

15-213

"The course that gives CMU its Zip!"

Cache Memories

Oct 11, 2001

Topics

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

class14.ppt

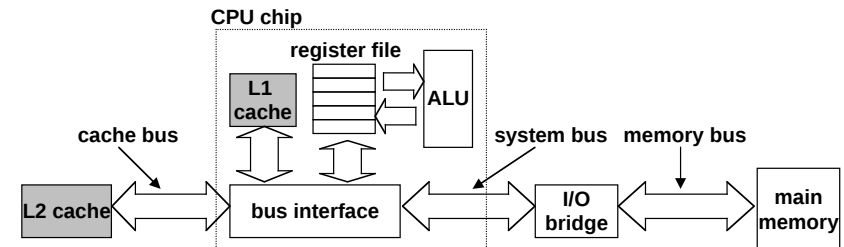
Cache memories

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 bus structure:

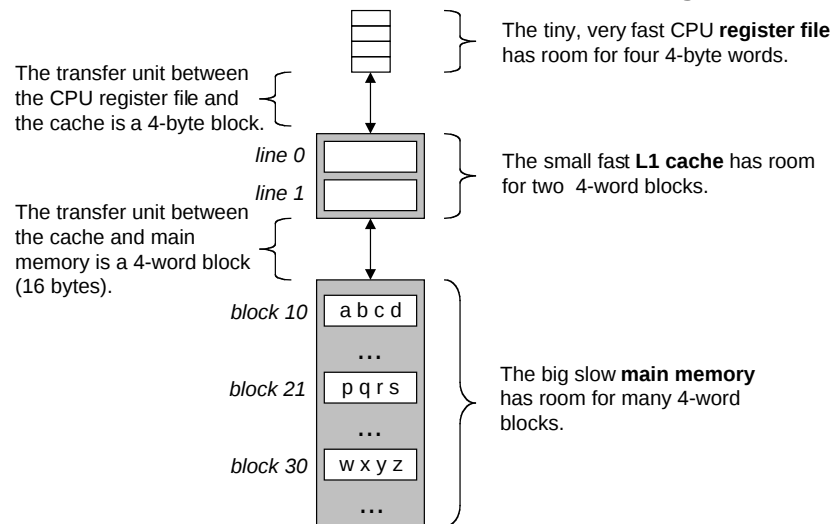


class14.ppt

- 2 -

CS 213 F'01

Inserting an L1 cache between the CPU and main memory

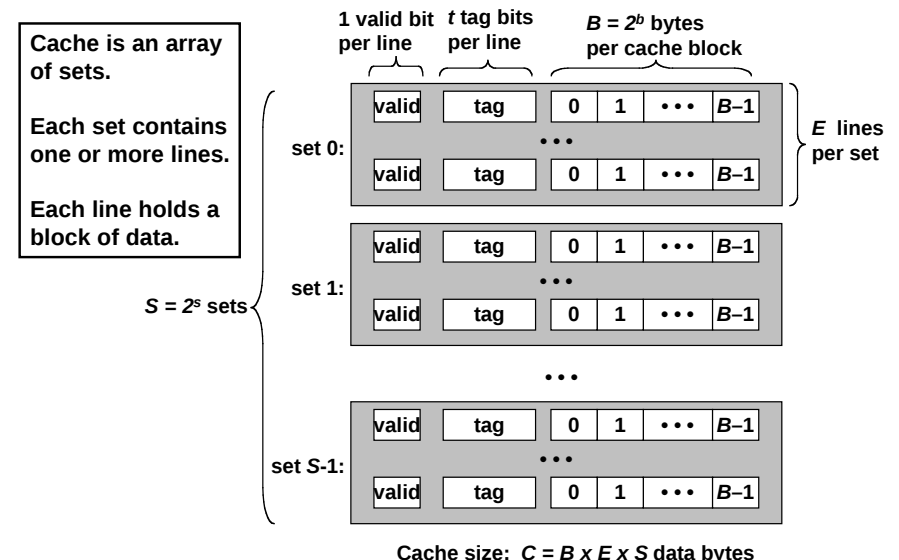


class14.ppt

- 3 -

