Memory Hierarchy

Computer Organization and Assembly Languages

Yung-Yu Chuang

2006/01/05

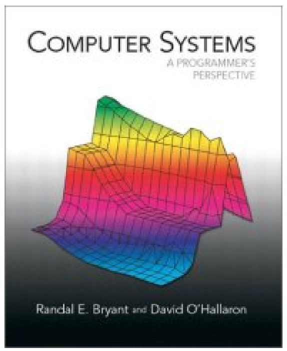
with slides by CMU15-213

Announcement

- Grade for hw#4 is online
- **Please DO submit homework if you haven't**
- Please sign up a demo time on 1/16 or 1/17 at the following page
  http://www.csie.ntu.edu.tw/~b90095/index.cgi/Assembly_Demo
- Hand in your report to TA at your demo time
- The length of report depends on your project type. It can be html, pdf, doc, ppt…

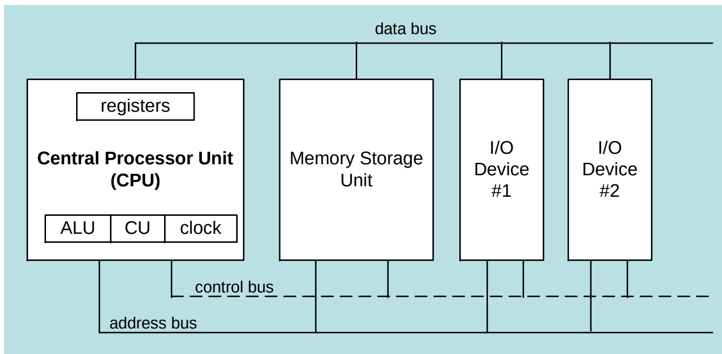
Reference

- Chapter 6 from "*Computer System: A Programmer's Perspective*"



Computer system model

- We assume memory is a linear array which holds both instruction and data, and CPU can access memory in a constant time.

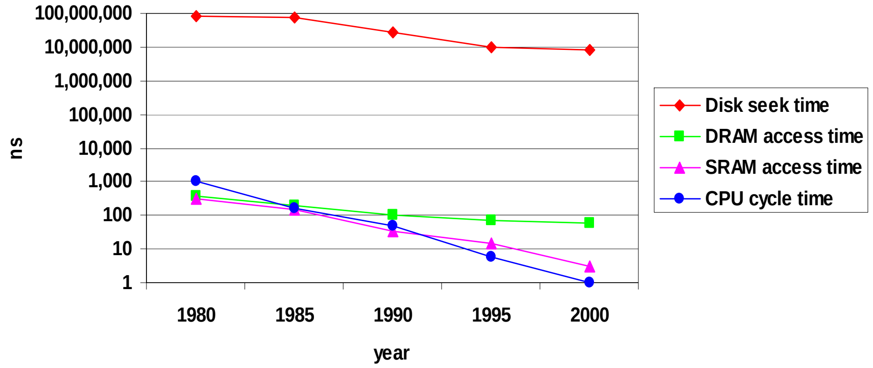


SRAM vs DRAM

	Tran. per bit	Access time	Needs refresh?	Cost	Applications
SRAM	4 or 6	1X	No	100X	cache memories
DRAM	1	10X	Yes	1X	Main memories, frame buffers

The CPU-Memory gap

The gap widens between DRAM, disk, and CPU speeds.

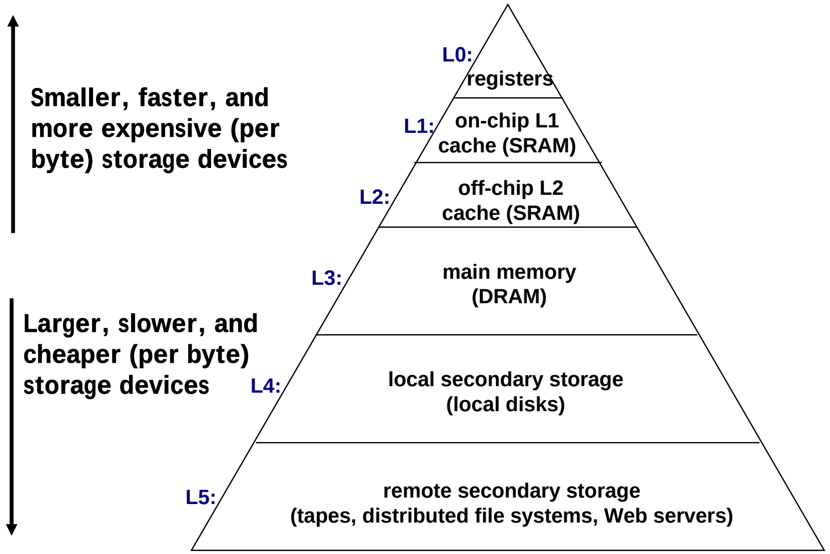


	register	cache	memory	disk
Access time (cycles)	1	1-10	50-100	20,000,000

Memory hierarchies

- Some fundamental and enduring properties of hardware and software:
  - Fast storage technologies cost more per byte, have less capacity, and require more power (heat!).
  - The gap between CPU and main memory speed is widening.
  - Well-written programs tend to exhibit good locality.
- They suggest an approach for organizing memory and storage systems known as a memory hierarchy.

Memory system in practice



Why it works?

- Most programs tend to access the storage at any particular level more frequently than the storage at the lower level.
- Locality: tend to access the same set of data items over and over again or tend to access sets of nearby data items.

