

15-213

"The course that gives CMU its Zip!"

Cache Memories October 6, 2006

Topics

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

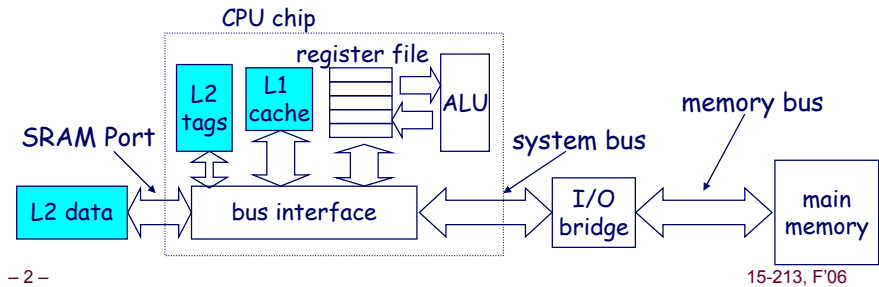
Cache Memories

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

- Hold frequently accessed blocks of main memory

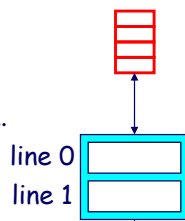
CPU looks first for data in L1, then in L2, then in main memory.

Typical system structure:



Inserting an L1 Cache Between the CPU and Main Memory

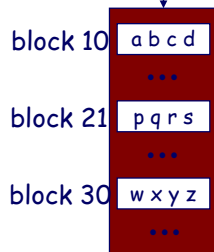
The transfer unit between the CPU register file and the cache is a 4-byte block.



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



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

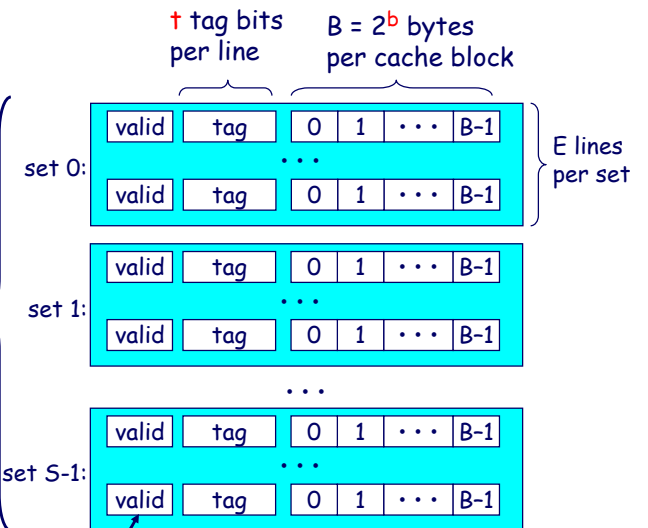
General Organization of a Cache

Cache is an array of sets.

Each set contains one or more lines.

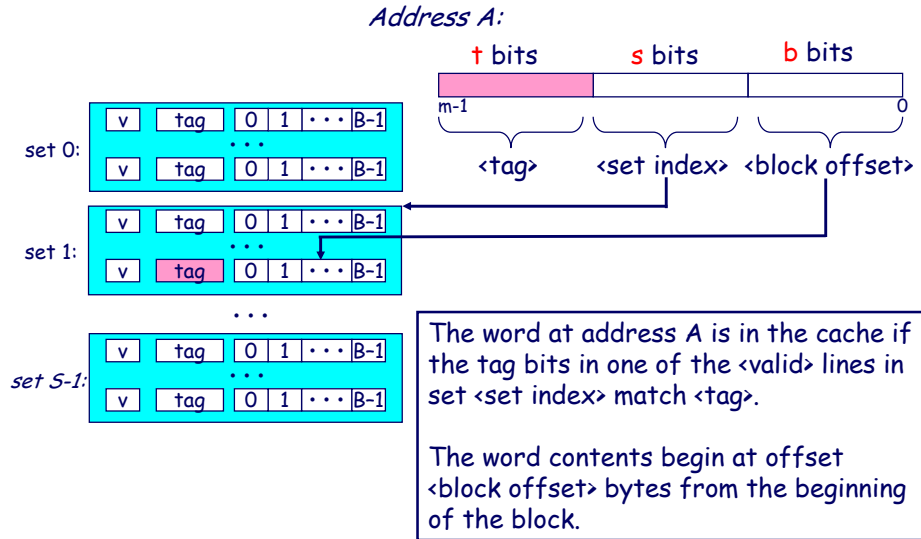
Each line holds a block of data.