CS 213 F'01

General organization of a cache memory

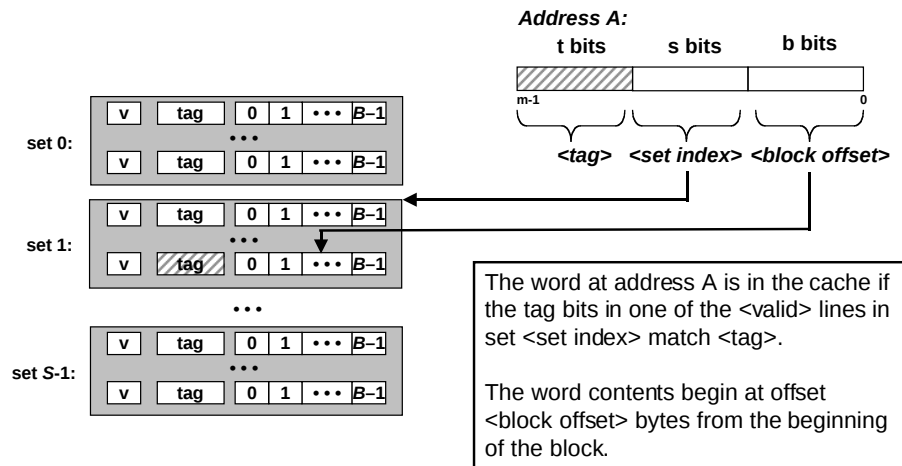


class14.ppt

- 4 -

CS 213 F'01

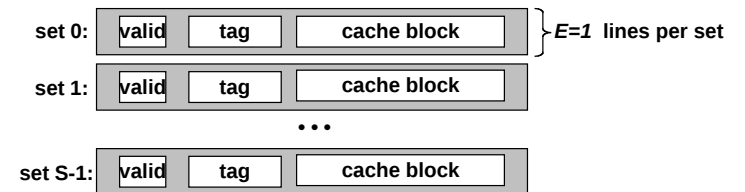
Addressing caches



Direct-mapped cache

Simplest kind of cache

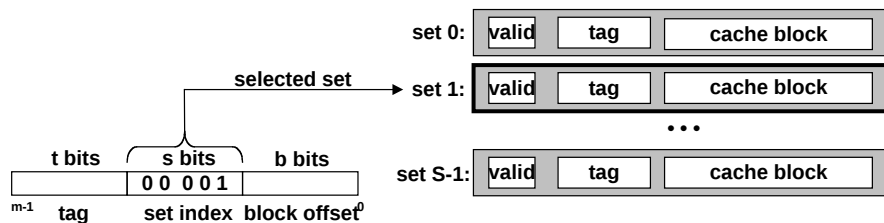
Characterized by exactly one line per set.



Accessing direct-mapped caches

Set selection

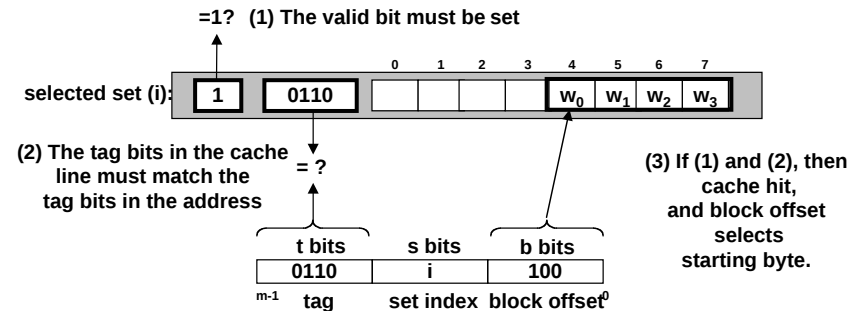
- Use the set index bits to determine the set of interest.