Why learn it?

- A programmer needs to understand this because the memory hierarchy has a big impact on performance.
- You can optimize your program so that its data is more frequently stored in the higher level of the hierarchy.
- For example, the difference of running time for matrix multiplication could up to a factor of 6 even if the same amount of arithmetic instructions are performed.

Locality

- Principle of Locality: programs tend to reuse data and instructions near those they have used recently, or that were recently referenced themselves.
  - Temporal locality: recently referenced items are likely to be referenced in the near future.
  - Spatial locality: items with nearby addresses tend to be referenced close together in time.
- In general, *programs with good locality run faster then programs with poor locality*
- Locality is the reason why cache and virtual memory are designed in architecture and operating system. Another example is web browser caches recently visited webpages.

Locality example

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

- **Data**
  - Reference array elements in succession (stride-1 reference pattern): **Spatial locality**
  - Reference sum each iteration: **Temporal locality**
- **Instructions**
  - Reference instructions in sequence: **Spatial locality**
  - Cycle through loop repeatedly: **Temporal locality**

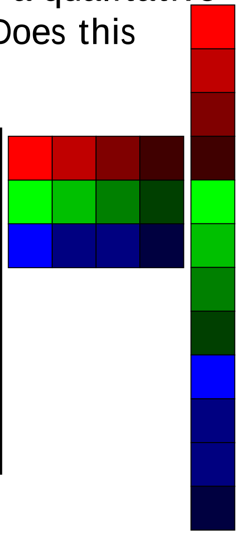
Locality example

- Being able to look at code and get a qualitative sense of its locality is important. Does this function have good locality?

```
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;
}
```

stride-1 reference pattern



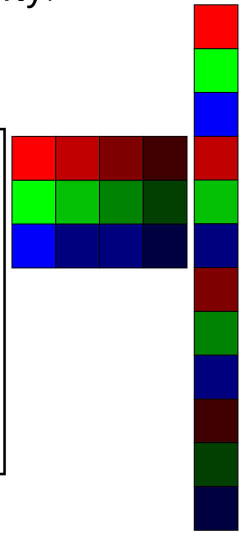
Locality example

- Does this function have good locality?

```
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;
}
```

stride-N reference pattern



Locality example

```
typedef struct {
  float v[3];
  float a[3];
} point ;
point p[N];
```

```
for (i=0; i<n; i++) {
  for (j=0; j<3; j++)
    p[i].v[j]=0;
  for (j=0; j<3; j++)
    p[i].a[j]=0;
}
```

A

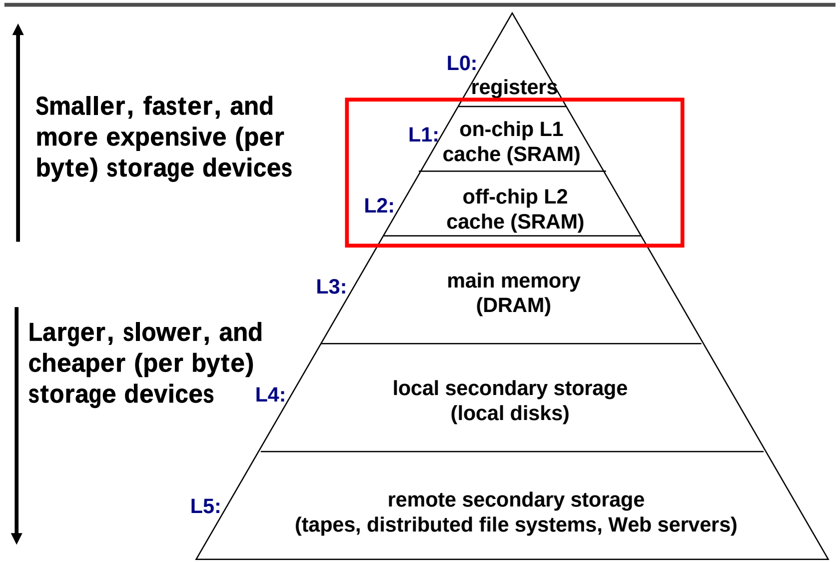
```
for (i=0; i<n; i++) {
  for (j=0; j<3; j++) {
    p[i].v[j]=0;
    p[i].a[j]=0;
  }
}
```

B

```
for (j=0; j<3; j++) {
  for (i=0; i<n; i++)
    p[i].v[j]=0;
  for (i=0; i<n; i++)
    p[i].a[j]=0;
}
```

C

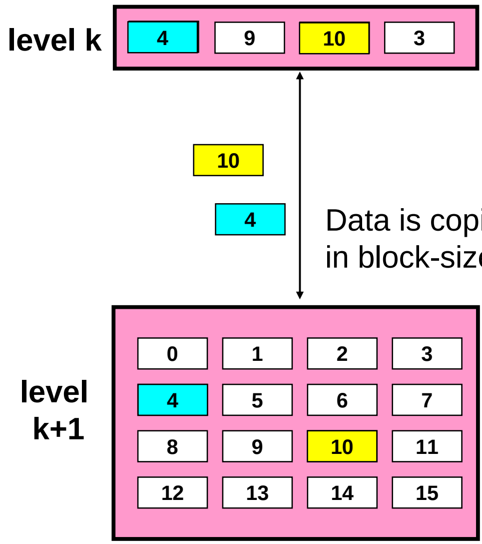
Memory hierarchies



Caches

- **Cache:** a smaller, faster storage device that acts as a staging area for a subset of the data in a larger, slower device.
- Fundamental idea of a memory hierarchy:
  - For each k, the faster, smaller device at level k serves as a cache for the larger, slower device at level k+1.
- Why do memory hierarchies work?
  - Programs tend to access the data at level k more often than they access the data at level k+1.
  - Thus, the storage at level k+1 can be slower, and thus larger and cheaper per bit.

