CS 105
Tour of the Black Holes of Computing

Cache Memories

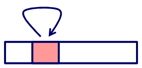
Topics

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

Locality

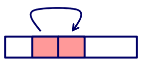
Principle of Locality: Programs tend to use data and instructions with addresses equal or near to those they have used recently

Temporal locality:



- Recently referenced items are likely to be referenced again in the near future

Spatial locality:



- Items with nearby addresses tend to be referenced close together in time

Locality Example

```
sum = 0;
for (i = 0; i < n; i++)
    sum += a[i];
return sum;
```

Data references

- Reference array elements in succession (stride-1 reference pattern). — Spatial locality
- Reference variable `sum` each iteration. — Temporal locality

Instruction references

- Reference instructions in sequence. — Spatial locality
- Cycle through loop repeatedly. — Temporal locality

Qualitative Estimates of Locality

Claim: 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 with respect to array `a`?

```
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

Question: Does this function have good locality with respect to array `a`?

```
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;
}
```

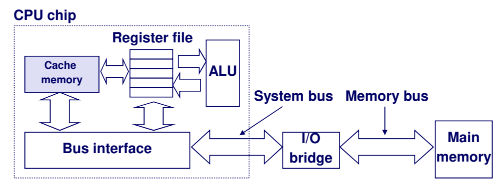

Cache Memories

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

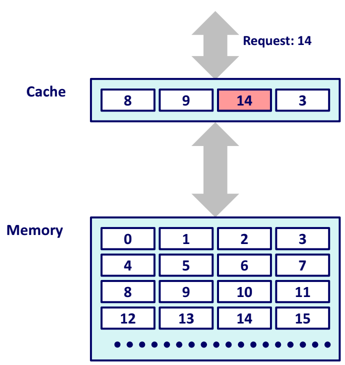
- Hold frequently accessed blocks of main memory

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

Typical system structure:



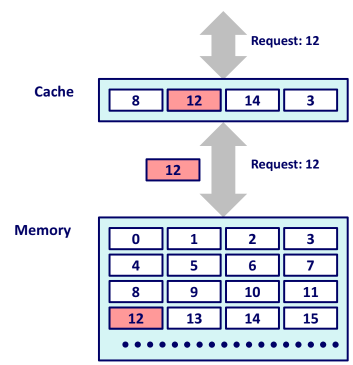
General Cache Concepts: Hit



Data in block b is needed

Block b is in cache:
Hit!

General Cache Concepts: Miss



Data in block b is needed

Block b is not in cache:

Miss!

Block b is fetched from memory

Block b is stored in cache

- Placement policy: determines where b goes
- Replacement policy: determines which block gets evicted (victim)

General Caching Concepts: Types of Cache Misses

Cold (compulsory) miss

- Cold misses occur because the cache is empty.

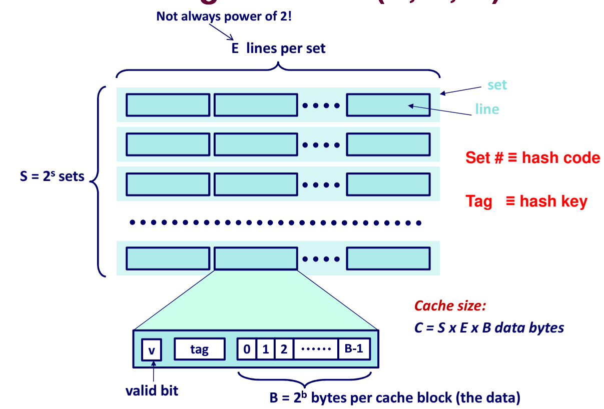
Conflict miss

- Most caches limit blocks at level k+1 to a small subset (sometimes a singleton) of the block positions at level k
