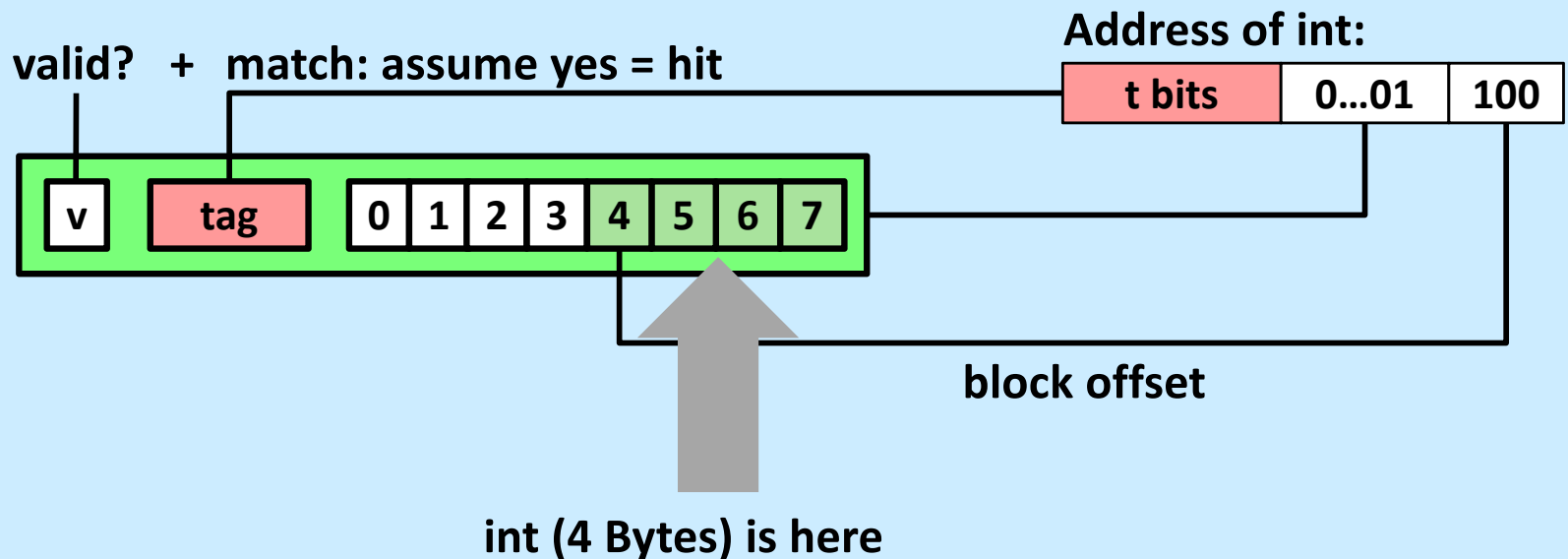


CS 33

Exploiting Caches

Example: Direct Mapped Cache (E = 1)

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



No match: old line is evicted and replaced

Conflict Misses: Aligned

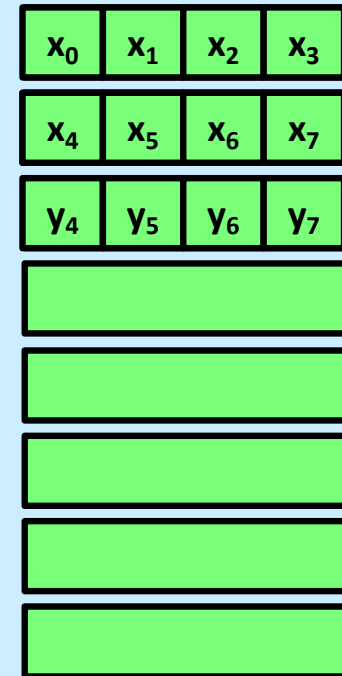
```
double dotprod(double x[8], double y[8]) {  
    double sum = 0.0;  
    int i;  
  
    for (i=0; i<8; i++)  
        sum += x[i] * y[i];  
  
    return sum;  
}
```