Caching in a memory hierarchy

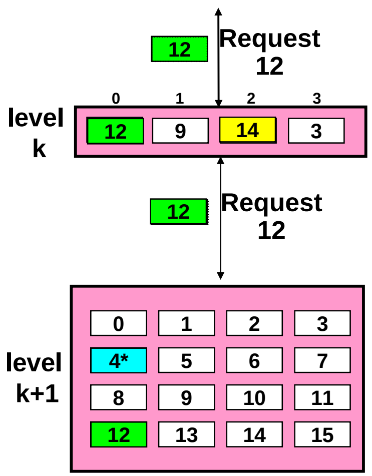


Smaller, faster, more Expensive device at level k caches a subset of the blocks from level k+1

Data is copied between levels in block-sized transfer units

Larger, slower, cheaper Storage device at level k+1 is partitioned into blocks.

General caching concepts



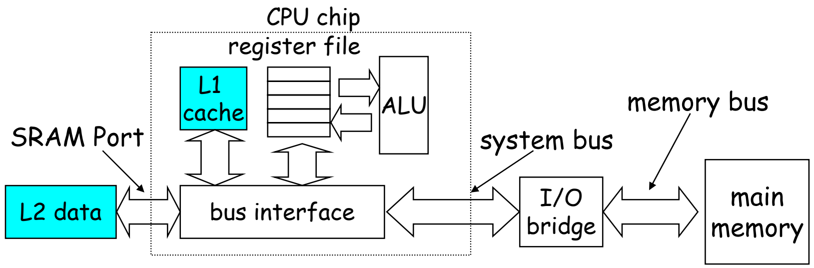
- Program needs object d, which is stored in some block b.
- Cache hit
  - Program finds b in the cache at level k. E.g., block 14.
- Cache miss
  - b is not at level k, so level k cache must fetch it from level k+1. E.g., block 12.
  - If level k cache is full, then some current block must be replaced (evicted). Which one is the “victim”?
    - Placement policy: where can the new block go? E.g., b mod 4
    - Replacement policy: which block should be evicted? E.g., LRU

Type of cache misses

- **Cold (compulsory) miss:** occurs because the cache is empty.
- **Capacity miss:** occurs when the active cache blocks (working set) is larger than the cache.
- **Conflict miss**
  - Most caches limit blocks at level k+1 to a small subset of the block positions at level k, e.g. block i at level k+1 must be placed 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.

Cache memories

- Cache memories are small, fast SRAM-based memories managed automatically in hardware.
- CPU looks first for data in L1, then in L2, then in main memory.
- Typical system structure:

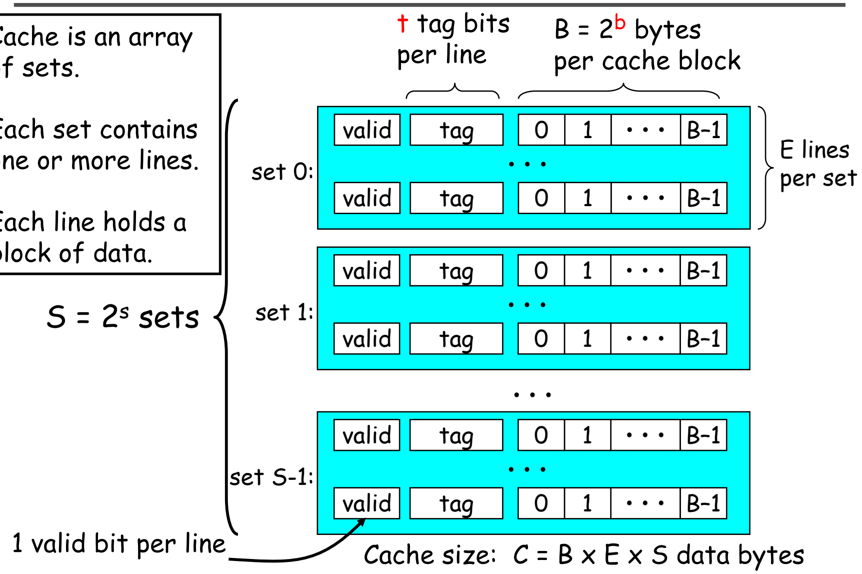


General organization of a cache

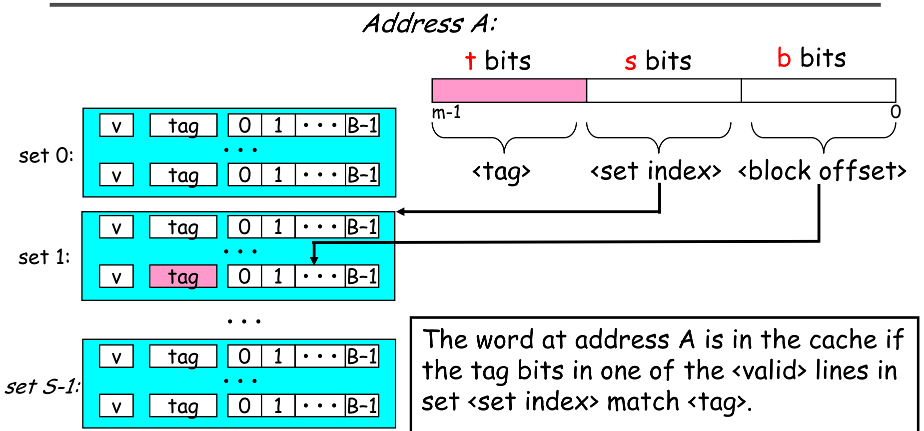
Cache is an array of sets.