  - E.g. Block i at level k+1 must go in block (i mod 4) at level k
- Conflict misses occur when the level k cache is large enough, but multiple data objects all map to the same level k block
  - E.g. Referencing blocks 0, 8, 0, 8, 0, 8, ... would miss every time

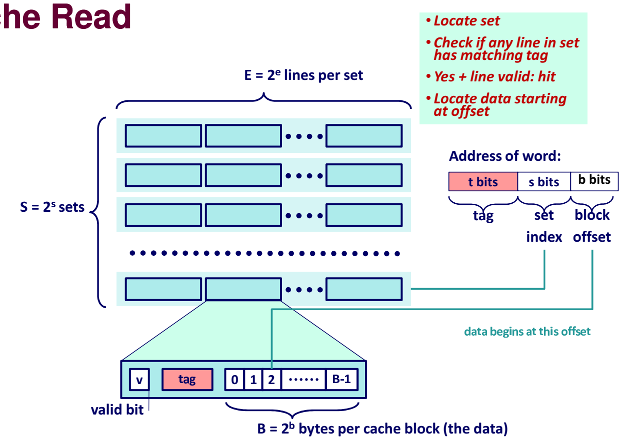
Capacity miss

- Occurs when set of active cache blocks (working set) is larger than the cache

General Cache Organization (S, E, B)

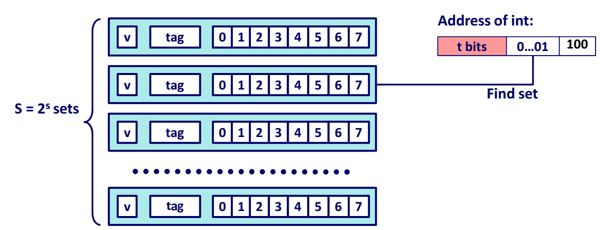


Cache Read



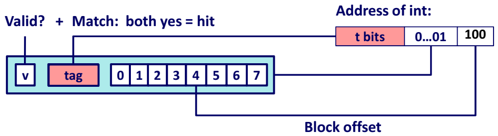
Example: Direct Mapped Cache (E = 1)

Direct mapped: One line per set
Assume cache block size 8 bytes



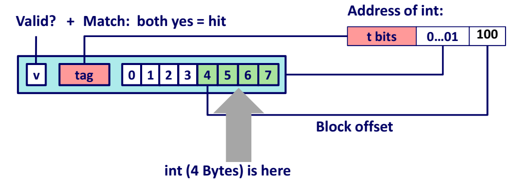
Example: Direct Mapped Cache (E = 1)

Direct mapped: One line per set
Assume cache block size 8 bytes



Example: Direct Mapped Cache (E = 1)

Direct mapped: One line per set
Assume cache block size 8 bytes



If tag doesn't match: old line is *evicted* and replaced

Direct-Mapped Cache Simulation

t=1	s=2	b=1
x	xx	x

M=16 bytes (4-bit addresses), B=2 bytes/block,
S=4 sets, E=1 Blocks/set

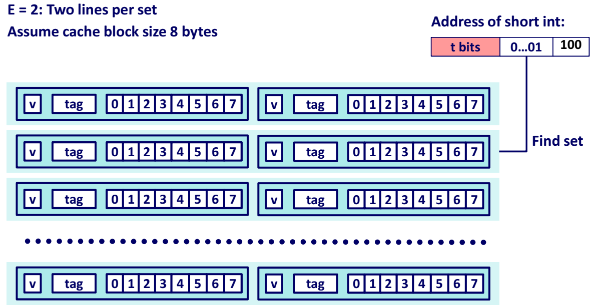
Address trace (reads, one byte per read):

0	**[$\underline{0}\underline{00}0_2$],**	miss
1	**[$\underline{0}\underline{00}1_2$],**	hit
7	**[$\underline{0}\underline{11}1_2$],**	miss
8	**[$\underline{1}\underline{00}0_2$],**	miss
0	**[$\underline{0}\underline{00}0_2$]**	miss

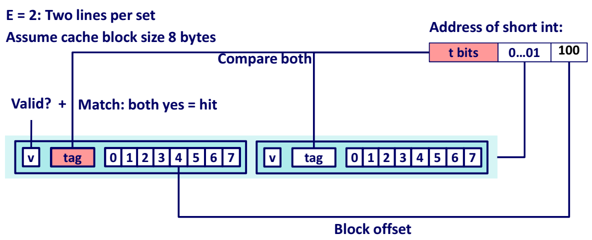
	v	Tag	Block
Set 0	1	0	M[0-1]
Set 1			
Set 2			
Set 3	1	0	M[6-7]

E-way Set-Associative Cache (Here: E = 2)

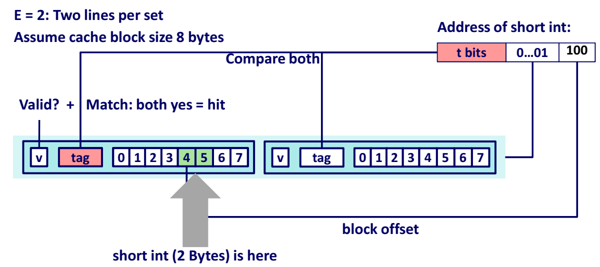
E = 2: Two lines per set
Assume cache block size 8 bytes



E-way Set-Associative Cache (Here: E = 2)



E-way Set-Associative Cache (Here: E = 2)



No match:

- One line in set is selected for eviction and replacement
- Replacement policies: random, least recently used (LRU), …

2-Way Set-Associative Cache Simulation

t=2	s=1	b=1
xx	x	x

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

Address trace (reads, one byte per read):

0	[$0\underline{0}0_2$],	miss
1	[$0\underline{0}1_2$],	hit
7	[$0\underline{1}1_2$],	miss
8	[$1\underline{0}0_2$],	miss
0	[$0\underline{0}0_2$]	hit

	v	Tag	Block
Set 0	1	00	M[0-1]
	1	10	M[8-9]
Set 1	1	01	M[6-7]
	0		

What About Writes?

Multiple copies of data exist:

- L1, L2, L3, Main Memory, Disk

What to do on a write hit?

- Write-through (write immediately to memory)
- Write-back (defer write to memory until replacement of line)
  - Need a "dirty" bit (line different from memory or not)

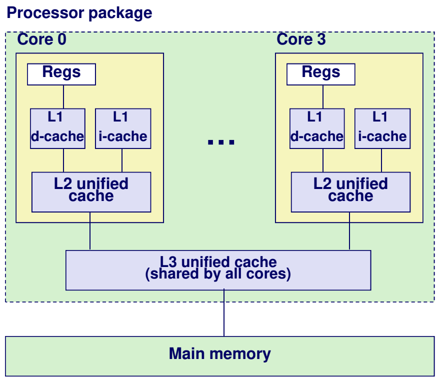
What to do on a write miss?

- Write-allocate (load into cache, update line in cache)
  - Good if more writes to the location follow
- No-write-allocate (writes straight to memory, does not load into cache)

Typical

- Write-through + No-write-allocate
- Write-back + Write-allocate

Intel Core i7 Cache Hierarchy



L1 i-cache and d-cache:
32 KB, 8-way,
Access: 4 cycles

L2 unified cache:
256 KB, 8-way,
Access: 10 cycles

L3 unified cache:
8 MB, 16-way,
Access: 40-75 cycles

Block size: 64 bytes for all caches.

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 line is in the cache
- Typical numbers:
  - 4 clock cycles for L1
  - 10 clock cycles for L2

Miss Penalty

- Additional time required because of a miss
  - Typically 50-200 cycles for main memory

Let’s Think About Those Numbers

Huge difference between a hit and a miss

- Could be 100x, if just L1 and main memory

Would you believe 99% hits is twice as good as 97%?

- Consider:
  Cache hit time of 1 cycle
  Miss penalty of 100 cycles
- Average access time:
  97% hits: 1 cycle + 0.03 * 100 cycles = 4 cycles
  99% hits: 1 cycle + 0.01 * 100 cycles = 2 cycles

This is why “miss rate” is used instead of “hit rate”

Writing Cache-Friendly Code

Make the common case go fast

- Focus on the inner loops of the core functions

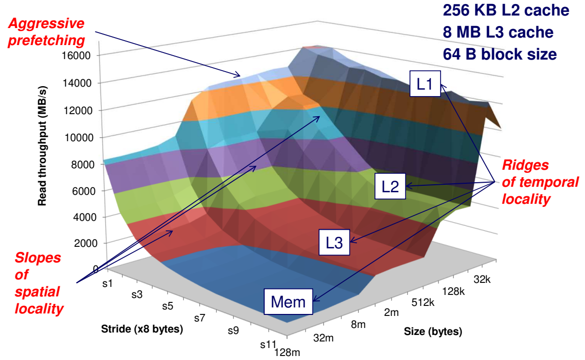
Minimize misses in the inner loops

- Repeated references to variables are good (temporal locality)
- Stride-1 reference patterns are good (spatial locality)

Key idea: Our qualitative notion of locality is quantified by our understanding of cache memories

The Memory Mountain

Core i7 Haswell
2.1 GHz
32 KB L1 d-cache
256 KB L2 cache
8 MB L3 cache
64 B block size



Matrix-Multiplication Example

Description:

- Multiply N x N matrices
- Matrix elements are doubles (8 bytes)
- $O(N^3)$ total operations
- N reads per source element
- N values summed per destination
  - But may be able to keep in register

Variable `sum` held in register

