Cache Memories
September 30, 2008

Topics

- Generic cache memory organization
- Direct mapped caches
- Set associative caches
- Impact of caches on performance
Announcements

Exam grading done

- Everyone should have gotten email with score (out of 72)
  - mean was 50, high was 70
  - solution sample should be up on website soon
- Getting your exam back
  - some got them in recitation
  - working on plan for everyone else (worst case = recitation on Monday)
- If you think we made a mistake in grading
  - please read the syllabus for details about the process for handling it
General cache mechanics

Smaller, faster, more expensive memory caches a subset of the blocks.

Data is copied between levels in block-sized transfer units.

Larger, slower, cheaper memory is partitioned into "blocks".
Cache Performance Metrics

Miss Rate
- Fraction of memory references not found in cache (misses / accesses)
  - 1 – hit rate
- Typical numbers (in percentages):
  - 3-10% for L1
  - can be quite small (e.g., < 1%) for L2, depending on size, etc.

Hit Time
- Time to deliver a line in the cache to the processor
  - includes time to determine whether the line is in the cache
- Typical numbers:
  - 1-2 clock cycle for L1
  - 5-20 clock cycles for L2

Miss Penalty
- Additional time required because of a miss
  - typically 50-200 cycles for main memory (Trend: increasing!)
Let’s think about those numbers

Huge difference between a hit and a miss
- 100X, if just L1 and main memory

Would you believe 99% hits is twice as good as 97%?
- Consider these numbers:
  
  cache hit time of 1 cycle
  miss penalty of 100 cycles

So, average access time is:

97% hits: 1 cycle + 0.03 * 100 cycles = 4 cycles
99% hits: 1 cycle + 0.01 * 100 cycles = 2 cycles
Many types of caches

**Examples**
- Hardware: L1 and L2 CPU caches, TLBs, …
- Software: virtual memory, FS buffers, web browser caches, …

**Many common design issues**
- Each cached item has a “tag” (an ID) plus contents
- Need a mechanism to efficiently determine whether given item is cached
  - Combinations of indices and constraints on valid locations
- On a miss, usually need to pick something to replace with the new item
  - Called a “replacement policy”
- On writes, need to either propagate change or mark item as “dirty”
  - Write-through vs. write-back
Hardware cache memories

Cache memories are small, fast SRAM-based memories managed automatically in hardware

- Hold frequently accessed blocks of main memory

CPU looks first for data in L1, then in main memory

Typical system structure:
Inserting an L1 Cache Between the CPU and Main Memory

The transfer unit between the CPU register file and the cache is a 4-byte word.

The transfer unit between the cache and main memory is a 4-word block (16 bytes).

The tiny, very fast CPU register file has room for four 4-byte words.

The small fast L1 cache has room for two 4-word blocks.

The big slow main memory has room for many 4-word blocks.
Inserting an L1 Cache Between the CPU and Main Memory

The tiny, very fast CPU register file has room for four 4-byte words.

The small fast L1 cache has room for two 4-word blocks.

The big slow main memory has room for many 4-word blocks.

The transfer unit between the CPU register file and the cache is a 4-byte word.

The transfer unit between the cache and main memory is a 4-word block (16 bytes).
### General Organization of a Cache

Cache is an array of sets
- Each set contains one or more lines
- Each line holds a block of data

- \( S = 2^s \) sets

- \( E \) lines per set
- \( B = 2^b \) bytes per cache block
- \( C = B \times E \times S \) data bytes

<table>
<thead>
<tr>
<th>Set</th>
<th>Valid</th>
<th>Tag</th>
<th>( 0 )</th>
<th>( 1 )</th>
<th>( \ldots )</th>
<th>( B-1 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>( \text{valid} )</td>
<td>( \text{tag} )</td>
<td>( 0 )</td>
<td>( 1 )</td>
<td>( \ldots )</td>
<td>( B-1 )</td>
</tr>
<tr>
<td>( \ldots )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>( \text{valid} )</td>
<td>( \text{tag} )</td>
<td>( 0 )</td>
<td>( 1 )</td>
<td>( \ldots )</td>
<td>( B-1 )</td>
</tr>
<tr>
<td>( \ldots )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( S-1 )</td>
<td>( \text{valid} )</td>
<td>( \text{tag} )</td>
<td>( 0 )</td>
<td>( 1 )</td>
<td>( \ldots )</td>
<td>( B-1 )</td>
</tr>
</tbody>
</table>