32 B = 4 doubles

Different Alignments

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

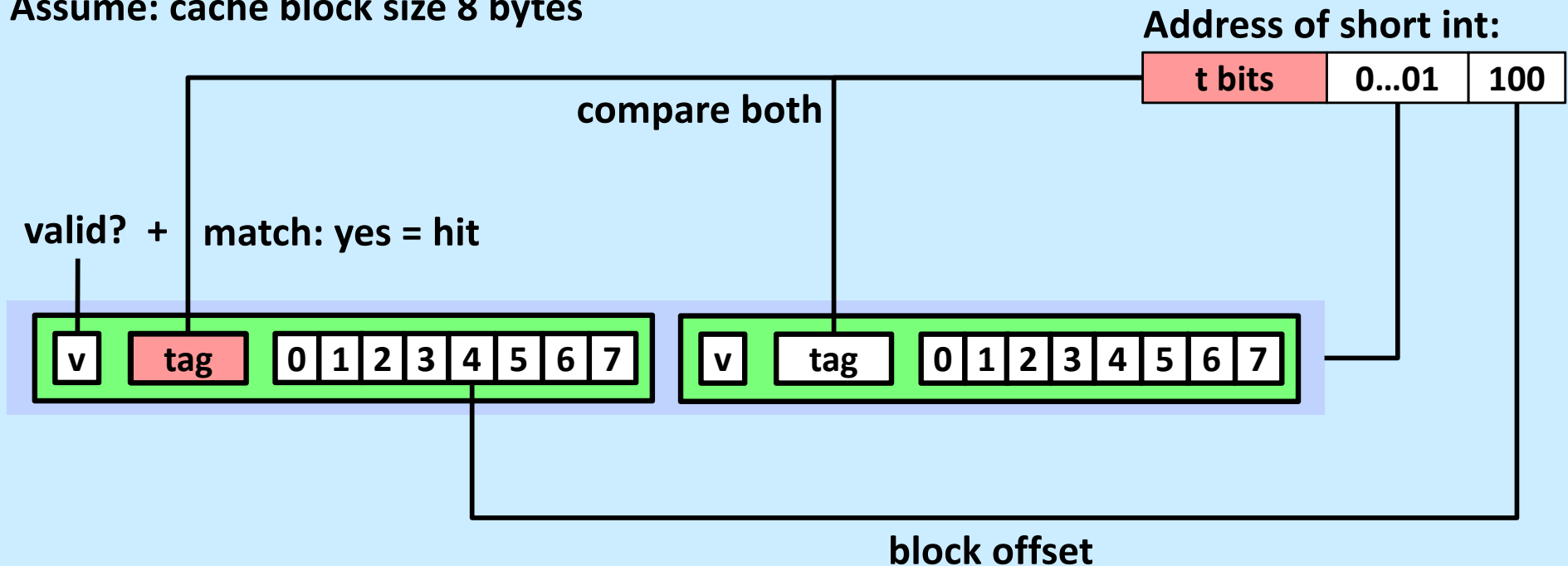


32 B = 4 doubles

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

E = 2: two lines per set

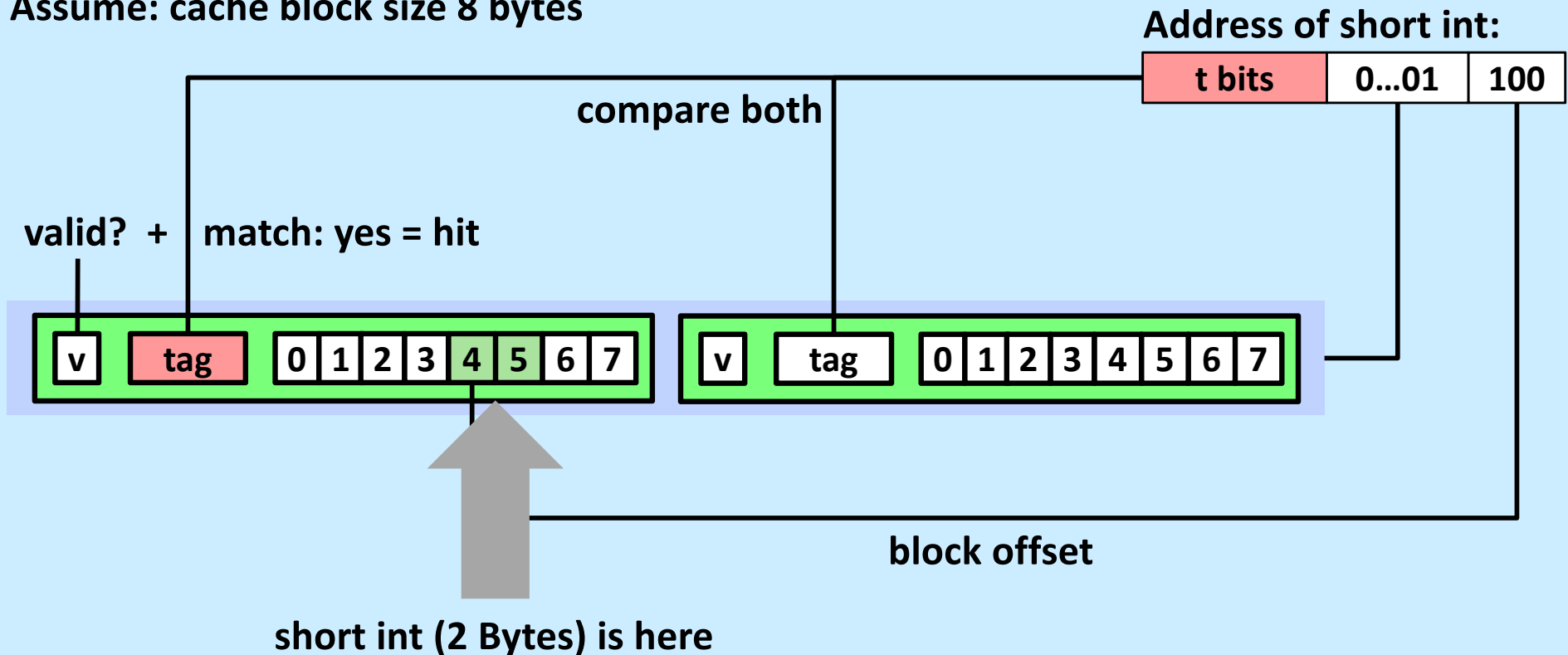
Assume: cache block size 8 bytes



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

E = 2: two lines per set

Assume: cache block size 8 bytes



No match:

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

Quiz 1

Address of int:

100	01	100
-----	----	-----

0	v	tag=0	0	0	0	0	1	1	1	1
1	v	tag=0	4	4	4	4	5	5	5	5
2	v	tag=2	8	8	8	8	9	9	9	9
3	v	tag=4	c	c	c	c	d	d	d	d
	v	tag=2	2	2	2	2	3	3	3	3
	v	tag=4	6	6	6	6	7	7	7	7
	v	tag=3	a	a	a	a	b	b	b	b
	v	tag=a	e	e	e	e	f	f	f	f

Given the address above and the cache contents as shown, what is the value of the *int* at the given address?