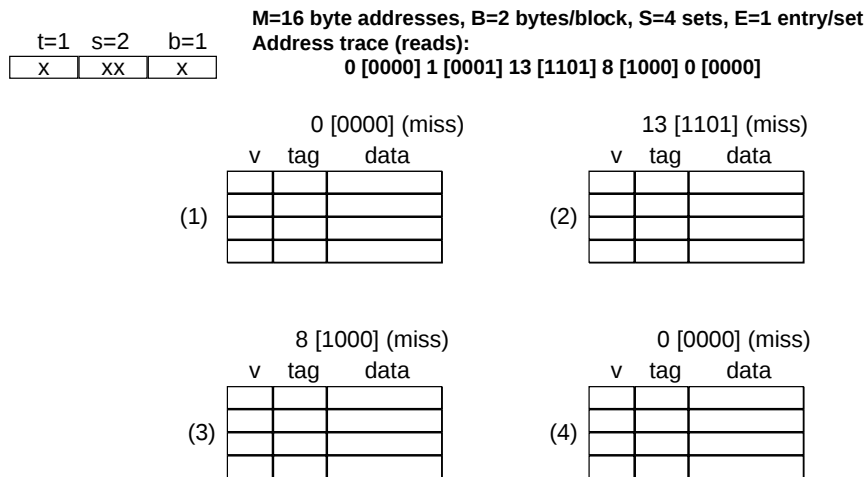
Accessing direct-mapped caches

Line matching and word selection

- find a valid line in the selected set with a matching tag (line matching)
- then extract the word (word selection)



Direct-mapped cache simulation

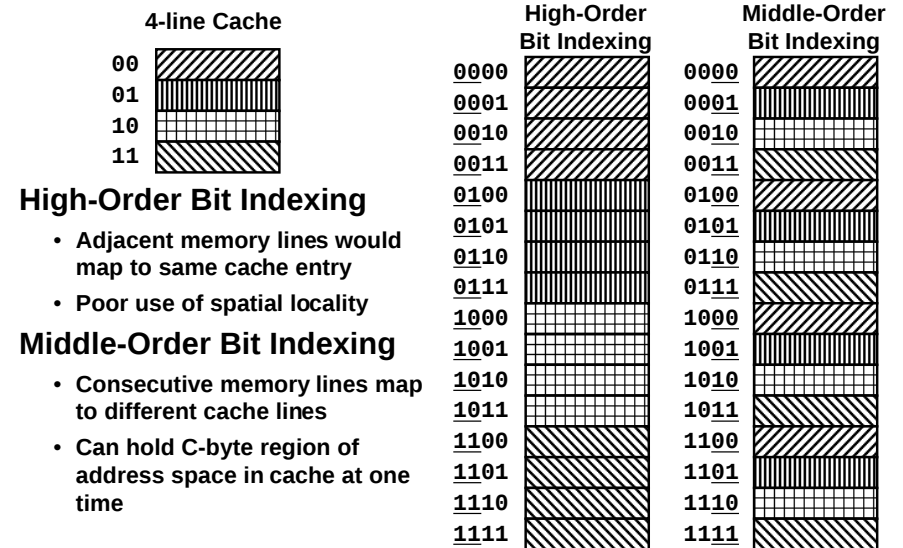


class14.ppt

- 9 -

CS 213 F'01

Why use middle bits as index?