Cache size: \( C = B \times E \times S \) data bytes
General Organization of a Cache

Cache is an array of sets
Each set contains one or more lines
Each line holds a block of data

$S = 2^s$ sets

1 valid bit per line
$\dagger$ tag bits per line
$B = 2^b$ bytes per cache block

Cache size: $C = B \times E \times S$ data bytes
Addressing Caches

Address A:

The word at address A is in the cache if the tag bits in one of the <valid> lines in set <set index> match <tag>

The word contents begin at offset <block offset> bytes from the beginning of the block
Addressing Caches

Address A:

1. Locate the set based on <set index>
2. Locate the line in the set based on <tag>
3. Check that the line is valid
4. Locate the data in the line based on <block offset>
Example: Direct-Mapped Cache

Simplest kind of cache, easy to build
(only 1 tag compare required per access)
Characterized by exactly one line per set.

Cache size: \( C = B \times S \) data bytes
Accessing Direct-Mapped Caches

**Set selection**

- Use the set index bits to determine the set of interest.

![Diagram of cache blocks and set selection](image)
Accessing Direct-Mapped Caches

Line matching and word selection

- **Line matching**: Find a valid line in the selected set with a matching tag
- **Word selection**: Then extract the word

---

selected set (i):

![Diagram showing cache line and tag comparison]

- =1? (1) The valid bit must be set
- = ? (2) The tag bits in the cache line must match the tag bits in the address

If (1) and (2), then cache hit

- \[ t \text{ bits} \quad s \text{ bits} \quad b \text{ bits} \]
- \[ 0110 \quad i \quad 100 \]
- \[ m^{-1} \text{ tag} \quad \text{set index} \quad \text{block offset}^{0} \]
Accessing Direct-Mapped Caches

Line matching and word selection

- **Line matching**: Find a valid line in the selected set with a matching tag
- **Word selection**: Then extract the word

(3) If cache hit, block offset selects starting byte.

<table>
<thead>
<tr>
<th>t bits</th>
<th>s bits</th>
<th>b bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>0110</td>
<td>i</td>
<td>100</td>
</tr>
</tbody>
</table>

selected set (i):

```
1 0110
```

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>b₀</td>
<td>b₁</td>
<td>b₂</td>
<td>b₃</td>
</tr>
</tbody>
</table>

\[ m^{-1} \text{ tag} \quad \text{set index} \quad \text{block offset}^0 \]
# Direct-Mapped Cache Simulation

*M=16 byte addresses, B=2 bytes/block, S=4 sets, E=1 entry/set*

Address trace (reads):

- 0 \( [0000_2] \), miss
- 1 \( [0001_2] \), hit
- 7 \( [0111_2] \), miss
- 8 \( [1000_2] \), miss
- 0 \( [0000_2] \) miss

<table>
<thead>
<tr>
<th>v</th>
<th>tag</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>( M[0-1] )</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>( M[6-7] )</td>
</tr>
</tbody>
</table>
Example: Set Associative Cache

Characterized by more than one line per set

E-way associative cache
Accessing Set Associative Caches

Set selection

- identical to direct-mapped cache

<table>
<thead>
<tr>
<th>Set 0:</th>
<th>Set 1:</th>
<th>Set S-1:</th>
</tr>
</thead>
<tbody>
<tr>
<td>valid</td>
<td>valid</td>
<td>valid</td>
</tr>
<tr>
<td>tag</td>
<td>tag</td>
<td>tag</td>
</tr>
<tr>
<td>cache block</td>
<td>cache block</td>
<td>cache block</td>
</tr>
</tbody>
</table>

### Tag Set Index Block Offset

<table>
<thead>
<tr>
<th>m-1 tag</th>
<th>set index</th>
<th>block offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>0 0 0 1</td>
<td>0</td>
</tr>
</tbody>
</table>

† bits s bits b bits
Accessing Set Associative Caches

Line matching and word selection

- must compare the tag in each valid line in the selected set.