```
/* ijk */
for (i = 0; i < n; i++) {
  for (j = 0; j < n; j++) {
    sum = 0.0;
    for (k = 0; k < n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}
```

matmult/mm.c

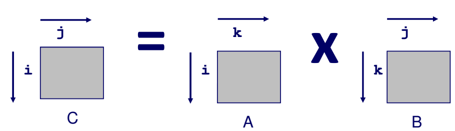
Miss-Rate Analysis for Matrix Multiply

Assume:

- Block size = 32B (big enough for four doubles)
- Matrix dimension (N) is very large
  - Approximate 1/N as 0.0
- Cache is not even big enough to hold multiple rows

Analysis Method:

- Look at access pattern of inner loop



Layout of C Arrays in Memory (review)

C arrays allocated in row-major order

- Each row in contiguous memory locations

Stepping through columns in one row:

- ```
  for (i = 0; i < N; i++)
    sum += a[0][i];
  ```
- Accesses successive elements
- If block size (B) > sizeof(a_{ij}) bytes, exploit spatial locality
  - Miss rate = sizeof(a_{ij}) / B

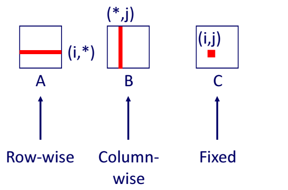
Stepping through rows in one column:

- ```
  for (i = 0; i < n; i++)
    sum += a[i][0];
  ```
- Accesses distant elements
- No spatial locality!
  - Miss rate = 1 (i.e. 100%)

Matrix Multiplication (ijk)

```
/* ijk */
for (i = 0; i < n; i++) {
  for (j = 0; j < n; j++) {
    sum = 0.0;
    for (k = 0; k < n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}
```
matmult/mm.c

Inner loop:



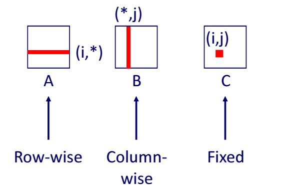
Misses per inner loop iteration:

A	B	C
0.25	1.0	0.0

Matrix Multiplication (jik)