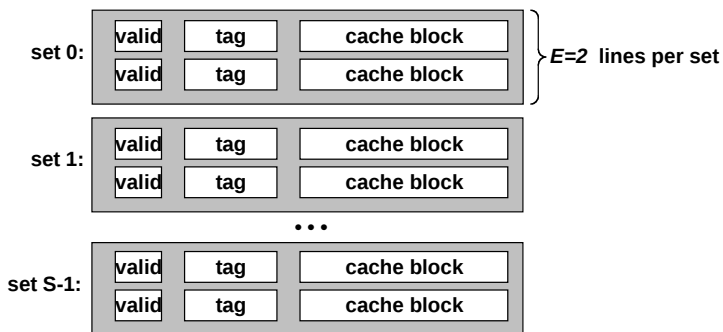
class14.ppt

- 10 -

CS 213 F'01

Set associative caches

Characterized by more than one line per set



class14.ppt

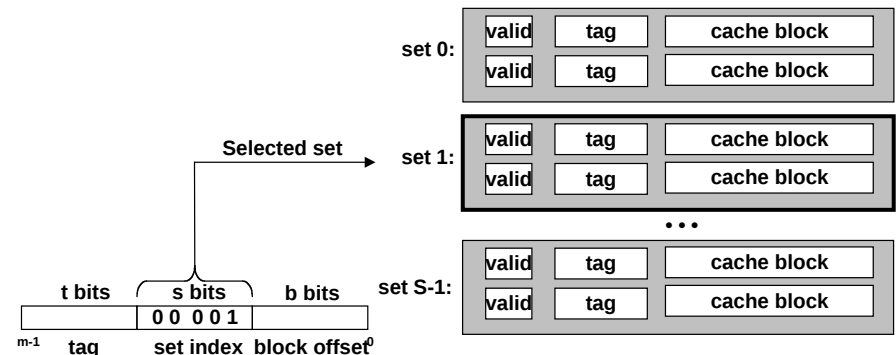
- 11 -

CS 213 F'01

Accessing set associative caches

Set selection

- identical to direct-mapped cache



class14.ppt

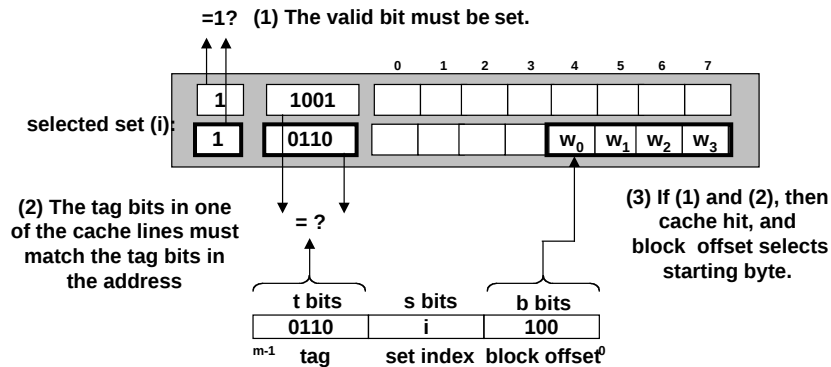
- 12 -

CS 213 F'01

Accessing set associative caches

Line matching and word selection

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



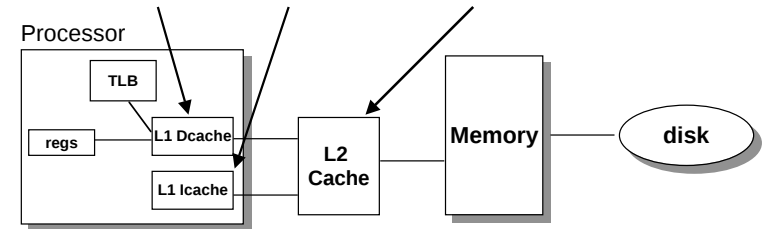
class14.ppt

- 13 -

CS 213 F'01

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

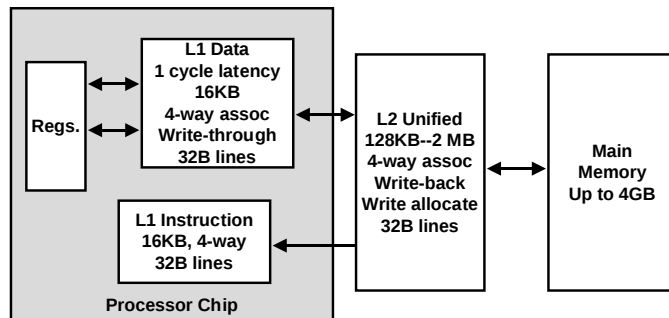
larger line size, higher associativity, more likely to write back

class14.ppt

- 14 -

CS 213 F'01

Intel Pentium cache hierarchy



class14.ppt

- 15 -

CS 213 F'01

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 clock cycle for L1
 - 3-8 clock cycles for L2

Miss Penalty

- additional time required because of a miss
 - Typically 25-100 cycles for main memory

class14.ppt

- 16 -

CS 213 F'01

Writing cache friendly code

Repeated references to variables are good (temporal locality)

Stride-1 reference patterns are good (spatial locality)