- a) 1111
- b) 3333
- c) 4444
- d) 7777

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	[00 <u>00</u> ₂],	miss
1	[00 <u>01</u> ₂],	hit
7	[01 <u>11</u> ₂],	miss
8	[10 <u>00</u> ₂],	miss
0	[00 <u>00</u> 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		

A Higher-Level Example

```
int sum_array_rows(double a[16][16])
{
    int i, j;
    double sum = 0;

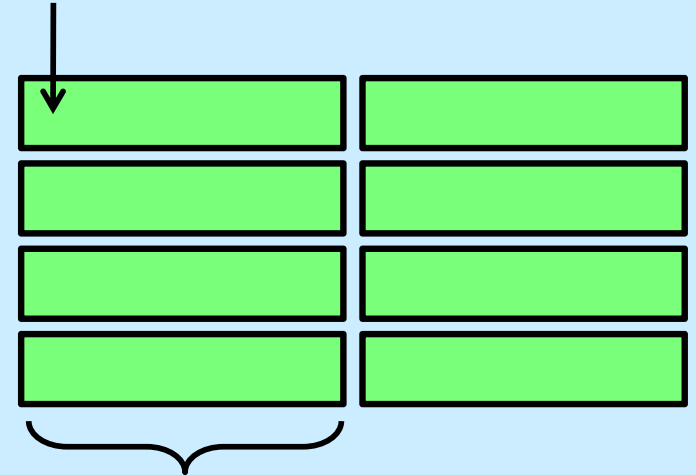
    for (i = 0; i < 16; i++)
        for (j = 0; j < 16; j++)
            sum += a[i][j];
    return sum;
}
```

```
int sum_array_rows(double a[16][16])
{
    int i, j;
    double sum = 0;

    for (j = 0; j < 16; j++)
        for (i = 0; i < 16; i++)
            sum += a[i][j];
    return sum;
}
```

Ignore the variables sum, i, j

assume: cold (empty) cache,
a[0][0] goes here



A Higher-Level Example

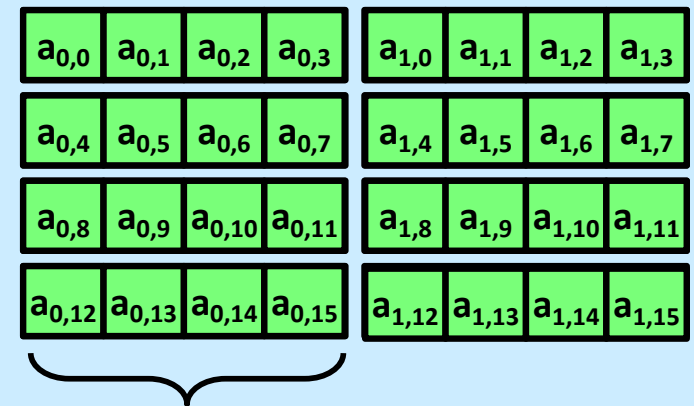
Ignore the variables `sum`, `i`, `j`

```
int sum_array_rows(double a[16][16])
{
    int i, j;
    double sum = 0;

    for (i = 0; i < 16; i++)
        for (j = 0; j < 16; j++)
            sum += a[i][j];
    return sum;
}
```

```
int sum_array_cols(double a[16][16])
{
    int i, j;
    double sum = 0;

    for (j = 0; j < 16; j++)
        for (i = 0; i < 16; i++)
            sum += a[i][j];
    return sum;
}
```



32 B = 4 doubles

A Higher-Level Example

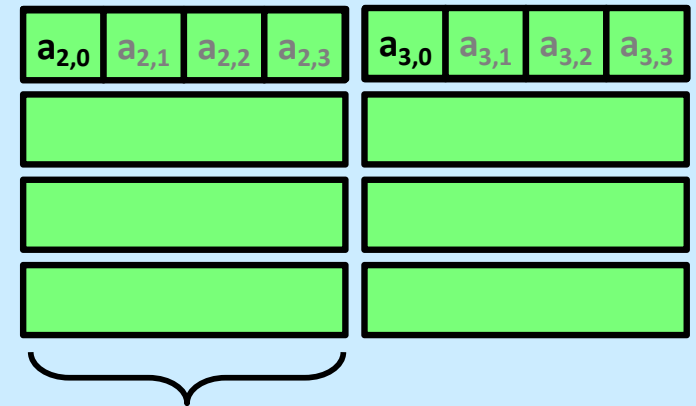
Ignore the variables sum , i , j

```
int sum_array_rows(double a[16][16])
{
    int i, j;
    double sum = 0;

    for (i = 0; i < 16; i++)
        for (j = 0; j < 16; j++)
            sum += a[i][j];
    return sum;
}
```

```
int sum_array_cols(double a[16][16])
{
    int i, j;
    double sum = 0;

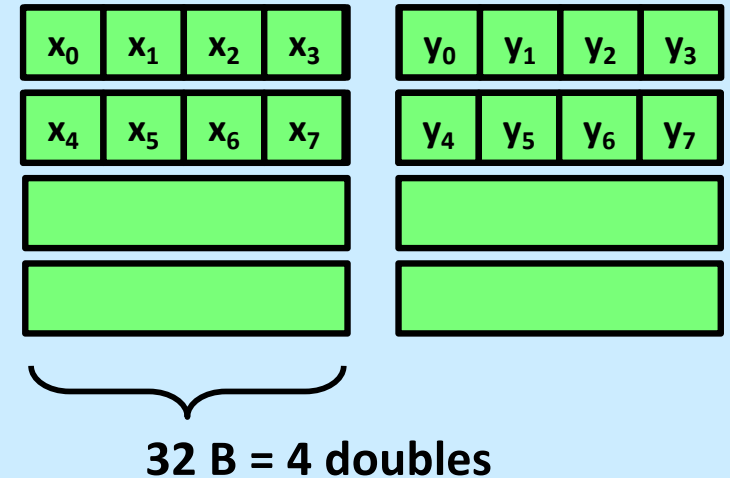
    for (j = 0; j < 16; j++)
        for (i = 0; i < 16; i++)
            sum += a[i][j];
    return sum;
}
```



