4 November

- 8 classes to go!
- Read 7.3-7.5
- Section 7.5 especially important!
- New Assignment on the web

Direct-Mapping Example

- With 8 byte BLOCKS, the bottom 3 bits determine the byte in the BLOCK
- With 4 cache BLOCKS, the next 2 bits determine which BLOCK to use

<table>
<thead>
<tr>
<th>Memory</th>
<th>1024</th>
<th>1000</th>
<th>1040</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tag Data</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1024 44 99</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000 17 23</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1016 29 38</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Miss Penalty and Rate

- The MISS PENALTY is the time it takes to read the memory if it isn’t in the cache
  - 50 to 100 cycles is common.
- The MISS RATE is the fraction of accesses which MISS
- The HIT RATE is the fraction of accesses which HIT
- MISS RATE + HIT RATE = 1

Suppose a particular cache has a MISS PENALTY of 100 cycles and a HIT RATE of 95%. The CPI for load is normally 5 but on a miss it is 105. What is the average CPI for load?

Average CPI = 5 * 0.95 + 105 * 0.05 = 10

Suppose MISS PENALTY = 120 cycles?
then CPI = 11 (slower memory doesn’t hurt much)

Some Associativity can help

- Direct-Mapped caches are very common but can cause problems...
- SET ASSOCIATIVITY can help.
- Multiple Direct-mapped caches, then compare multiple TAGS
  - 2-way set associative = 2 direct mapped + 2 TAG comparisons
  - 4-way set associative = 4 direct mapped + 4 TAG comparisons
- Now array size = power of 2 doesn’t get us in trouble
- But
  - slower
  - less memory in same area
  - maybe direct mapped wins...

Associative Cache

- With 8 byte BLOCKS, the bottom 3 bits determine the byte in the BLOCK
- With 4 cache BLOCKS, the next 2 bits determine which BLOCK to use
What about store?

- What happens in the cache on a store?
  - WRITE BACK CACHE → put it in the cache, write on replacement
  - WRITE THROUGH CACHE → put in cache and in memory
- What happens on store and a MISS?
  - WRITE BACK will fetch the line into cache
  - WRITE THROUGH might just put it in memory

Cache Block Size and Hit Rate

- Increasing the block size tends to decrease miss rate:

<table>
<thead>
<tr>
<th>Program</th>
<th>Block size in words</th>
<th>Instruction miss rate</th>
<th>Data miss rate</th>
<th>Effective combined miss rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcc</td>
<td>1 KB</td>
<td>4.1%</td>
<td>2.3%</td>
<td>4.4%</td>
</tr>
<tr>
<td>spice</td>
<td>2 KB</td>
<td>3.0%</td>
<td>1.3%</td>
<td>3.2%</td>
</tr>
</tbody>
</table>

- Use split caches because there is more spatial locality in code:

Cache Performance

- Simplified model:
  
  \[
  \text{execution time} = (\text{execution cycles} + \text{stall cycles}) \times \text{cycle time}
  \]
  
  \[
  \text{stall cycles} = \# \text{ of instructions} \times \text{miss ratio} \times \text{miss penalty}
  \]

- Two ways of improving performance:
  - decreasing the miss ratio
  - decreasing the miss penalty

What happens if we increase block size?

Associative Performance

Multilevel Caches

- We can reduce the miss penalty with a 2\textsuperscript{nd} level cache
- Add a second level cache:
  - often primary cache is on the same chip as the processor
  - use SRAMs to add another cache above primary memory (DRAM)
  - miss penalty goes down if data is in 2nd level cache
- Example:
  - Base CPI=1.0 on a 500MHz machine with a 5% miss rate, 200ns DRAM access
  - Adding 2nd level cache with 20ns access time decreases miss rate to 2%
- Using multilevel caches:
  - try and optimize the hit time on the 1st level cache
  - try and optimize the miss rate on the 2nd level cache

Matrix Multiply

- A VERY common operation in scientific programs
- Multiply a L×M matrix by an M×N matrix to get an L×N matrix result
- This requires L×N inner products each requiring M * and +
- So 2*L*M*N floating point operations
- Definitely a FLOATING POINT INTENSIVE application
- L=M=N=100, 2 Million floating point operations
Matrix Multiply

```c
const int L = 2;
const int M = 3;
const int N = 4;
void mm(double A[L][M], double B[M][N], double C[L][N])
{
    for (int i = 0; i < L; i++)
        for (int j = 0; j < N; j++)
            double sum = 0.0;
            for (int k = 0; k < M; k++)
                sum = sum + A[i][k] * B[k][j];
            C[i][j] = sum;
}
```

Matrix Memory Layout

Our memory is a 1D array of bytes

How can we put a 2D thing in a 1D memory?

```
0 0 1 0 2
0 0 0 0 1
0 0 0 0 1
0 0 0 0 2
0 0 0 0 2
```

`double A[2][3]:`

Row Major  Column Major

```
0 0
0 2
1 0
1 1
1 0
0 0
```

`addr = base + (i*2 + j)*8`

Change Index * to +

The inner loop takes all the time

```
L1: l.d $f1, 0($t1)
    add $t1, $t1, AColStep
    l.d $f2, 0($t2)
    add $t2, $t2, BRowStep
    mul $f3, $f1, $f2
    add $f4, $f4, $f3
    bne $t0, zero, L1
```

The inner loop takes all the time

```
L1: l.d $f1, 0($t1)
    add $t1, $t1, AColStep
    l.d $f2, 0($t2)
    add $t2, $t2, BRowStep
    mul $f3, $f1, $f2
    add $f4, $f4, $f3
    bne $t0, LastA, L1
```

Eliminate k, use an address instead

The inner loop takes all the time

```
L1: l.d $f1, 0($t1)
    add $t1, $t1, AColStep
    l.d $f2, 0($t2)
    add $t2, $t2, BRowStep
    mul $f3, $f1, $f2
    add $f4, $f4, $f3
    bne $t1, LastA, L1
```

We made it faster

The inner loop takes all the time

```
L1: l.d $f1, 0($t1)
    add $t1, $t1, AColStep
    l.d $f2, 0($t2)
    add $t2, $t2, BRowStep
    mul $f3, $f1, $f2
    add $f4, $f4, $f3
    bne $t1, LastA, L1
```

Now this is FAST! Only 7 instructions in the inner loop!

BUT...

When we try it on big matrices it slows way down.

What Up?
Now where is the time?

The inner loop takes all the time:
\[
\text{for(int } k=0; k<M; k++)
\]
\[
\text{sum = sum } + A[i][k] \times B[k][j];
\]

L1: 1.d $f1$, 0($t1$)
add $t1$, $t1$, AColStep
l.d $f2$, 0($t2$) lots of time wasted here!
add $t2$, $t2$, BRowStep
mul.d $f3$, $f1$, $f2$
add.d $f4$, $f4$, $f3$
bne $t1$, LastA, L1

Why?

The inner loop takes all the time:
\[
\text{for(int } k=0; k<M; k++)
\]
\[
\text{sum = sum } + A[i][k] \times B[k][j];
\]

L1: 1.d $f1$, 0($t1$)
add $t1$, $t1$, AColStep
l.d $f2$, 0($t2$) This load usually hits (maybe 3 of 4)
add $t2$, $t2$, BRowStep
mul.d $f3$, $f1$, $f2$
add.d $f4$, $f4$, $f3$
bne $t1$, LastA, L1

Matrix Multiply Simulation

Simulation of 2k direct-mapped cache with 32 and 16 byte blocks

Cycles/MAC

Matrix Size N x N

classes to go

- Read 7.3-7.5
- Section 7.5 especially important!