$$S = 2^s \text{ sets}$$

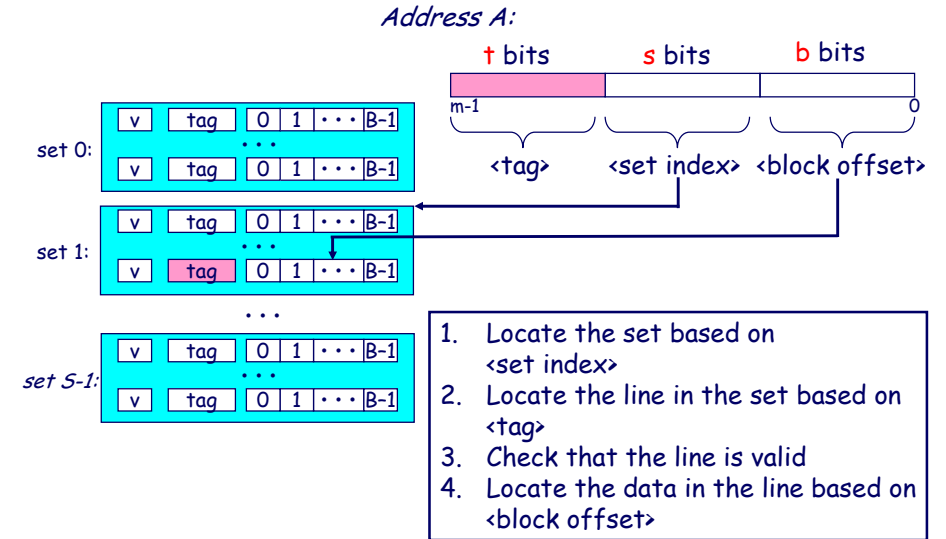


Cache size: $C = B \times E \times S$ data bytes

Addressing Caches



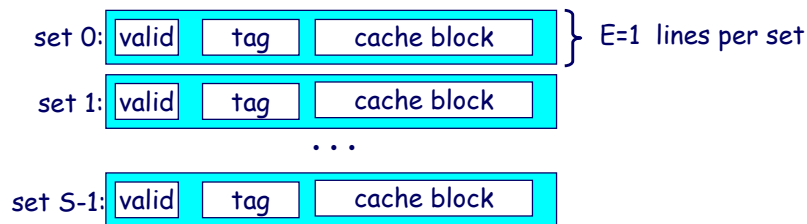
Addressing Caches



Direct-Mapped Cache

Simplest kind of cache, easy to build
(only 1 tag compare required per access)

Characterized by exactly one line per set.

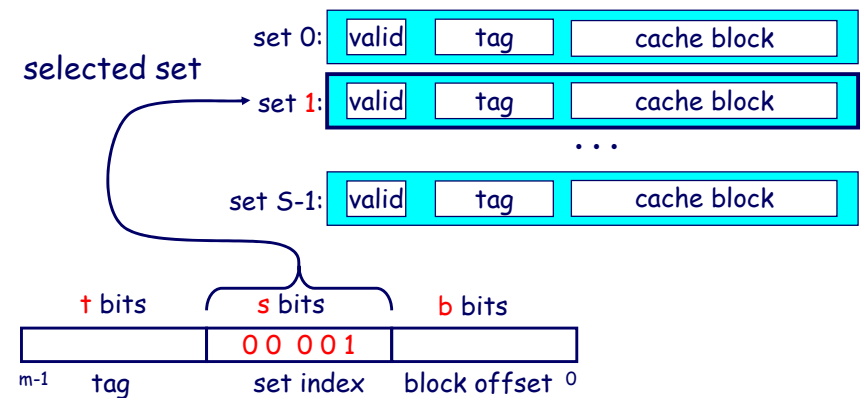


Cache size: $C = B \times S$ data bytes

Accessing Direct-Mapped Caches

Set selection

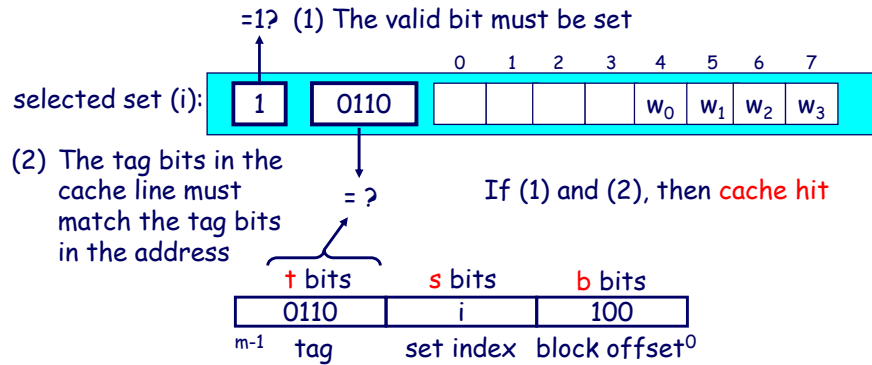
- Use the set index bits to determine the set of interest.