Example

- cold cache, 4-byte words, 4-word cache blocks

```
int sumarrayrows(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;
}
```

```
int sumarraycols(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 =

class14.ppt

- 17 -

Miss rate =

CS 213 F'01

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.

class14.ppt

- 18 -

CS 213 F'01

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 = fcyc2(test, elems, stride, 0); /* call test(elems, stride) */
    return (size / stride) / (cycles / Mhz); /* convert cycles to MB/s */
}
```

class14.ppt

- 19 -

CS 213 F'01

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 */

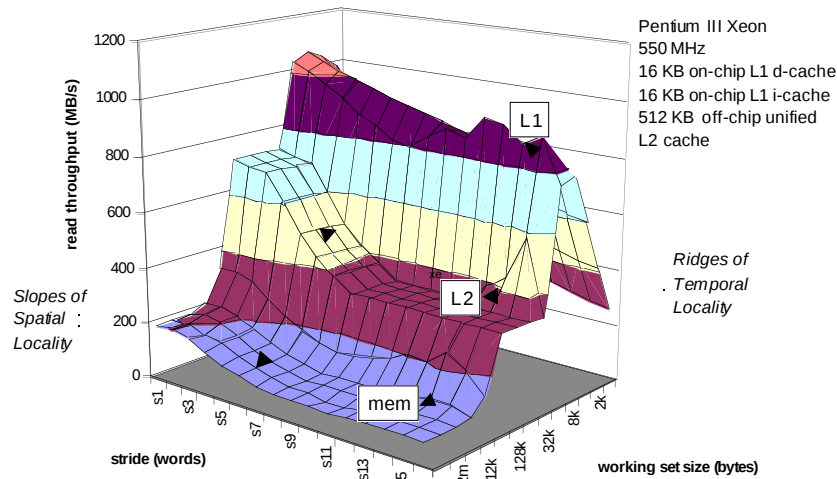
    init_data(data, MAXELEMS); /* Initialize each element in data to 1 */
    Mhz = mhz(0); /* Estimate the clock frequency */
    for (size = MAXBYTES; size >= MINBYTES; size >>= 1) {
        for (stride = 1; stride <= MAXSTRIDE; stride++)
            printf("%.1f\t", run(size, stride, Mhz));
        printf("\n");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

class14.ppt

- 20 -

CS 213 F'01

The Memory Mountain



class14.ppt

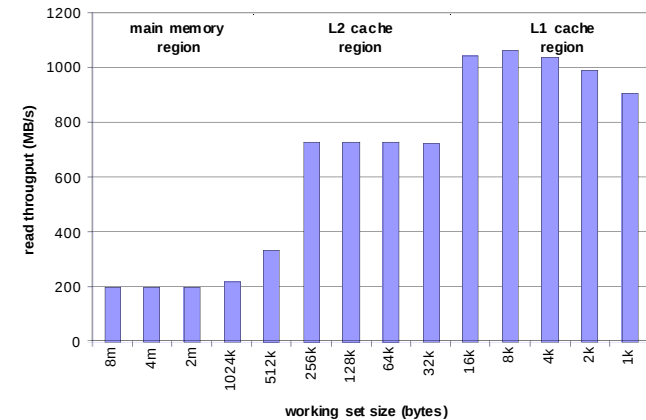
- 21 -

CS 213 F'01

Ridges of temporal locality

Slice through the memory mountain with stride=1

- illuminates read throughputs of different caches and memory



class14.ppt

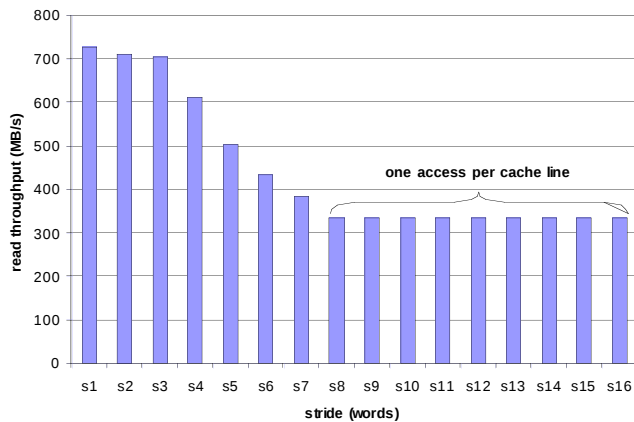
- 22 -

CS 213 F'01

A slope of spatial locality

Slice through memory mountain with size=256KB

- shows cache block size.