Each set contains one or more lines.

Each line holds a block of data.



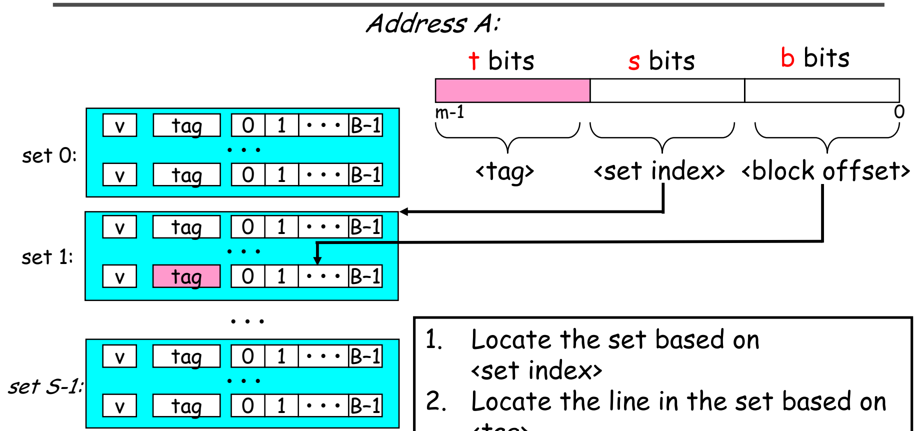
Addressing caches



The word at address A is in the cache if the tag bits in one of the <valid> lines in set <set index> match <tag>.

The word contents begin at offset <block offset> bytes from the beginning of the block.

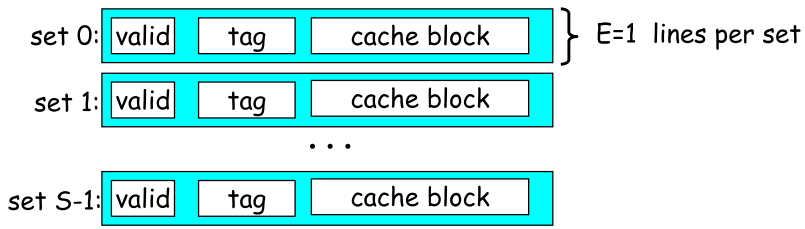
Addressing caches



1. Locate the set based on <set index>
2. Locate the line in the set based on <tag>
3. Check that the line is valid
4. Locate the data in the line based on <block offset>

Direct-mapped cache

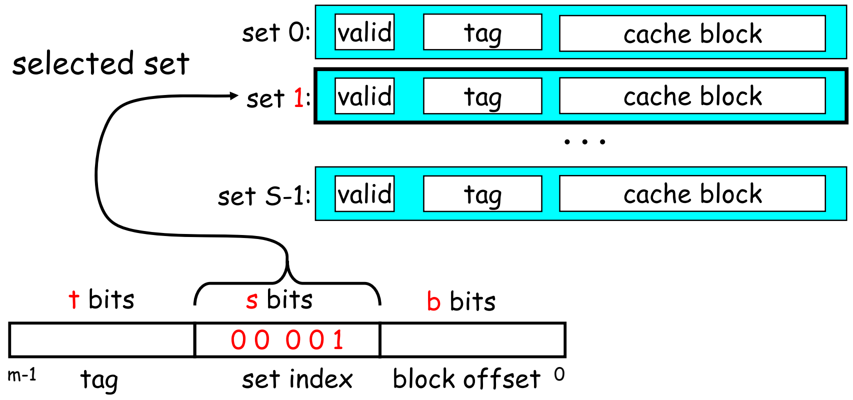
- Simplest kind of cache, easy to build (only 1 tag compare required per access)
- Characterized by exactly one line per set.