Accessing Direct-Mapped Caches

Line matching and word selection

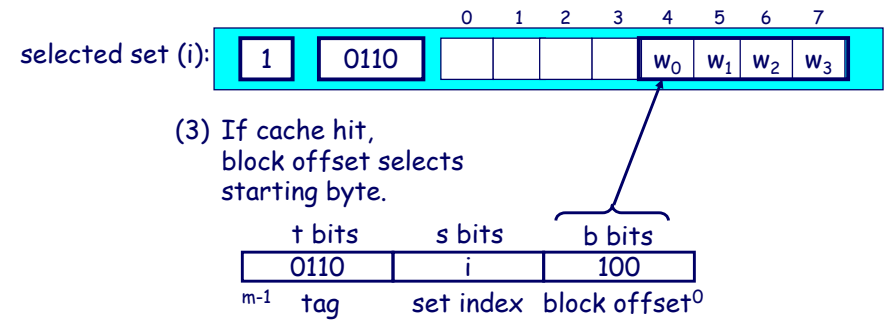
- **Line matching:** Find a valid line in the selected set with a matching tag
- **Word selection:** Then extract the word



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



Direct-Mapped Cache Simulation

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

$t=1$ $s=2$ $b=1$

| | | |
|---|----|---|
| x | xx | x |
|---|----|---|

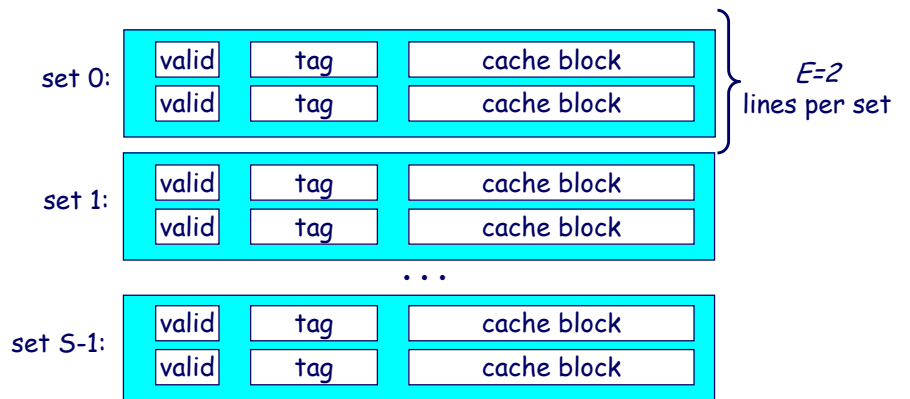
Address trace (reads):

| | | |
|---|-----------------------|------|
| 0 | [0000 ₂], | miss |
| 1 | [0001 ₂], | hit |
| 7 | [0111 ₂], | miss |
| 8 | [1000 ₂], | miss |
| 0 | [0000 ₂] | miss |

| v | tag | data |
|---|-----|----------|
| 1 | 0 | $M[0-1]$ |
| | | |
| | | |
| 1 | 0 | $M[6-7]$ |