=1? (1) The valid bit must be set

(2) The tag bits in one of the cache lines must match the tag bits in the address

If (1) and (2), then cache hit
Accessing Set Associative Caches

Line matching and word selection

- Word selection is the same as in a direct mapped cache

(3) If cache hit, block offset selects starting byte.

<table>
<thead>
<tr>
<th>t bits</th>
<th>s bits</th>
<th>b bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>0110</td>
<td>i</td>
<td>100</td>
</tr>
</tbody>
</table>

\[
\begin{array}{cccccc}
\text{tag} & \text{set index} & \text{block offset}^0 \\
0110 & i & 100 \\
\end{array}
\]
2-Way Associative Cache Simulation

M=16 byte addresses, B=2 bytes/block,
S=2 sets, E=2 entry/set

Address trace (reads):

0 \ [0000_2], \ miss
1 \ [0001_2], \ hit
7 \ [0111_2], \ miss
8 \ [1000_2], \ miss
0 \ [0000_2] \ hit
Notice that middle bits used as index
Why Use Middle Bits as Index?

High-Order Bit Indexing
- Adjacent memory lines would map to same cache entry
- Poor use of spatial locality

Middle-Order Bit Indexing
- Consecutive memory lines map to different cache lines
- Can hold $S*B*E$-byte region of address space in cache at one time
Why Use Middle Bits as Index?

High-Order Bit Indexing
- Adjacent memory lines would map to same cache entry
- Poor use of spatial locality

Middle-Order Bit Indexing
- Consecutive memory lines map to different cache lines
- Can hold $S*B*E$-byte region of address space in cache at one time
Sidebar: Multi-Level Caches

Options: separate data and instruction caches, or a unified cache

- **Processor**: Regs, L1 d-cache, L1 i-cache
- **Unified L2 Cache**
- **Memory**
- **Disk**

### Options:

- **Size**:
  - 200 B
  - 8-64 KB
  - 1-4MB SRAM
  - 128 MB DRAM
  - 30 GB

- **Speed**:
  - 3 ns
  - 3 ns
  - 6 ns
  - 60 ns
  - 8 ms

- **$/Mbyte**:
  - $100/MB
  - $1.50/MB
  - $0.05/MB

- **Line Size**:
  - 8 B
  - 32 B
  - 32 B
  - 8 KB

**Larger, slower, cheaper**
What about writes?

Multiple copies of data exist:
- L1
- L2
- Main Memory
- Disk

What to do when we write?
- Write-through
- Write-back
  - need a dirty bit
  - What to do on a write-miss?

What to do on a replacement?
- Depends on whether it is write through or write back
Software caches are more flexible

Examples
- File system buffer caches, web browser caches, etc.

Some design differences
- Almost always fully associative
  - so, no placement restrictions
  - index structures like hash tables are common
- Often use complex replacement policies
  - misses are very expensive when disk or network involved
  - worth thousands of cycles to avoid them
- Not necessarily constrained to single “block” transfers
  - may fetch or write-back in larger units, opportunistically
Locality Example #1

Being able to look at code and get a qualitative sense of its locality is a key skill for a professional programmer.

Question: Does this function have good locality?

```c
int sum_array_rows(int a[M][N])
{
    int i, j, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum;
}
```
Locality Example #2

Question: Does this function have good locality?

```c
int sum_array_cols(int a[M][N])
{
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];

    return sum;
}
```
Locality Example #3

**Question:** Can you permute the loops so that the function scans the 3-d array \( a[] \) with a stride-1 reference pattern (and thus has good spatial locality)?

```c
int sum_array_3d(int a[M][N][N])
{
    int i, j, k, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            for (k = 0; k < N; k++)
                sum += a[k][i][j];

    return sum;
}
```