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

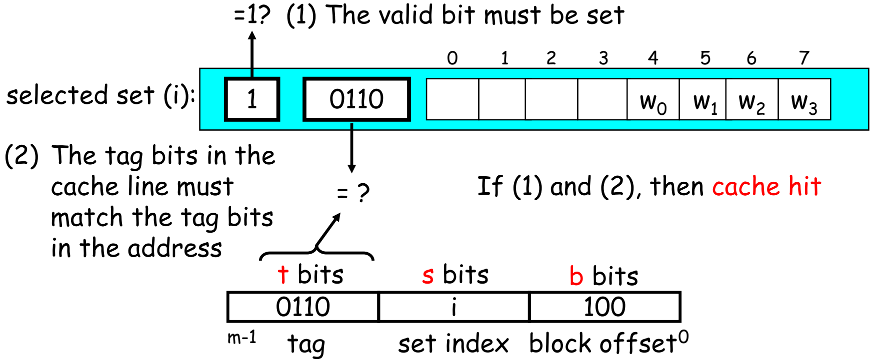
Accessing direct-mapped caches

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



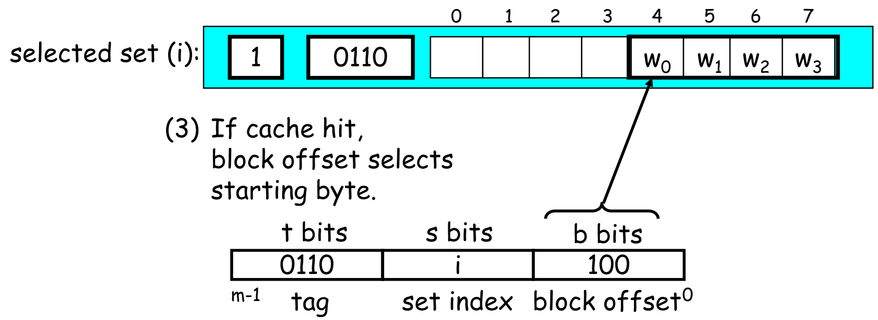
Accessing direct-mapped caches

- 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

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

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

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]

What's wrong with direct-mapped?

```
float dotprod(float x[8], y[8]) {
  float sum=0.0;
  for (int i=0; i<8; i++)
    sum+= x[i]*y[i];
  return sum;
}
```

block size=16 bytes

element	address	set	element	address	set
x[0]	0	0	y[0]	32	0
x[1]	4	0	y[1]	36	0
x[2]	8	0	y[2]	40	0
x[3]	12	0	y[3]	44	0
x[4]	16	1	y[4]	48	1
x[5]	20	1	y[5]	52	1
x[6]	24	1	y[6]	56	1
x[7]	28	1	y[7]	60	1

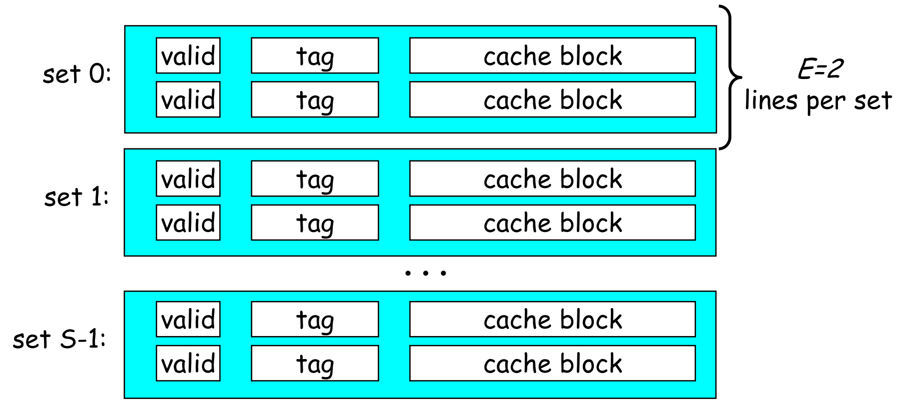
Solution? padding

```
float dotprod(float x[12], y[8]) {
  float sum=0.0;
  for (int i=0; i<8; i++)
    sum+= x[i]*y[i];
  return sum;
}
```

element	address	set	element	Address	set
x[0]	0	0	y[0]	48	1
x[1]	4	0	y[1]	52	1
x[2]	8	0	y[2]	56	1
x[3]	12	0	y[3]	60	1
x[4]	16	1	y[4]	64	0
x[5]	20	1	y[5]	68	0
x[6]	24	1	y[6]	72	0
x[7]	28	1	y[7]	76	0

Set associative caches

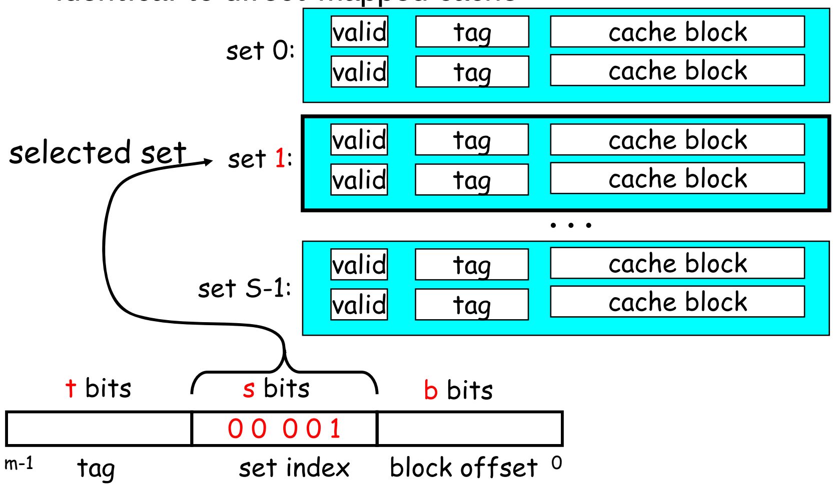
- Characterized by more than one line per set