class14.ppt

- 23 -

CS 213 F'01

Matrix multiplication example

Major Cache Effects to Consider

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

- Block size
 - Exploit spatial locality

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

```
/* 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

class14.ppt

- 24 -

CS 213 F'01

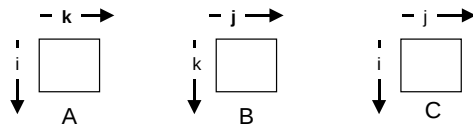
Miss rate analysis for matrix multiply

Assume:

- Line size = 32B (big enough for 4 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



class14.ppt

– 25 –

CS 213 F'01

Layout of arrays in memory

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%)

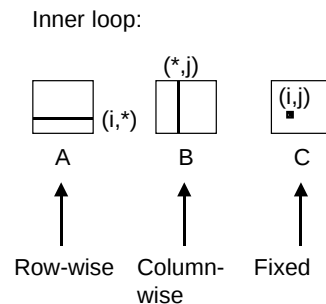
class14.ppt

– 26 –

CS 213 F'01

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;
    }
}
```



Misses per Inner Loop Iteration:

A	B	C
0.25	1.0	0.0

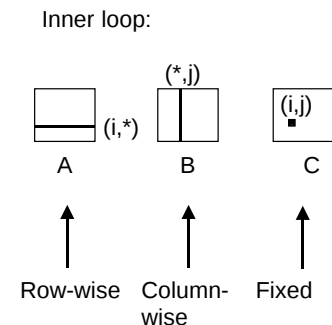
class14.ppt

– 27 –

CS 213 F'01

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;
    }
}
```



Misses per Inner Loop Iteration:

A	B	C
0.25	1.0	0.0

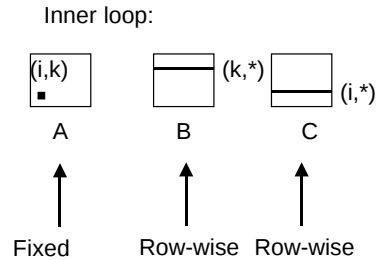
class14.ppt

– 28 –

CS 213 F'01

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];
  }
}
```

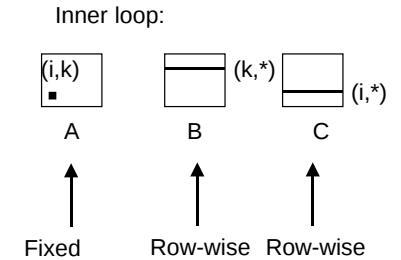


Misses per Inner Loop Iteration:

<u>A</u>	<u>B</u>	<u>C</u>
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];
  }
}
```

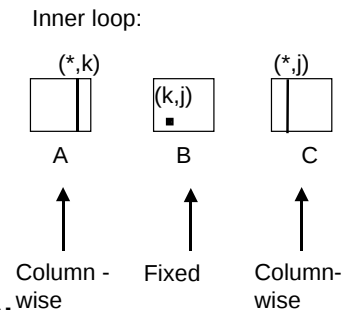


Misses per Inner Loop Iteration:

<u>A</u>	<u>B</u>	<u>C</u>
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;
  }
}
```

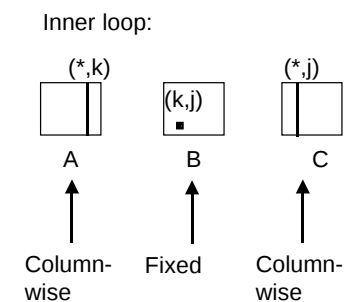


Misses per Inner Loop Iteration:

<u>A</u>	<u>B</u>	<u>C</u>
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;
  }
}
```



Misses per Inner Loop Iteration:

<u>A</u>	<u>B</u>	<u>C</u>
1.0	0.0	1.0

Summary of matrix multiplication

ijk (& jik):

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

kij (& ikj):

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

jki (& kji):

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