Set Associative Caches

Characterized by more than one line per set

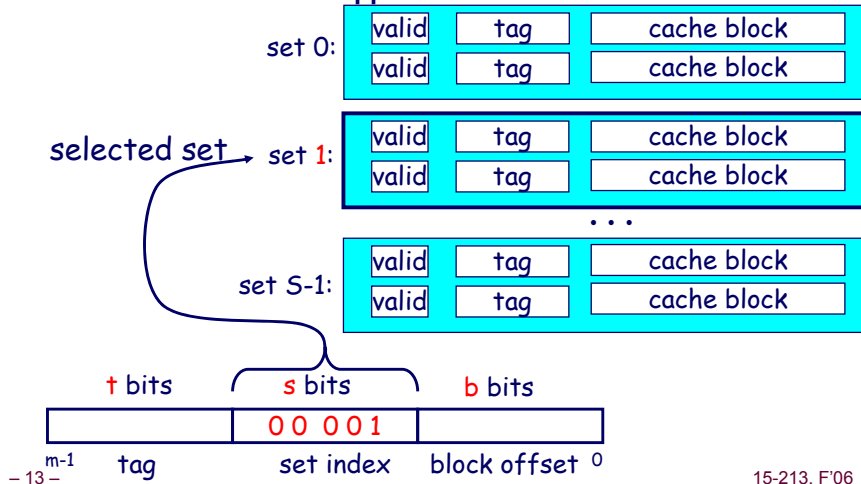


E-way associative cache

Accessing Set Associative Caches

Set selection

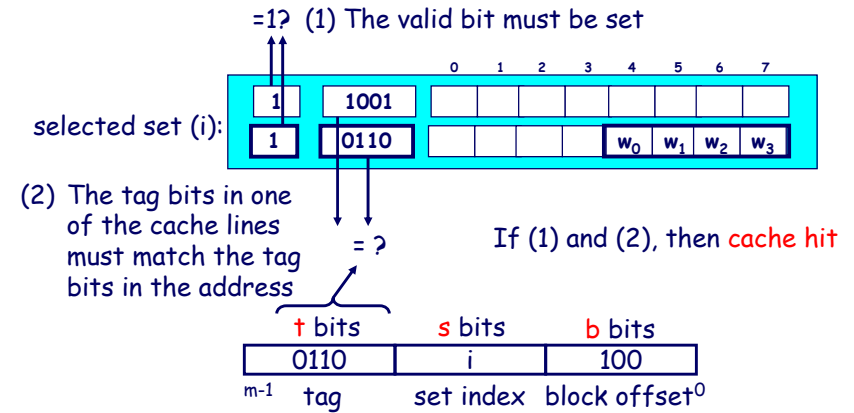
- identical to direct-mapped cache



Accessing Set Associative Caches

Line matching and word selection

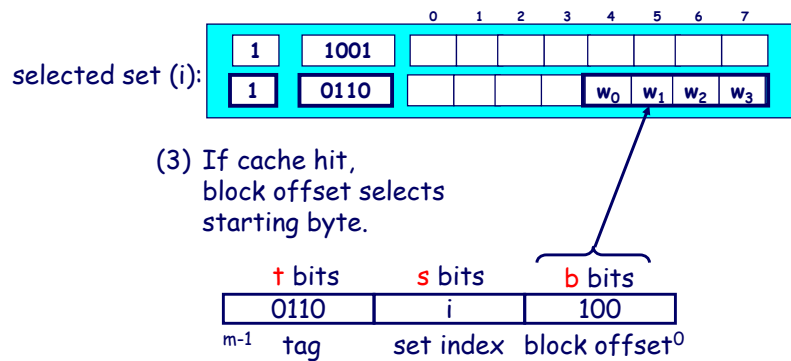
- must compare the tag in each valid line in the selected set.



Accessing Set Associative Caches

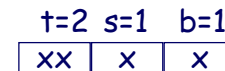
Line matching and word selection

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



2-Way Associative Cache Simulation

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

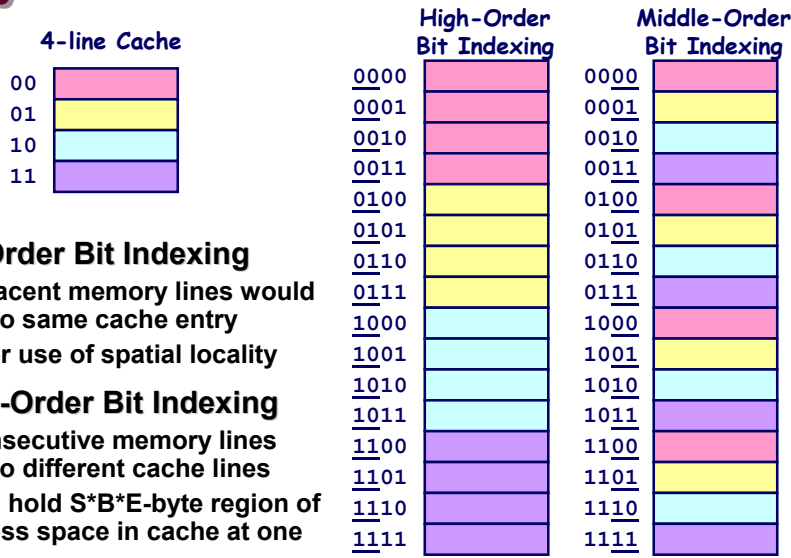


Address trace (reads):

| | | |
|---|----------------------|------|
| 0 | [0000 ₂] | miss |
| 1 | [0001 ₂] | hit |
| 7 | [0111 ₂] | miss |
| 8 | [1000 ₂] | miss |
| 0 | [0000 ₂] | hit |

| v | tag | data |
|---|-----|--------|
| 1 | 00 | M[0-1] |
| 1 | 10 | M[8-9] |
| 1 | 01 | M[6-7] |
| 0 | | |

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

- 17 -

15-213, F'06

Maintaining a Set-Associate Cache

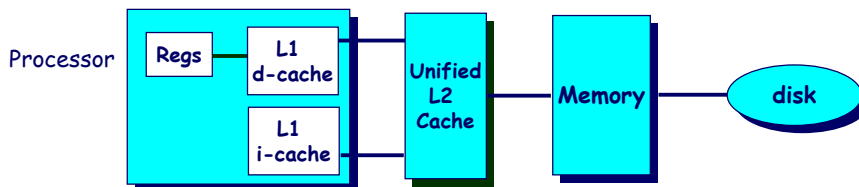
- How to decide which cache line to use in a set?
 - Least Recently Used (LRU), Requires $\lceil \lg_2(E!) \rceil$ extra bits
 - Not recently Used (NRU)
 - Random
- Virtual vs. Physical addresses:
 - The memory system works with physical addresses, but it takes time to translate a virtual to a physical address. So most L1 caches are virtually indexed, but physically tagged.

- 18 -

15-213, F'06

Multi-Level Caches

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



| | | | | | |
|------------|-------|---------|------------|-------------|-----------|
| 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



- 19 -

15-213, F'06

What about writes?

Multiple copies of data exist:

- L1
- L2
- Main Memory
- Disk

What to do when we write?