E-way associative cache

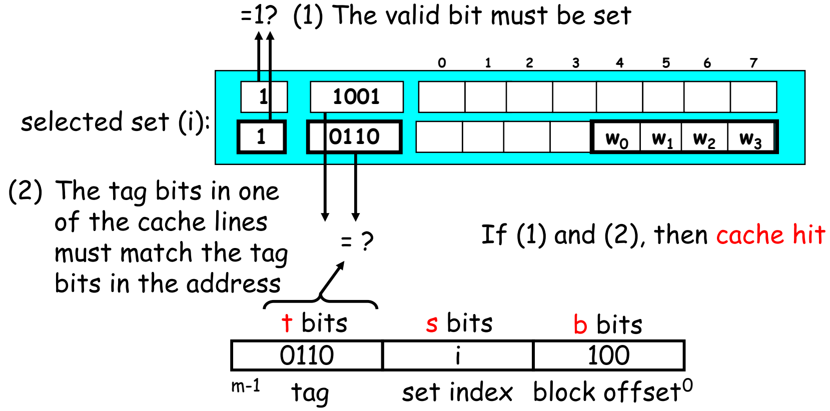
Accessing set associative caches

- Set selection
  - identical to direct-mapped cache



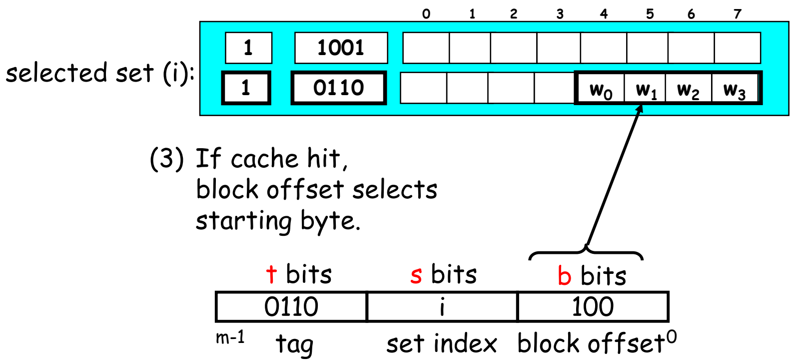
Accessing set associative caches

- 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

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

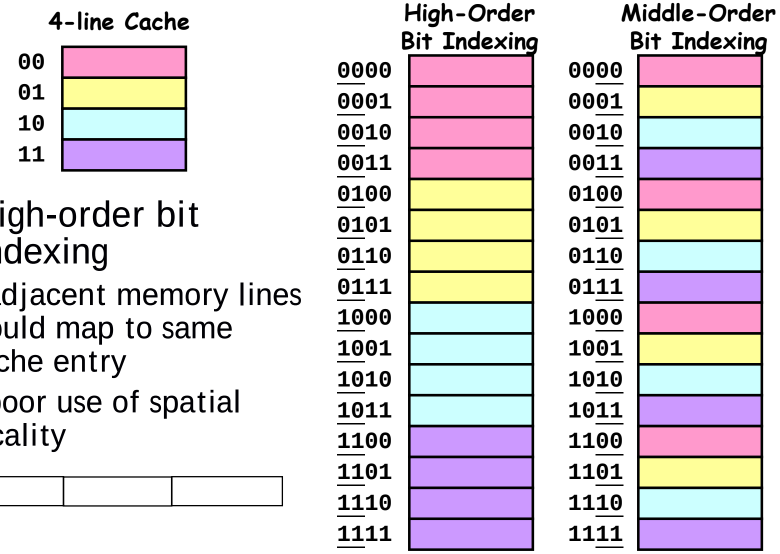
M=16 byte addresses, B=2 bytes/block, S=2 sets, E=2 entry/set

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?



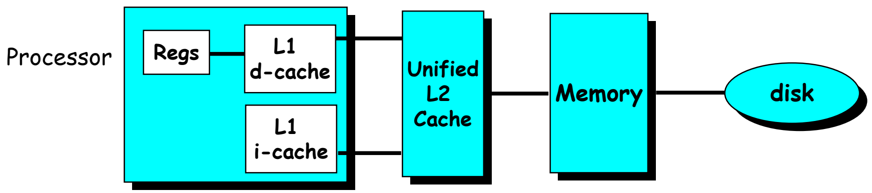
- High-order bit indexing
  - adjacent memory lines would map to same cache entry
  - poor use of spatial locality

What about writes?

- Multiple copies of data exist:
  - L1
  - L2
  - Main Memory
  - Disk
- What to do when we write?
  - Write-through
  - Write-back (need a dirty bit)
- What to do on a replacement?
  - Depends on whether it is write through or write back

Multi-level caches

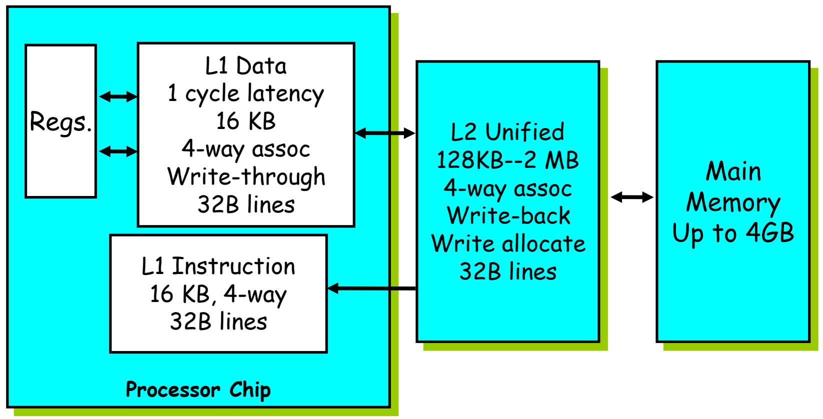
- Options: separate data and instruction caches, or a unified cache



