November 14

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

Cache

- Sits between CPU and main memory
- Very fast table that stores a *TAG* and *DATA*
  - *TAG* is the memory address
  - *DATA* is a copy of memory at the address given by TAG

Block	Tag	Data
0	1000	17
1	1040	1
2	1032	97
3	1008	11

Memory

1000	17
1004	23
1008	11
1012	5
1016	29
1020	38
1024	44
1028	99
1032	97
1036	25
1040	1
1044	4

Cache Access

- On load we look in the TAG entries for the address we're loading
  - Found → a *HIT*, return the DATA
  - Not Found → a *MISS*, go to memory for the data and put it and the address (TAG) in the cache

Tag	Data
1000	17
1040	1
1032	97
1008	11

Memory

1000	17
1004	23
1008	11
1012	5
1016	29
1020	38
1024	44
1028	99
1032	97
1036	25
1040	1
1044	4

Cache Lines

- Usually get more data than requested (Why?)
  - a *BLOCK (or line)* is the unit of memory stored in the cache
  - usually much bigger than 1 word, 32 bytes per line is common
  - bigger BLOCK means fewer misses because of locality
  - but bigger BLOCK means longer time on miss
  - and greater likelihood of conflict

Tag	Data	
1000	17	23
1040	1	4
1032	97	25
1008	11	5

Memory

1000	17
1004	23
1008	11
1012	5
1016	29
1020	38
1024	44
1028	99
1032	97
1036	25
1040	1
1044	4

Finding the TAG in the Cache

- A 1MByte cache may have 32k different BLOCKS each of 32 bytes
- We can't afford to sequentially search the 32k different TAGs
- *ASSOCIATIVE* memory uses hardware to compare the address to the TAGs in parallel but it is expensive and 1MByte is thus unlikely
- *DIRECT MAPPED CACHE* computes the cache entry from the address
  - multiple addresses map to the same cache line
  - use TAG to determine if right
- Choose some bits of the address to determine the Cache BLOCK
  - low 5 bits determine which byte within the BLOCK
  - we need 15 bits to determine which of the 32k different lines has the data
  - which of the 32 – 5 = 27 remaining bits should we use?

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

1024d = 1000000**00**000b → line = 00b = 0d

1000d = 0111111**01**000b → line = 01b = 1d

1040d = 1000001**0**000b → line = 10b = 2d

Tag	Data	
1024	44	99
1000	17	23
1040	1	4
1016	29	38

Memory

1000	17
1004	23
1008	11
1012	5
1016	29
1020	38
1024	44
1028	99
1032	97
1036	25
1040	1
1044	4

Direct Mapping Miss

•What happens when we now ask for address 1008?

1008d = 0111111000b → line = 10b = 2d

but earlier we put 1040d there...

1040d = 1000001000b → line = 10b = 2d

Tag	Data	
1024	44	99
1000	17	23
1008	11	5
1016	29	38

Memory

1000	17
1004	23
1008	11
1012	5
1016	29
1020	38
1024	44
1028	99
1032	97
1036	25
1040	1
1044	4

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 = 10 5 * 0.95 + 105 * 0.05 = 10

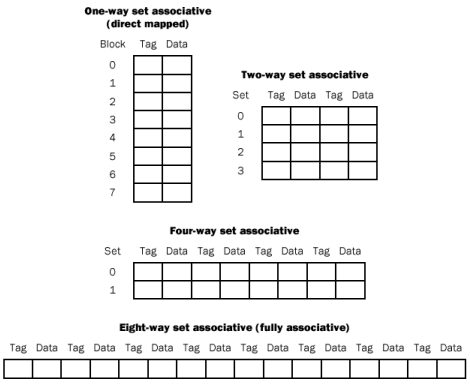
Suppose MISS PENALTY = 120 cycles?

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

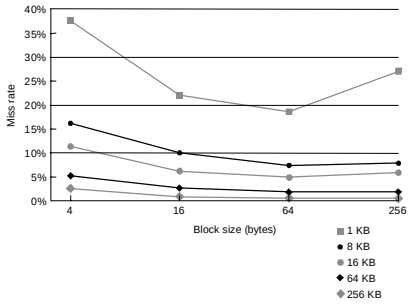


What about store?