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

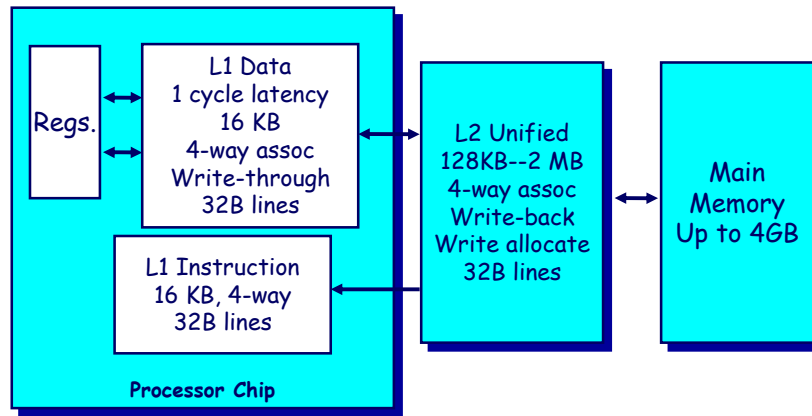
What to do on a replacement?

- Depends on whether it is write through or write back

- 20 -

15-213, F'06

Intel Pentium III Cache Hierarchy



- 21 -

15-213, F'06

Cache Performance Metrics

Miss Rate

- Fraction of memory references not found in cache (misses / references)
- Typical numbers:
 - 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

Aside for architects:
 -Increasing cache size?
 -Increasing block size?
 -Increasing associativity?

Miss Penalty

- Additional time required because of a miss
 - Typically 50-200 cycles for main memory (Trend: increasing!)

- 22 -

15-213, F'06

Writing Cache Friendly Code

- Repeated references to variables are good (**temporal locality**)
- Stride-1 reference patterns are good (**spatial locality**)
- Examples:
 - cold cache, 4-byte words, 4-word cache blocks

```
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;
}
```

Miss rate = 1/4 = 25%

```
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;
}
```

Miss rate = 100%

- 23 -

15-213, F'06

The Memory Mountain

Read throughput (read bandwidth)

- Number of bytes read from memory per second (MB/s)

Memory mountain

- Measured read throughput as a function of spatial and temporal locality.
- Compact way to characterize memory system performance.

- 24 -

15-213, F'06

Memory Mountain Test Function

```

/* The test function */
void test(int elems, int stride) {
    int i, result = 0;
    volatile int sink;

    for (i = 0; i < elems; i += stride)
        result += data[i];
    sink = result; /* So compiler doesn't optimize away the loop */
}

/* Run test(elems, stride) and return read throughput (MB/s) */
double run(int size, int stride, double Mhz)
{
    double cycles;
    int elems = size / sizeof(int);

    test(elems, stride); /* warm up the cache */
    cycles = fcy2(test, elems, stride, 0); /* call test(elems, stride) */
    return (size / stride) / (cycles / Mhz); /* convert cycles to MB/s */
}

```

Memory Mountain Main Routine

```

/* mountain.c - Generate the memory mountain. */
#define MINBYTES (1 << 10) /* Working set size ranges from 1 KB */
#define MAXBYTES (1 << 23) /* ... up to 8 MB */
#define MAXSTRIDE 16 /* Strides range from 1 to 16 */
#define MAXELEMS MAXBYTES/sizeof(int)

int data[MAXELEMS]; /* The array we'll be traversing */

int main()
{
    int size; /* Working set size (in bytes) */
    int stride; /* Stride (in array elements) */
    double Mhz; /* Clock frequency */

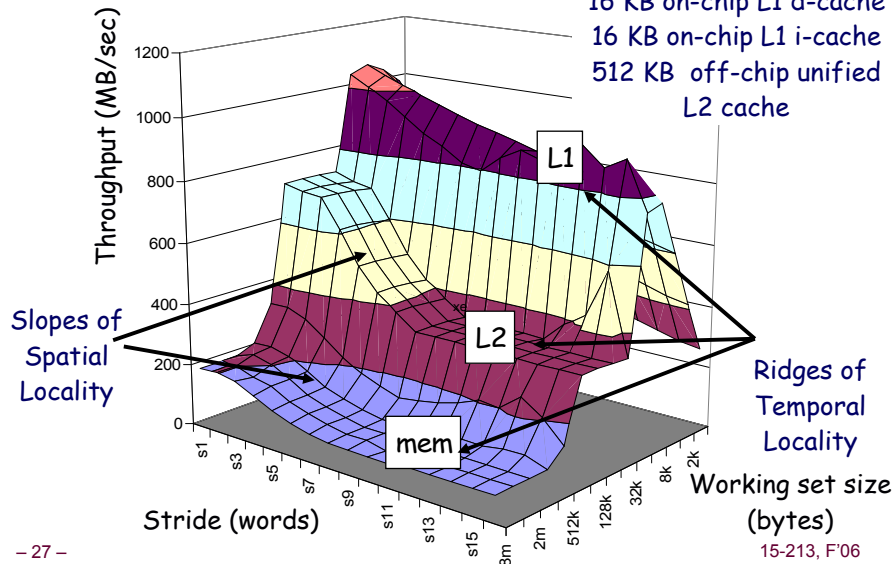
    init_data(data, MAXELEMS); /* Initialize each element in data to 1 */
    Mhz = mhz(0); /* Estimate the clock frequency */
    for (size = MAXBYTES; size >= MINBYTES; size >= 1) {
        for (stride = 1; stride <= MAXSTRIDE; stride++)
            printf("%.1f\t", run(size, stride, Mhz));
        printf("\n");
    }
    exit(0);
}