```
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;
  }
}
```

```
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];
  }
}
```

```
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;
  }
}
```

class14.ppt

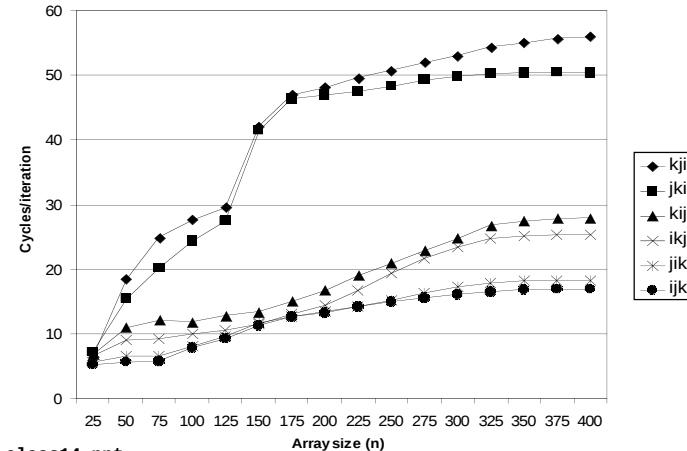
- 33 -

CS 213 F'01

Pentium matrix multiply performance

Notice that miss rates are helpful but not perfect predictors.

– Code scheduling matters, too.



class14.ppt

- 34 -

CS 213 F'01

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.

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

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

class14.ppt

- 35 -

CS 213 F'01

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;
      }
    }
  }
}
```

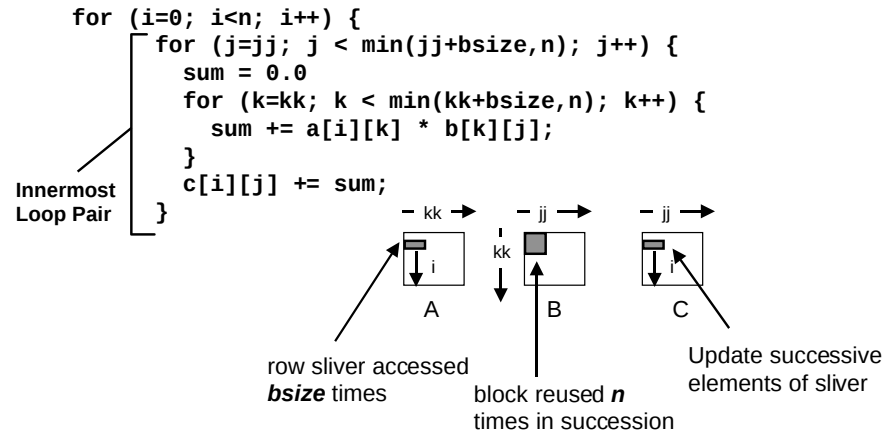
class14.ppt

- 36 -

CS 213 F'01

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



class14.ppt

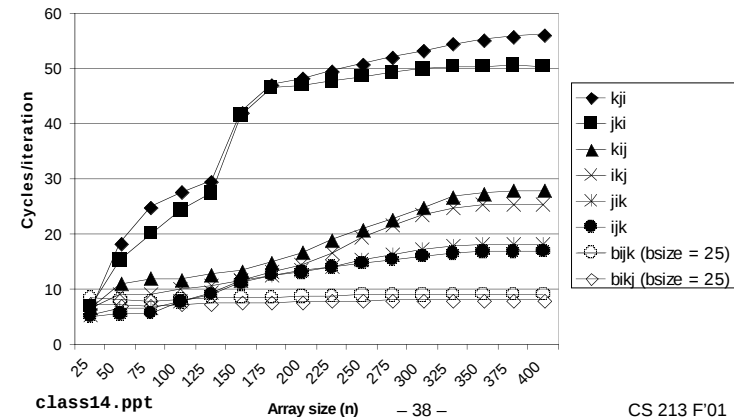
- 37 -

CS 213 F'01

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 accessed
 - Nested loop structure
 - Blocking (see text) is a general technique

All machines like “cache friendly code”

- Getting absolute optimum performance 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)

class14.ppt

- 39 -

CS 213 F'01