32 B = 4 doubles

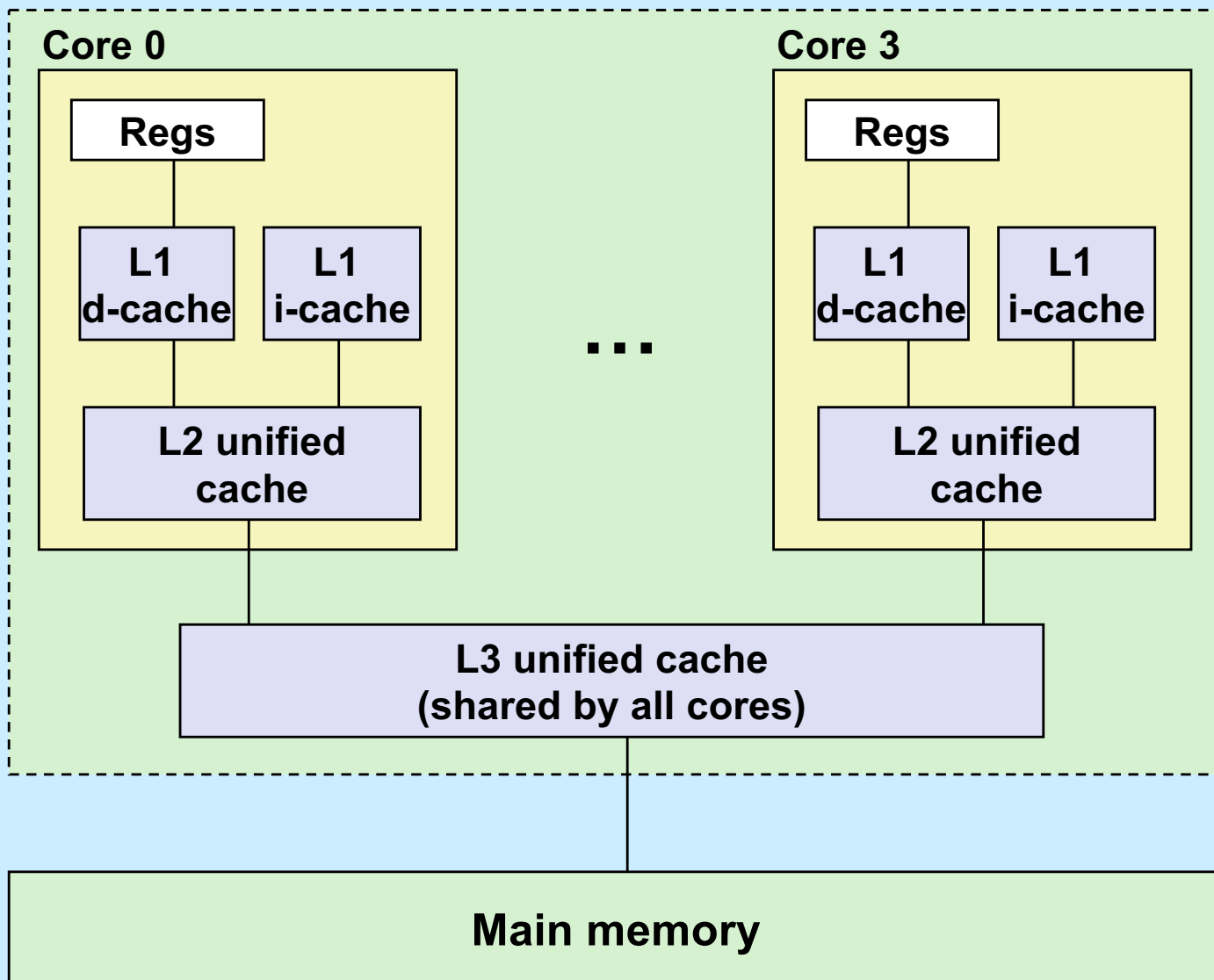
Conflict Misses

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



Intel Core i5 and i7 Cache Hierarchy

Processor package



L1 i-cache and d-cache:

32 KB, 8-way,
Access: 4 cycles

L2 unified cache:

256 KB, 8-way,
Access: 11 cycles

L3 unified cache:

8 MB, 16-way,
Access: 30-40 cycles

Block size: 64 bytes for
all caches

What About Writes?

- Multiple copies of data exist:
 - L1, L2, 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 immediately to memory)
 - Typical
 - write-through + no-write-allocate
 - write-back + write-allocate
-

Accessing Memory

- **Program references memory (load)**
 - if not in cache (*cache miss*), data is requested from RAM
 - » fetched in units of 64 bytes
 - aligned to 64-byte boundaries (low-order 6 bits of address are zeroes)
 - » if memory accessed sequentially, data is pre-fetched
 - » data stored in cache (in 64-byte *cache lines*)
 - stays there until space must be re-used (least recently used is kicked out first)
 - if in cache (*cache hit*) no access to RAM needed
- **Program modifies memory (store)**
 - data modified in cache
 - eventually written to RAM in 64-byte units

Cache Performance Metrics

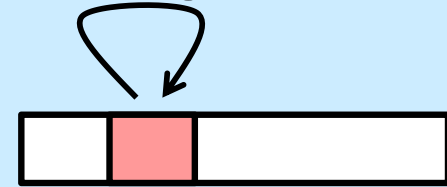
- **Miss rate**
 - fraction of memory references not found in cache (misses / accesses)
= 1 – hit rate
 - typical numbers (in percentages):
 - » 3-10% for L1
 - » can be quite small (e.g., < 1%) for L2, depending on size, etc.
 - **Hit time**
 - time to deliver a line in the cache to the processor
 - » includes time to determine whether the line is in the cache
 - typical numbers:
 - » 1-2 clock cycles for L1
 - » 5-20 clock cycles for L2
 - **Miss penalty**
 - additional time required because of a miss
 - » typically 50-200 cycles for main memory (trend: increasing!)
-