```
/* jik */
for (j = 0; j < n; j++) {
  for (i = 0; i < n; i++) {
    sum = 0.0;
    for (k = 0; k < n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum
  }
}
```
matmult/mm.c

Inner loop:



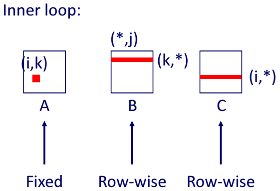
Misses per inner loop iteration:

A	B	C
0.25	1.0	0.0

Matrix Multiplication (kij)

```
/* kij */
for (k = 0; k < n; k++) {
  for (i = 0; i < n; i++) {
    r = a[i][k];
    for (j = 0; j < n; j++)
      c[i][j] += r * b[k][j];
  }
}
```
matmult/mm.c

Inner loop:



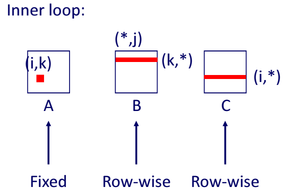
Misses per inner loop iteration:

A	B	C
0.0	0.25	0.25

Matrix Multiplication (ikj)

```
/* ikj */
for (i = 0; i < n; i++) {
  for (k = 0; k < n; k++) {
    r = a[i][k];
    for (j = 0; j < n; j++)
      c[i][j] += r * b[k][j];
  }
}
```
matmult/mm.c

Inner loop:

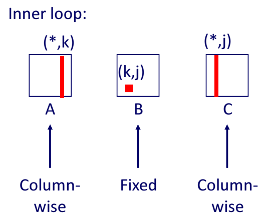


Misses per inner loop iteration:

A	B	C
0.0	0.25	0.25

Matrix Multiplication (jki)

```
/* jki */
for (j = 0; j < n; j++) {
  for (k = 0; k < n; k++) {
    r = b[k][j];
    for (i = 0; i < n; i++)
      c[i][j] += a[i][k] * r;
  }
}
                                    matmult/mm.c
```

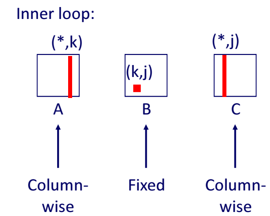


Misses per inner loop iteration:

A	B	C
1.0	0.0	1.0

Matrix Multiplication (kji)

```
/* kji */
for (k = 0; k < n; k++) {
  for (j = 0; j < n; j++) {
    r = b[k][j];
    for (i = 0; i < n; i++)
      c[i][j] += a[i][k] * r;
  }
}
                                    matmult/mm.c
```



Misses per inner loop iteration:

A	B	C
1.0	0.0	1.0

Summary of Matrix Multiplication

```
for (i = 0; i < n; i++) {
  for (j = 0; j < n; j++) {
    sum = 0.0;
    for (k = 0; k < n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}
```

ijk (& jik):

- 2 loads, 0 stores
- Misses/iter = **1.25**

```
for (k = 0; k < n; k++) {
  for (i = 0; i < n; i++) {
    r = a[i][k];
    for (j = 0; j < n; j++)
      c[i][j] += r * b[k][j];
  }
}
```

kij (& ikj):

- 2 loads, 1 store
- Misses/iter = **0.5**

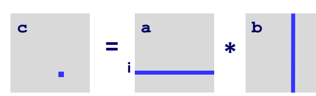
```
for (j = 0; j < n; j++) {
  for (k = 0; k < n; k++) {
    r = b[k][j];
    for (i = 0; i < n; i++)
      c[i][j] += a[i][k] * r;
  }
}
```

jki (& kji):

- 2 loads, 1 store
- Misses/iter = **2.0**

Better Matrix Multiplication

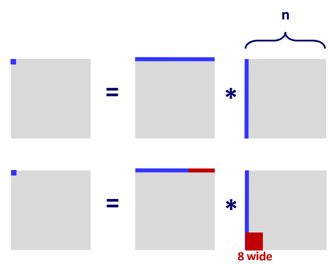
```
c = (double *) calloc(sizeof(double), n*n);