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 →

Intel Pentium III cache hierarchy



Writing cache friendly code

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

```
int sum_array_rows(int a[4][8])
{
  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[4][8])
{
  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%

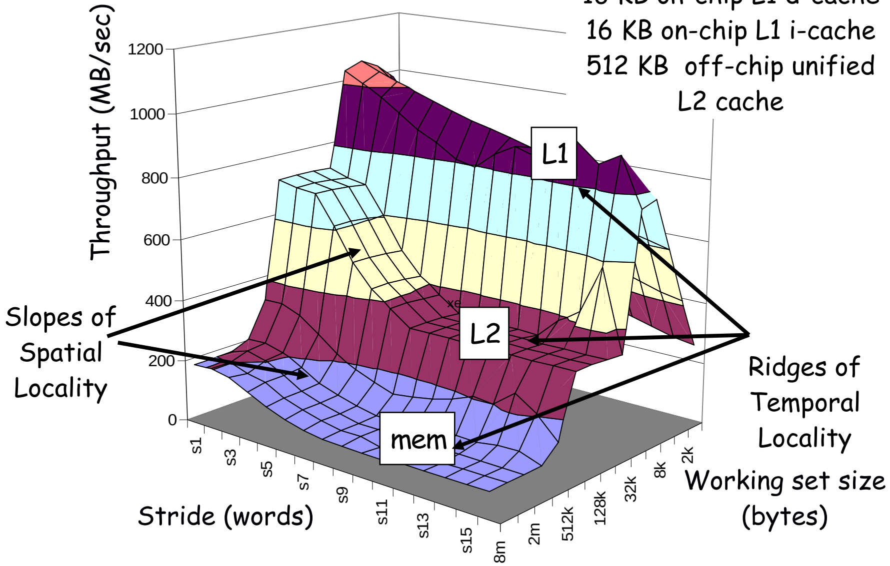
The memory mountain

- Read throughput: 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.

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

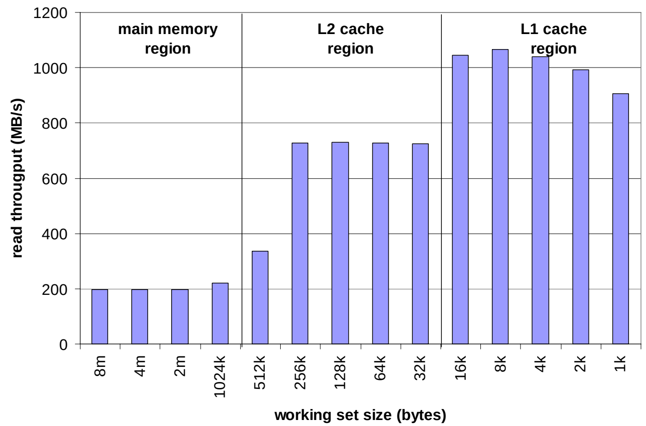
The memory mountain

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



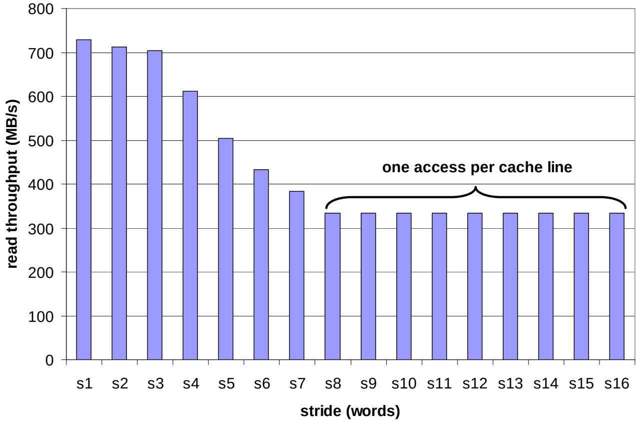
Ridges of temporal locality

- Slice through the memory mountain (stride=1)
  - illuminates read throughputs of different caches and memory



A slope of spatial locality

- Slice through memory mountain (size=256KB)
  - shows cache block size.



Matrix multiplication example

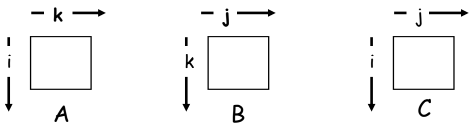
- Major cache effects to consider
  - Total cache size
    - Exploit temporal locality and keep the working set small (e.g., use blocking)
  - Block size
    - Exploit spatial locality
- Description:
  - Multiply N x N matrices
  - O(N^3) total operations
  - Accesses
    - N reads per source element
    - 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*

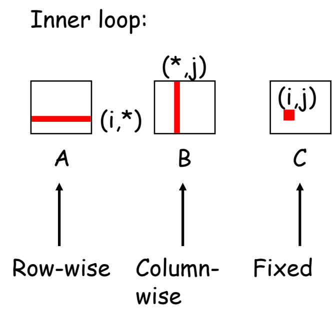
Miss rate analysis for matrix multiply

- 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



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;
  }
}
```

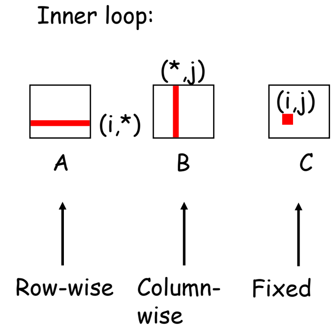


Misses per Inner Loop Iteration:

A	B	C
0.25	1.0	0.0

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
  }
}
```

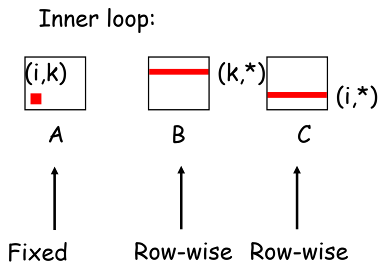


Misses per Inner Loop Iteration:

A	B	C
0.25	1.0	0.0

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];
  }
}
```