Hits vs. Misses

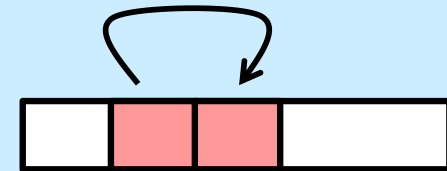
- Huge difference between hit and miss times
 - could be 100x, if just L1 and main memory
- 99% hit rate is twice as good as 97%!
 - consider:
cache hit time of 1 cycle
miss penalty of 100 cycles
 - average access time:
97% hits: $.97 * 1 \text{ cycle} + 0.03 * 100 \text{ cycles} \approx 4 \text{ cycles}$
99% hits: $.99 * 1 \text{ cycle} + 0.01 * 100 \text{ cycles} \approx 2 \text{ cycles}$
- This is why “miss rate” is used instead of “hit rate”

Locality

- **Principle of Locality:** programs tend to use data and instructions with addresses near or equal 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.
- cycle through loop repeatedly

Spatial locality

Temporal locality

Quiz 2

Does this function have good locality with respect to array a? The array a is MxN.

- a) yes
- b) no

```
int sum_array_cols(int N, int a[][N]) {  
    int i, j, sum = 0;  
  
    for (j = 0; j < N; j++)  
        for (i = 0; i < M; i++)  
            sum += a[i][j];  
    return sum;  
}
```

Writing Cache-Friendly Code

- **Make the common case fast**
 - focus on the inner loops of the core functions
- **Minimize the misses in the inner loops**
 - repeated references to variables are good (**temporal locality**)
 - stride-1 reference patterns are good (**spatial locality**)

Matrix Multiplication Example

- **Description:**
 - multiply $N \times N$ matrices
 - » each element is a double
 - $O(N^3)$ total operations
 - N reads per source element
 - N values summed per destination
 - » but may be able to hold in register

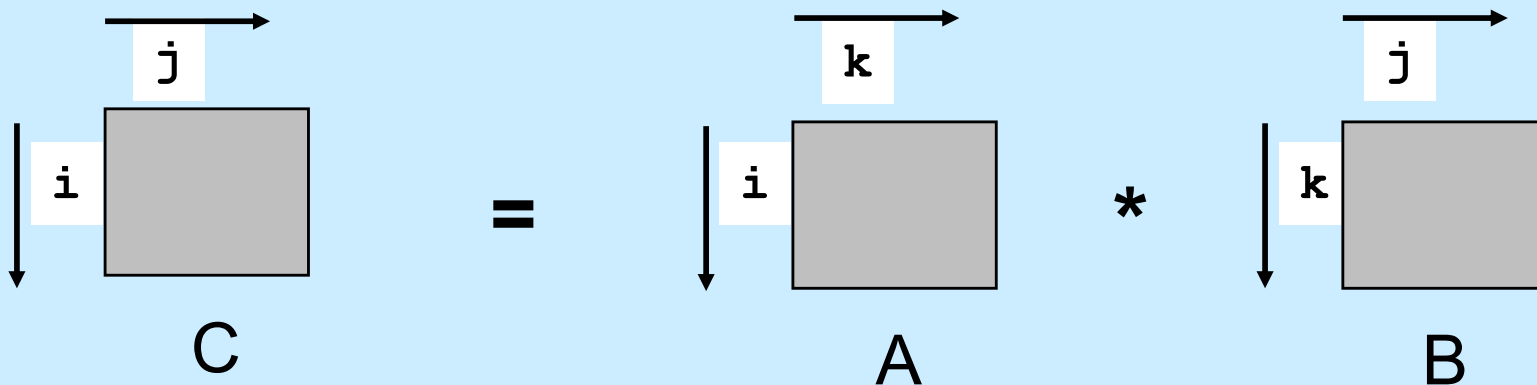
```
/* ijk */
for (i=0; i<n; i++) {
    for (j=0; j<n; j++) {
        sum = 0.0;
        for (k=0; k<n; k++)
            sum += a[i][k] * b[k][j];
        c[i][j] = sum;
    }
}
```

*Variable sum
held in register*

```
/* 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];
    }
}
```

Miss-Rate Analysis for Matrix Multiply

- **Assume:**
 - Block size = 64B (big enough for eight doubles)
 - matrix dimension (N) is very large
 - cache is not big enough to hold multiple rows
- **Analysis method:**
 - look at access pattern of inner loop



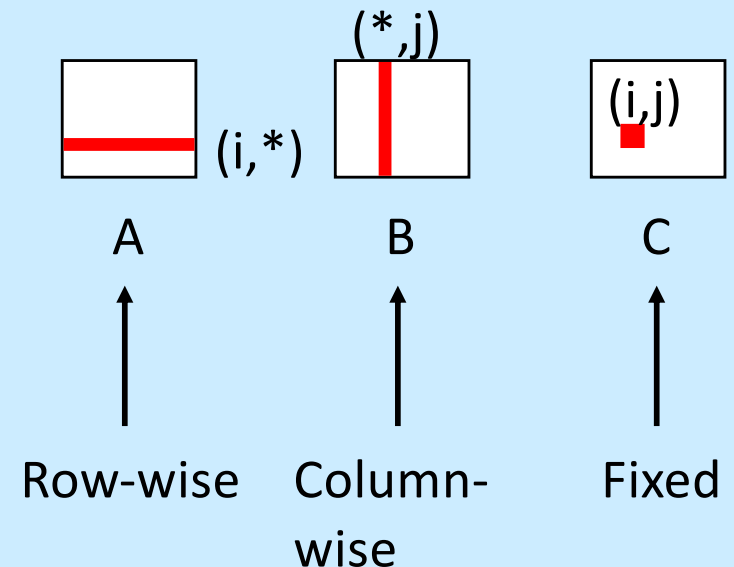
Layout of C Arrays in Memory (review)