```

The Memory Mountain

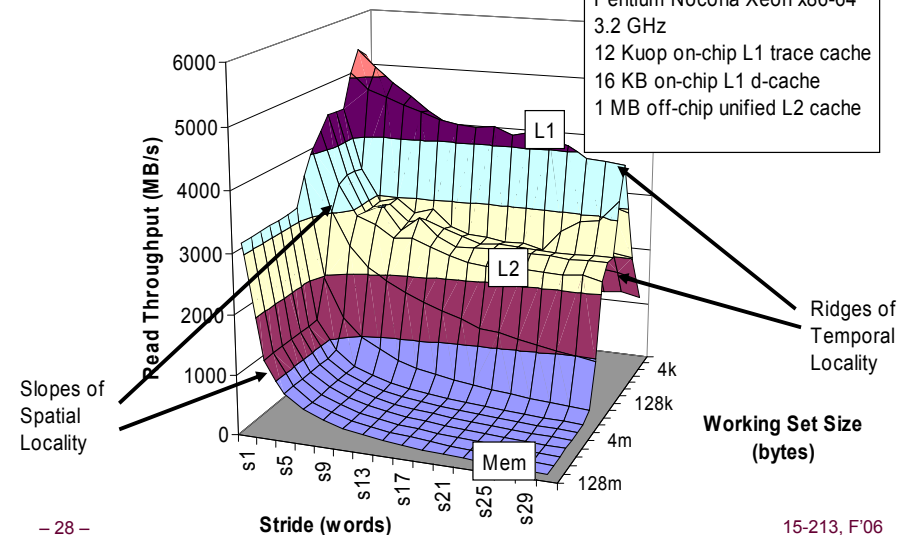
Pentium III
550 MHz

16 KB on-chip L1 d-cache
16 KB on-chip L1 i-cache
512 KB off-chip unified
L2 cache

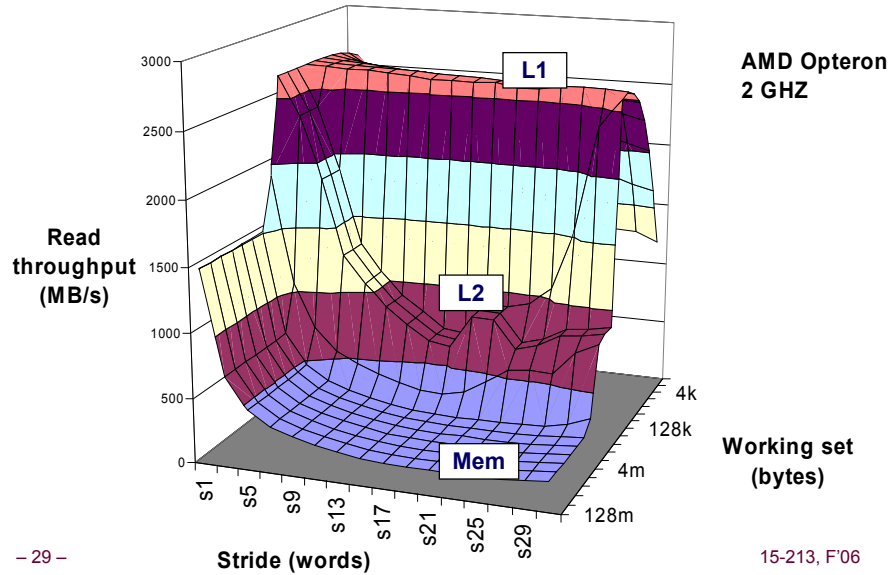


X86-64 Memory Mountain

Pentium Nocona Xeon x86-64
3.2 GHz
12 Kuop on-chip L1 trace cache
16 KB on-chip L1 d-cache
1 MB off-chip unified L2 cache



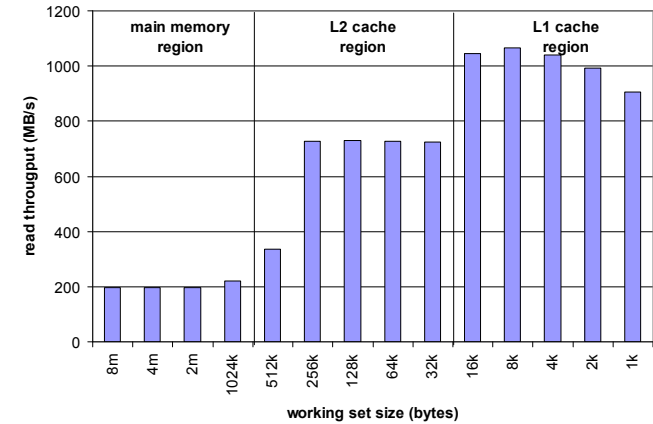
Opton Memory Mountain



Ridges of Temporal Locality

Slice through the memory mountain with stride=1

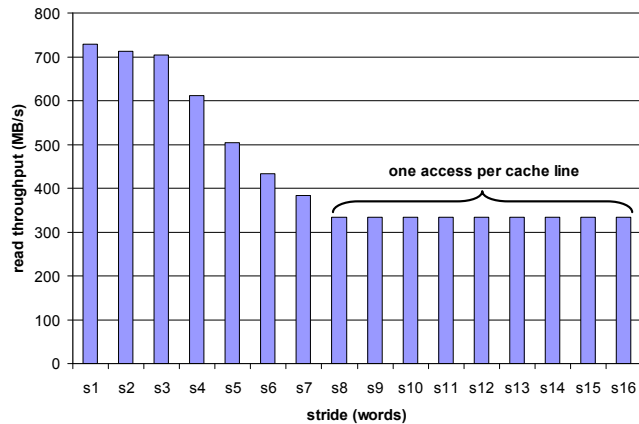
- illuminates read throughputs of different caches and memory