/* Multiply n x n matrices a and b  */
void mmm(double *a, double *b, double *c, int n) {
    int i, j, k;
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            for (k = 0; k < n; k++)
                c[i*n + j] += a[i*n + k] * b[k*n + j];
}
```



Cache Miss Analysis

Assume:
- Matrix elements are doubles
- Cache block = 8 doubles
- Cache size C << n (much smaller than n)

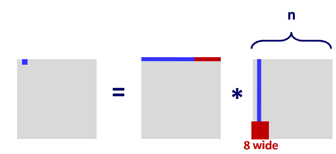
First iteration:
- n/8 + n = 9n/8 misses
- Afterwards in cache: (schematic)



Cache Miss Analysis

Assume:
- Matrix elements are doubles
- Cache block = 8 doubles
- Cache size C << n (much smaller than n)

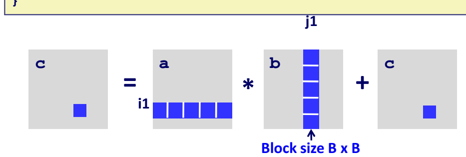
Second iteration:
- Again:
  n/8 + n = 9n/8 misses



Total misses:
- $9n/8 * n^2 = (9/8) * n^3$

Blocked Matrix Multiplication

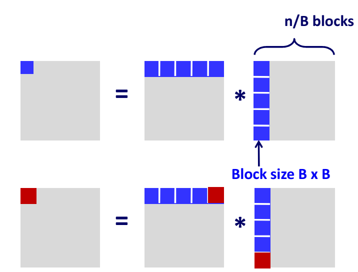
```
c = (double *) calloc(sizeof(double), n*n);

/* Multiply n x n matrices a and b  */
void mmm(double *a, double *b, double *c, int n) {
    int i, j, k;
    for (i = 0; i < n; i += B)
        for (j = 0; j < n; j += B)
            for (k = 0; k < n; k += B)
                /* B x B mini matrix multiplications */
                for (i1 = i; i1 < i+B; i++)
                    for (j1 = j; j1 < j+B; j++)
                        for (k1 = k; k1 < k+B; k++)
                            c[i1*n + j1] += a[i1*n + k1]*b[k1*n + j1];
}
                                                     matmult/bmm.c
```



Cache Miss Analysis

Assume:
- Cache block = 8 doubles
- Cache size C << n (much smaller than n)
- Three blocks ■ fit into cache: $3B^2 < C$

First (block) iteration:
- $B^2/8$ misses for each block
- $2n/B * B^2/8 = nB/4$
  (omitting matrix c)
- Afterwards in cache
  (schematic)



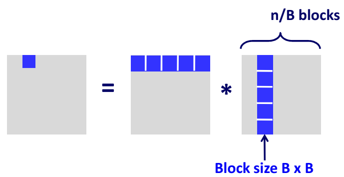
Cache Miss Analysis

Assume:

- Cache block = 8 doubles
- Cache size C << n (much smaller than n)
- Three blocks ■ fit into cache: $3B^2 < C$

Second (block) iteration:

- Same as first iteration
- $2n/B * B^2/8 = nB/4$



Total misses:

- $nB/4 * (n/B)^2 = n^3/(4B)$

Blocking Summary

No blocking: $(9/8) * n^3$

Blocking: $1/(4B) * n^3$

(plus $n^2/8$ misses for C)

Suggest largest possible block size B, but limit $3B^2 < C$!

Reason for dramatic difference:

- Matrix multiplication has inherent temporal locality:
  - Input data: $3n^2$, computation $2n^3$
  - Every array element used O(n) times!
- But program has to be written properly

Cache Summary

Cache memories can have significant performance impact

You can write your programs to exploit this!

- Focus on the inner loops, where bulk of computations and memory accesses occur.
- Try to maximize spatial locality by reading data objects with sequentially with stride 1.
- Try to maximize temporal locality by using a data object as often as possible once it's read from memory.