- **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) > 8 bytes, exploit spatial locality**
 - » compulsory miss rate = 8 bytes / Block
- **Stepping through rows in one column:**
 - **for** (`i = 0; i < n; i++`)
 `sum += a[i][0];`
 - **accesses widely separated 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:



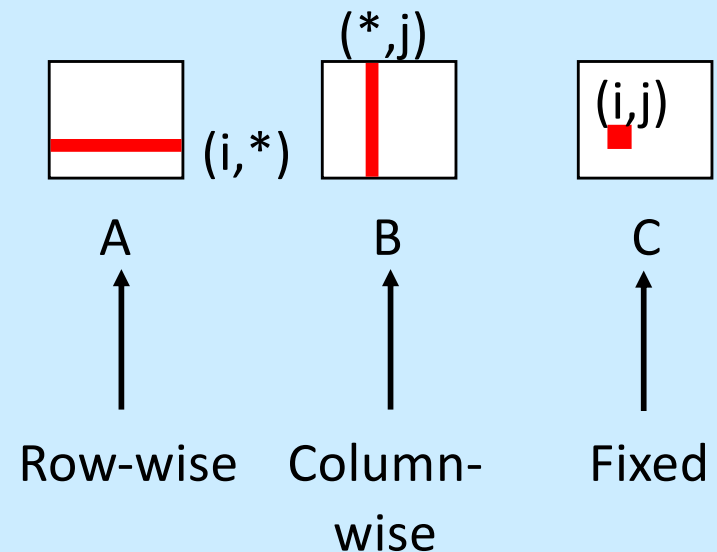
Misses per inner loop iteration:

<u>A</u>	<u>B</u>	<u>C</u>
0.125	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:



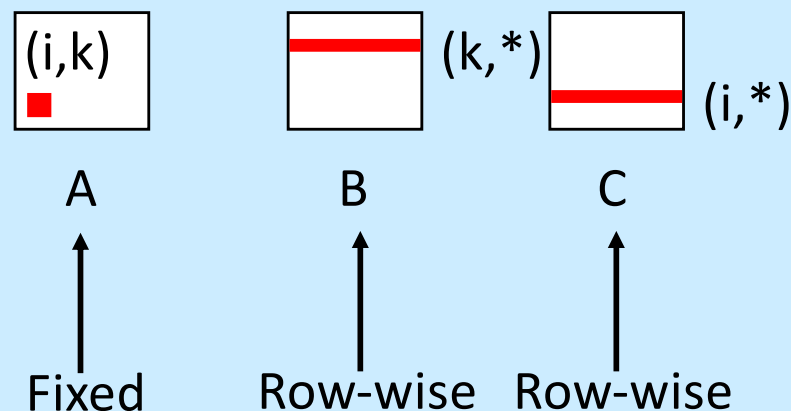
Misses per inner loop iteration:

<u>A</u>	<u>B</u>	<u>C</u>
0.125	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];
    }
}
```

Inner loop:



Misses per inner loop iteration:

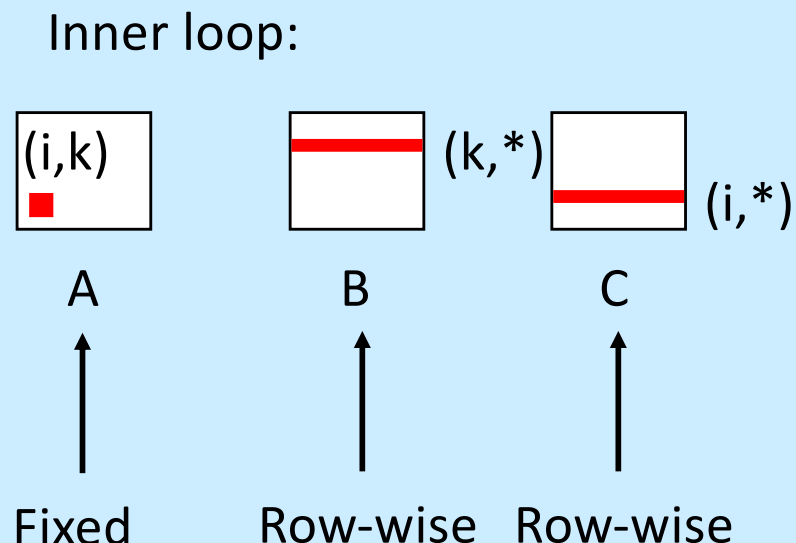
A
0.0

B
0.125

C
0.125

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
0.0

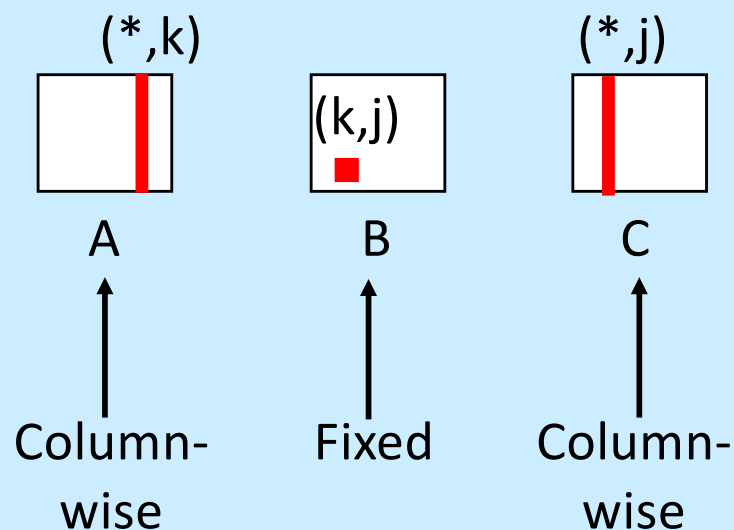
B
0.125

C
0.125

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;  
    }  
}
```

Inner loop:



