions

1. When we copy a block of data from main memory to the cache, where exactly should we put it?

2. How can we tell if a word is already in the cache, or if it has to be fetched from main memory first?

3. Eventually, the small cache memory might fill up. To load a new block from main RAM, we’d have to replace one of the existing blocks in the cache... which one?

4. How can write operations be handled by the memory system?
Writing to the cache can cause the cache to be inconsistent with main memory.
Write-through cache maintains consistency by writing to both main memory and the cache:

```
Index V Tag Data Address Data
... ... ... ...
110 1 11010 21763 1101 0110 21763
... ...
```

- Why is this not so good?
Write-back cache updates main memory only when a cache block needs to be replaced

- Use a dirty bit to track where the cache is inconsistent with memory

Okay, since loads read from the cache, not main memory
Write back to main memory when replacing a dirty cache block.

Example: Load from Mem[1000 1110], collides with current data

1) Dirty bit is 1 -> Update main memory

<table>
<thead>
<tr>
<th>Index</th>
<th>V</th>
<th>Dirty</th>
<th>Tag</th>
<th>Data</th>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>1</td>
<td>1</td>
<td>11010</td>
<td>21763</td>
<td>1101 0110</td>
<td>21763</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

2) Load new data -> new data is “clean”

<table>
<thead>
<tr>
<th>Index</th>
<th>V</th>
<th>Dirty</th>
<th>Tag</th>
<th>Data</th>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>1</td>
<td>0</td>
<td>10001</td>
<td>1225</td>
<td>1101 0110</td>
<td>21763</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
A **write miss** occurs when we try to write to an address that is not already in the cache.

Example: Store 21763 into Mem[1101 0110].

When we update Mem[1101 0110], should we also load it into the cache?
**Allocate on write** strategy loads new cache block on write misses

- If that data is needed again soon, it will be available in the cache
- This is the baseline behavior of processors.
Use non-temporal stores (**write-around/write-no-allocate**) if stored data will not be used soon

```c
for (int i = 0; i < LARGE; i++) {
    a[i] = i;
}
```

- No temporal locality
- **write around** policy, writes directly to main memory *without* affecting the cache.

![Memory diagram](attachment:memory_diagram.png)
Summary

- Hardware prefetching handles most streams and strides.
- Software prefetching is useful for linked data structures
  - Must be added by programmer (or very smart compiler)

- Writing to a cache poses a couple of interesting issues.
  - Write-through and write-back policies keep the cache consistent with main memory in different ways for write hits.
  - Write-around and allocate-on-write are two strategies to handle write misses, differing in whether updated data is loaded into the cache.