A Slope of Spatial Locality

Slice through memory mountain with size=256KB

- shows cache block size.



Matrix Multiplication Example

Major Cache Effects to Consider

- Total cache size
 - Exploit temporal locality and keep the working set small (e.g., use blocking)
- Block size
 - Exploit spatial locality

```

/* 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;
  }
}
    
```

Variable sum held in register

Description:

- Multiply $N \times N$ matrices
 - $O(N^3)$ total operations
 - Accesses
 - N reads per source element
 - N values summed per destination
- » but may be able to hold in register

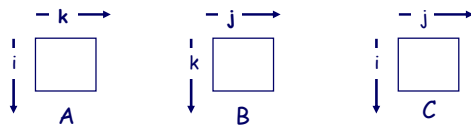
Miss Rate Analysis for Matrix Multiply

Assume:

- Line size = 32B (big enough for four 64-bit words)
- 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) > 4 bytes, exploit spatial locality
 - compulsory miss rate = 4 bytes / B

Stepping through rows in one column:

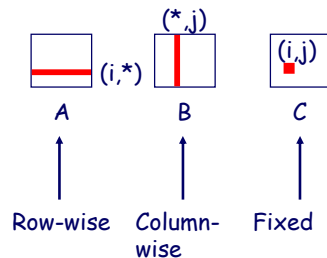
- for (i = 0; i < n; i++)
 - sum += a[i][0];
- accesses distant elements
- no spatial locality!
 - compulsory miss rate = 1 (i.e. 100%)

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;
  }
}
    
```

Inner loop:



Misses per Inner Loop Iteration:

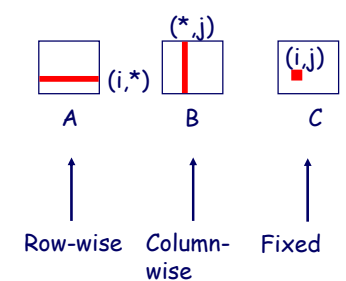
| A | B | C |
|------|-----|-----|
| 0.25 | 1.0 | 0.0 |

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;
  }
}
    
```

Inner loop:



Misses per Inner Loop Iteration:

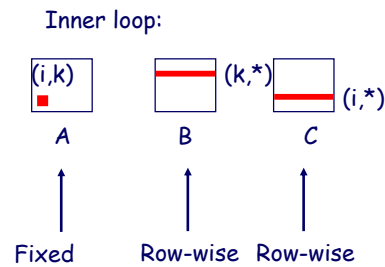
| A | B | C |
|------|-----|-----|
| 0.25 | 1.0 | 0.0 |

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];
  }
}

```



Misses per Inner Loop Iteration:

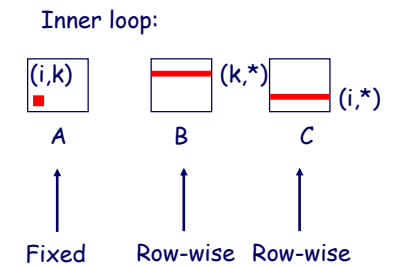
| | | |
|-----------------|------------------|------------------|
| $\frac{A}{0.0}$ | $\frac{B}{0.25}$ | $\frac{C}{0.25}$ |
|-----------------|------------------|------------------|

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];
  }
}

```



Misses per Inner Loop Iteration:

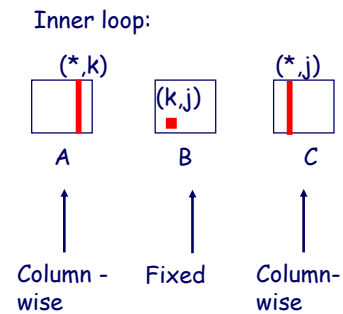
| | | |
|-----------------|------------------|------------------|
| $\frac{A}{0.0}$ | $\frac{B}{0.25}$ | $\frac{C}{0.25}$ |
|-----------------|------------------|------------------|

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;
  }
}

```



Misses per Inner Loop Iteration:

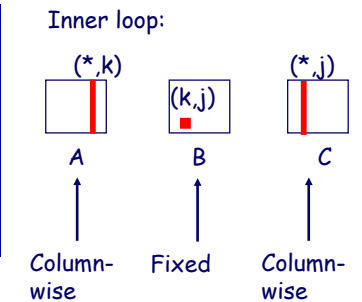
| | | |
|-----------------|-----------------|-----------------|
| $\frac{A}{1.0}$ | $\frac{B}{0.0}$ | $\frac{C}{1.0}$ |
|-----------------|-----------------|-----------------|

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;
  }
}