Misses per inner loop iteration:

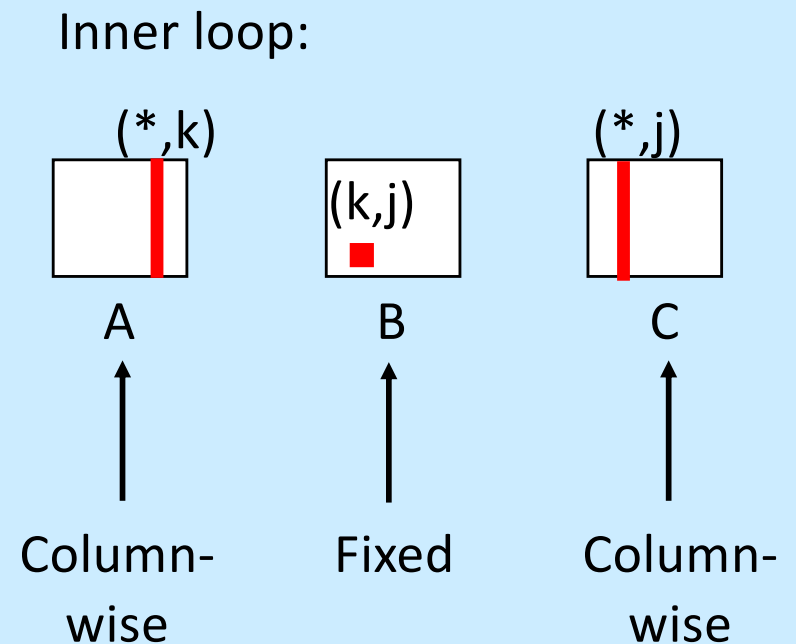
A
1.0

B
0.0

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:

A
1.0

B
0.0

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 = **1.125**

```
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):

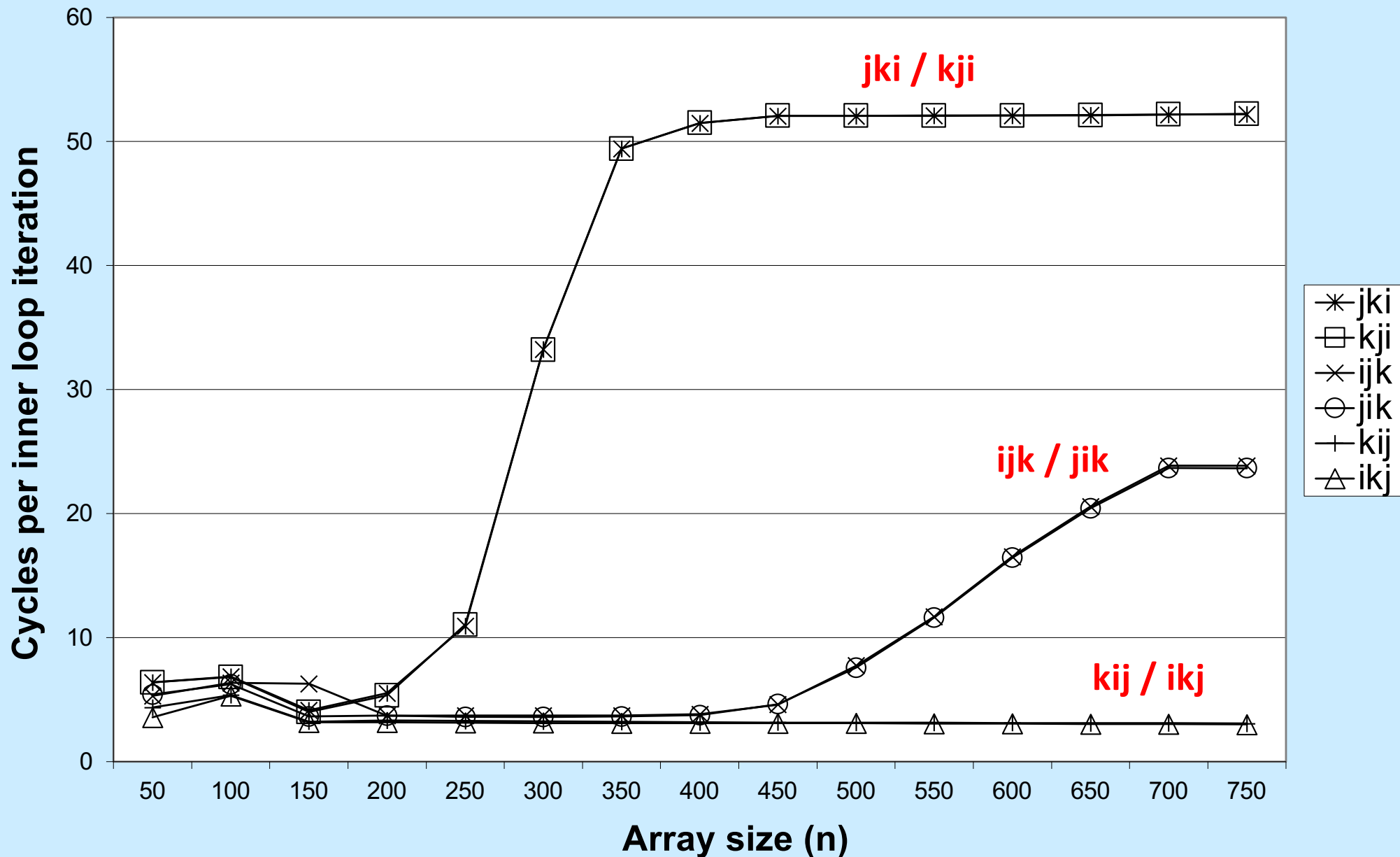
- 2 loads, 1 store
- misses/iter = **0.25**

```
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**

Core i7 Matrix Multiply Performance



In Real Life ...