- 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:



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

Program	Block size in words	Instruction miss rate	Data miss rate	Effective combined miss rate
gcc	1	6.1%	2.1%	5.4%
	4	2.0%	1.7%	1.9%
spice	1	1.2%	1.3%	1.2%
	4	0.3%	0.6%	0.4%

Cache Performance

- Simplified model:

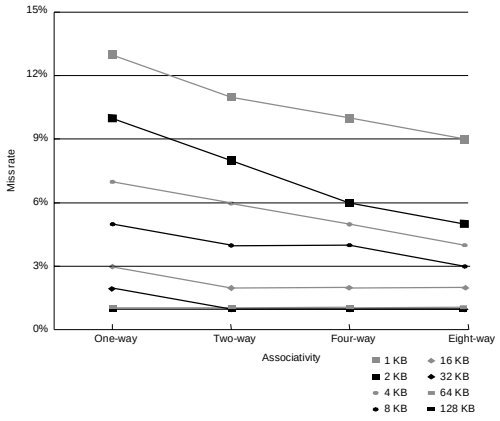
  execution time = (execution cycles + stall cycles) × cycle time

  stall cycles = # of instructions × miss ratio × 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 2nd 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 LxM matrix by an MxN matrix to get an LxN 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

```
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?

double A[2][3];

0 0	0 1	0 2
1 0	1 1	1 2

Row Major

0 0
0 1
0 2
1 0
1 1
1 2

addr = base+(i*3+j)*8

Column Major

0 0
1 0
0 1
1 1
0 2
1 2

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

Where does the time go?

The inner loop takes all the time

```
for(int k=0; k<M; k++)
    sum = sum + A[i][k] * B[k][j];
```

```
L1: mul $t1, i, M
    add $t1, $t1, k
    mul $t1, $t1, 8
    add $t1, $t1, A
    l.d $f1, 0($t1)
    mul $t2, k, N
    add $t2, $t2, j
    mul $t2, $t2, 8
    add $t2, $t2, B
    l.d $f2, 0($t2)
```

```
mul.d $f3, $f1, $f2
add.d $f4, $f4, $f3
add k, k, 1
slt $t0, k, M
bne $t0, $zero, L1
```


Change Index * to +

The inner loop takes all the time

```
for(int k=0; k<M; k++)
    sum = sum + A[i][k] * B[k][j];
```

```
L1: l.d $f1, 0($t1)
    add $t1, $t1, AColStep
    l.d $f2, 0($t2)
    add $t2, $t2, BRowStep
```

AColStep = 8

BRowStep = 8 * N

```
mul.d $f3, $f1, $f2
add.d $f4, $f4, $f3
add k, k, 1
slt $t0, k, M
bne $t0, $zero, L1
```

Eliminate k, use an address instead

The inner loop takes all the time

```
        for(int k=0; k<M; k++)
            sum = sum + A[i][k] * B[k][j];

L1: l.d $f1, 0($t1)
    add $t1, $t1, AColStep
    l.d $f2, 0($t2)
    add $t2, $t2, BRowStep

    mul.d $f3, $f1, $f2
    add.d $f4, $f4, $f3
    bne $t1, LastA, L1
```


We made it faster

The inner loop takes all the time

```
        for(int k=0; k<M; k++)
            sum = sum + A[i][k] * B[k][j];

L1: l.d $f1, 0($t1)
    add $t1, $t1, AColStep
    l.d $f2, 0($t2)
    add $t2, $t2, BRowStep

    mul.d $f3, $f1, $f2
    add.d $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.

Whas Up?

Now where is the time?

The inner loop takes all the time

```
        for(int k=0; k<M; k++)
            sum = sum + A[i][k] * B[k][j];

L1: l.d $f1, 0($t1)
    add $t1, $t1, AColStep
    l.d $f2, 0($t2)
    add $t2, $t2, BRowStep

    mul.d $f3, $f1, $f2
    add.d $f4, $f4, $f3
    bne $t1, LastA, L1
```

lots of time wasted here!

possibly a little stall right here

Why?

The inner loop takes all the time

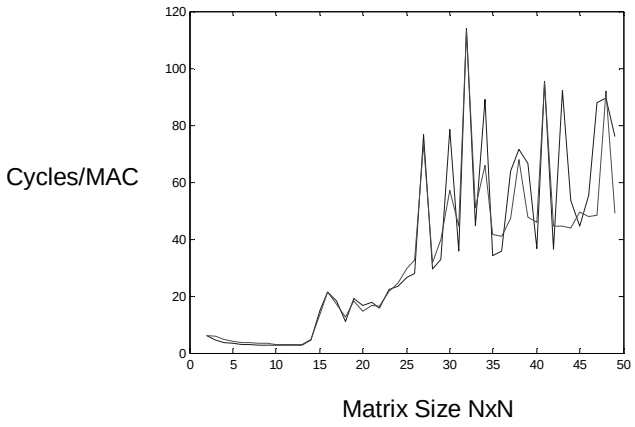
```
        for(int k=0; k<M; k++)
            sum = sum + A[i][k] * B[k][j];

L1: l.d $f1, 0($t1)
    add $t1, $t1, AColStep
    l.d $f2, 0($t2)
    add $t2, $t2, BRowStep

    mul.d $f3, $f1, $f2
    add.d $f4, $f4, $f3
    bne $t1, LastA, L1
```

This load usually hits (maybe 3 of 4)

This load always misses!

Matrix Multiply Simulation

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



classes to go

5

- Read 7.3-7.5
- Section 7.5 especially important!