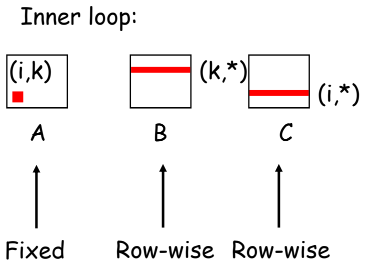


Misses per Inner Loop Iteration:

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];
  }
}
```

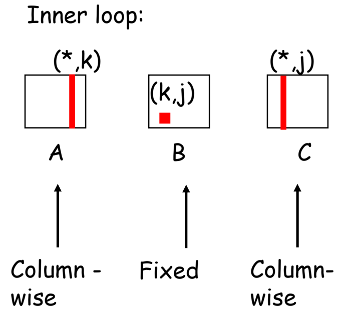


Misses per Inner Loop Iteration:

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;
  }
}
```



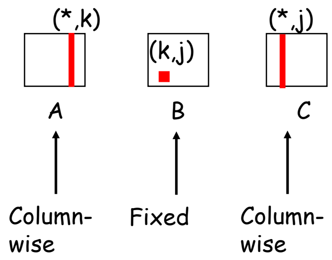
Misses per Inner Loop Iteration:

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;
  }
}
```

Inner loop:



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;
 }
}
```

```
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;
 }
}
```

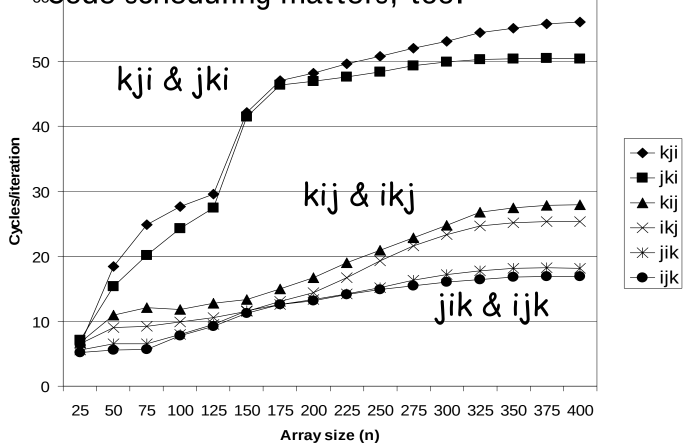
ijk (& jik):
- 2 loads, 0 stores
- misses/iter = **1.25**

kij (& ikj):
- 2 loads, 1 store
- misses/iter = **0.5**

jki (& kji):
- 2 loads, 1 store
- misses/iter = **2.0**

Pentium matrix multiply performance

- Miss rates are helpful but not perfect predictors.
  - Code scheduling matters, too.



Improving temporal locality by blocking

- Example: Blocked matrix multiplication
  - Here, "block" 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.

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

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

Blocked matrix multiply (bijk)

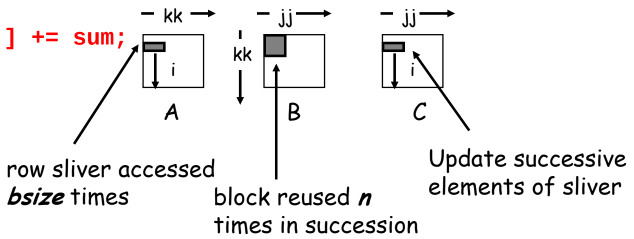
```
for (jj=0; jj<n; jj+=bsize) {
  for (i=0; i<n; i++)
    for (j=jj; j < min(jj+bsize,n); j++)
      c[i][j] = 0.0;
  for (kk=0; kk<n; kk+=bsize) {
    for (i=0; i<n; i++) {
      for (j=jj; j < min(jj+bsize,n); j++) {
        sum = 0.0
        for (k=kk; k < min(kk+bsize,n); k++) {
          sum +=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

Blocked matrix multiply analysis

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

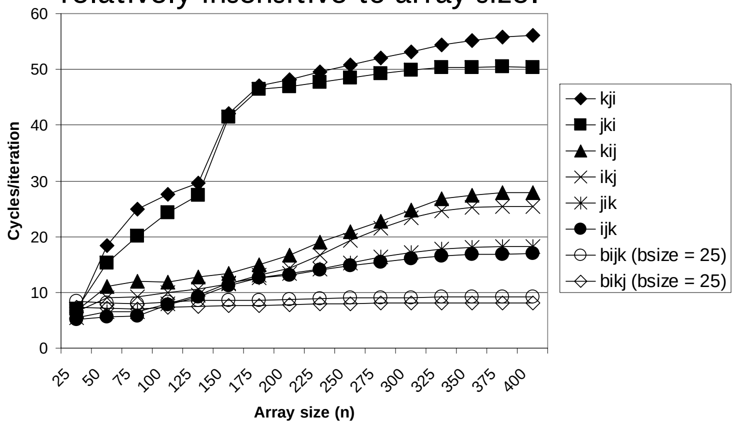
```
for (i=0; i<n; i++) {
  for (j=jj; j < min(jj+bsize,n); j++) {
    sum = 0.0
    for (k=kk; k < min(kk+bsize,n); k++) {
      sum += a[i][k] * b[k][j];
    }
    c[i][j] += sum;
  }
```

Innermost Loop Pair



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)