```



Misses per Inner Loop Iteration:

| | | |
|-----------------|-----------------|-----------------|
| $\frac{A}{1.0}$ | $\frac{B}{0.0}$ | $\frac{C}{1.0}$ |
|-----------------|-----------------|-----------------|

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

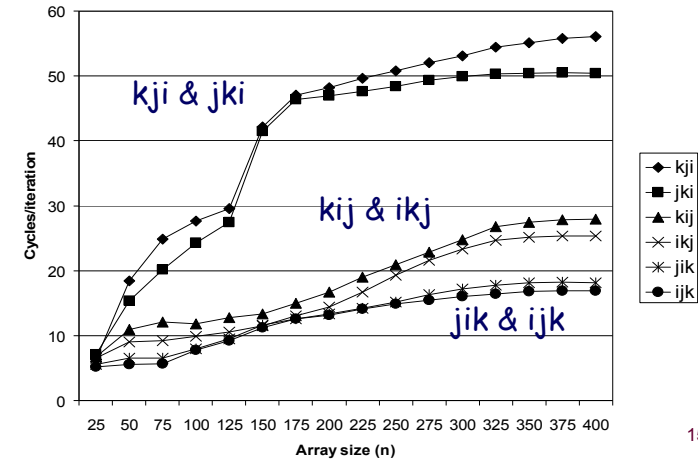
- 41 -

15-213, F'06

Pentium Matrix Multiply Performance

Miss rates are helpful but not perfect predictors.

- Code scheduling matters, too.



- 42 -

15-213, F'06

Improving Temporal Locality by Blocking

Example: Blocked matrix multiplication

- “block” (in this context) does not mean “cache block”.
- Instead, it mean a sub-block within the matrix.
- Example: N = 8; sub-block size = 4

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

Key idea: Sub-blocks (i.e., A_{xy}) can be treated just like scalars.

$$C_{11} = A_{11}B_{11} + A_{12}B_{21} \quad C_{12} = A_{11}B_{12} + A_{12}B_{22}$$

$$C_{21} = A_{21}B_{11} + A_{22}B_{21} \quad C_{22} = A_{21}B_{12} + A_{22}B_{22}$$

- 43 -

15-213, F'06

Blocked Matrix Multiply (bijk)

```
for (jj=0; jj<n; jj+=bsize) {
  for (i=0; i<n; i++)
    for (j=jj; j < min(jj+bsize,n); j++)
      c[i][j] = 0.0;

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

- 44 -

15-213, F'06

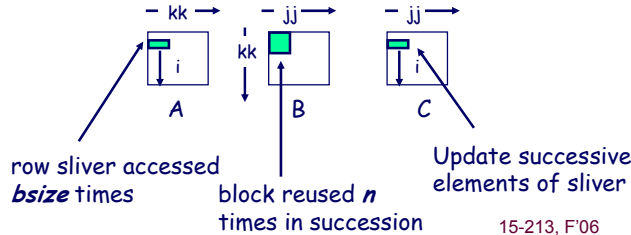
Blocked Matrix Multiply Analysis

- Innermost loop pair multiplies a $1 \times bsize$ sliver of A by a $bsize \times bsize$ block of B and accumulates into $1 \times bsize$ sliver of C
- Loop over i steps through n row slivers of A & C , using same B

```

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

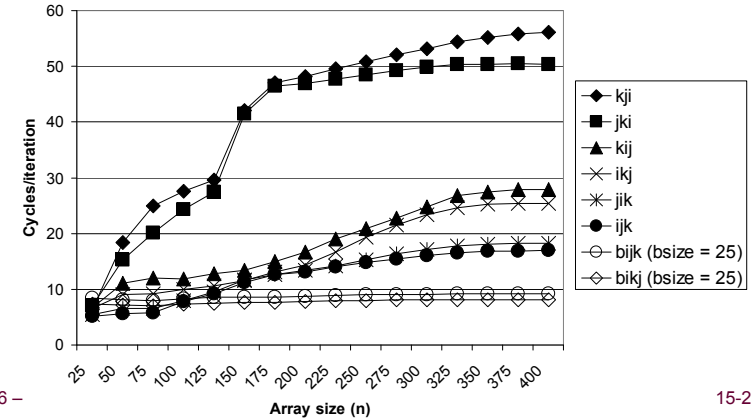
Innermost Loop Pair



Pentium Blocked Matrix Multiply Performance

Blocking (bijk and bikj) improves performance by a factor of two over unblocked versions (ijk and jik)

- relatively insensitive to array size.



Concluding Observations

Programmer can optimize for cache performance

- How data structures are organized
- How data are accessed
 - Nested loop structure
 - Blocking is a general technique

All systems favor "cache friendly code"

- Getting absolute optimum performance is very platform specific
 - Cache sizes, line sizes, associativities, etc.
- Can get most of the advantage with generic code
 - Keep working set reasonably small (temporal locality)
 - Use small strides (spatial locality)