- **Multiply two 1024x1024 matrices of doubles on sunlab machines**

- **ijk**

- » **4.185 seconds**

- **kij**

- » **0.798 seconds**

- **jki**

- » **11.488 seconds**

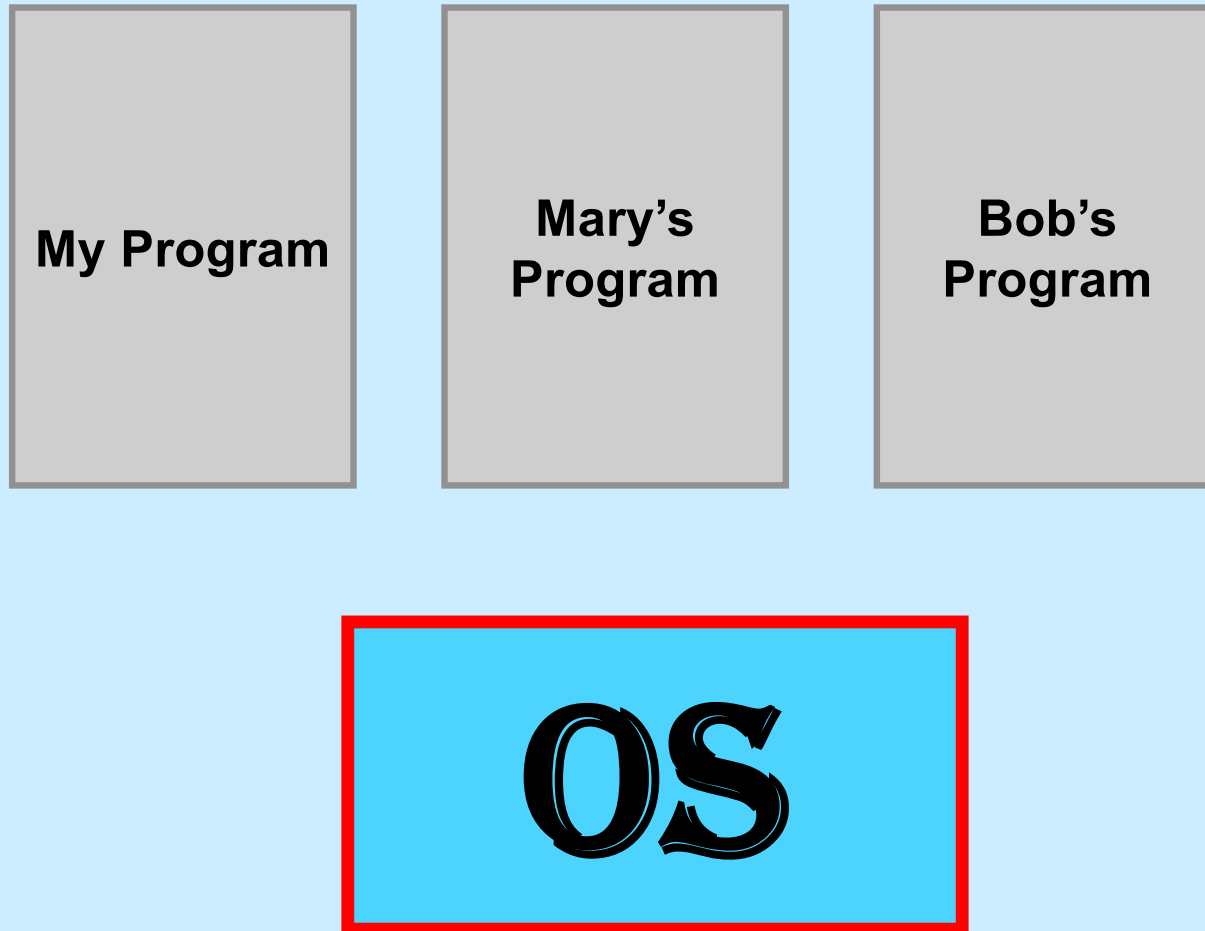
Concluding Observations

- **Programmer can optimize for cache performance**
 - organize data structures appropriately
- **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)

CS 33

Architecture and the OS

The Operating System



Processes

- **Containers for programs**
 - **virtual memory**
 - » **address space**
 - **scheduling**
 - » **one or more threads of control**
 - **file references**
 - » **open files**
 - **and lots more!**

Idiot Proof ...

```
int main( ) {  
    int i;  
    int A[1];  
  
    for (i=0; ; i++)  
        A[rand()] = i;  
}
```

Can I clobber
Mary's
program?

Mary's
Program

Fair Share

```
void runforever( ) {  
    while(1)  
        ;  
}  
  
int main( ) {  
    runforever();  
}
```

Can I
prevent Bob's
program from
running?

**Bob's
Program**

Architectural Support for the OS

- **Not all instructions are created equal ...**
 - **non-privileged instructions**
 - » can affect only current program
 - **privileged instructions**
 - » may affect entire system
- **Processor mode**
 - **user mode**
 - » can execute only non-privileged instructions
 - **privileged mode**
 - » can execute all instructions

Which Instructions Should Be Privileged?

- I/O instructions
- Those that affect how memory is mapped
- Halt instruction
- Some others ...

Who Is Privileged?

- **No one**
 - user code always runs in user mode
- **The operating-system kernel runs in privileged mode**
 - nothing else does
 - not even super user on Unix or administrator on Windows

Entering Privileged Mode

- **How is OS invoked?**
 - very carefully ...
 - strictly in response to interrupts and exceptions
 - (booting is a special case)

Interrupts and Exceptions

- **Things don't always go smoothly ...**
 - I/O devices demand attention
 - timers expire
 - programs demand OS services
 - programs demand storage be made accessible
 - programs have problems
- **Interrupts**
 - demand for attention by external sources
- **Exceptions**
 - executing program requires attention

Exceptions

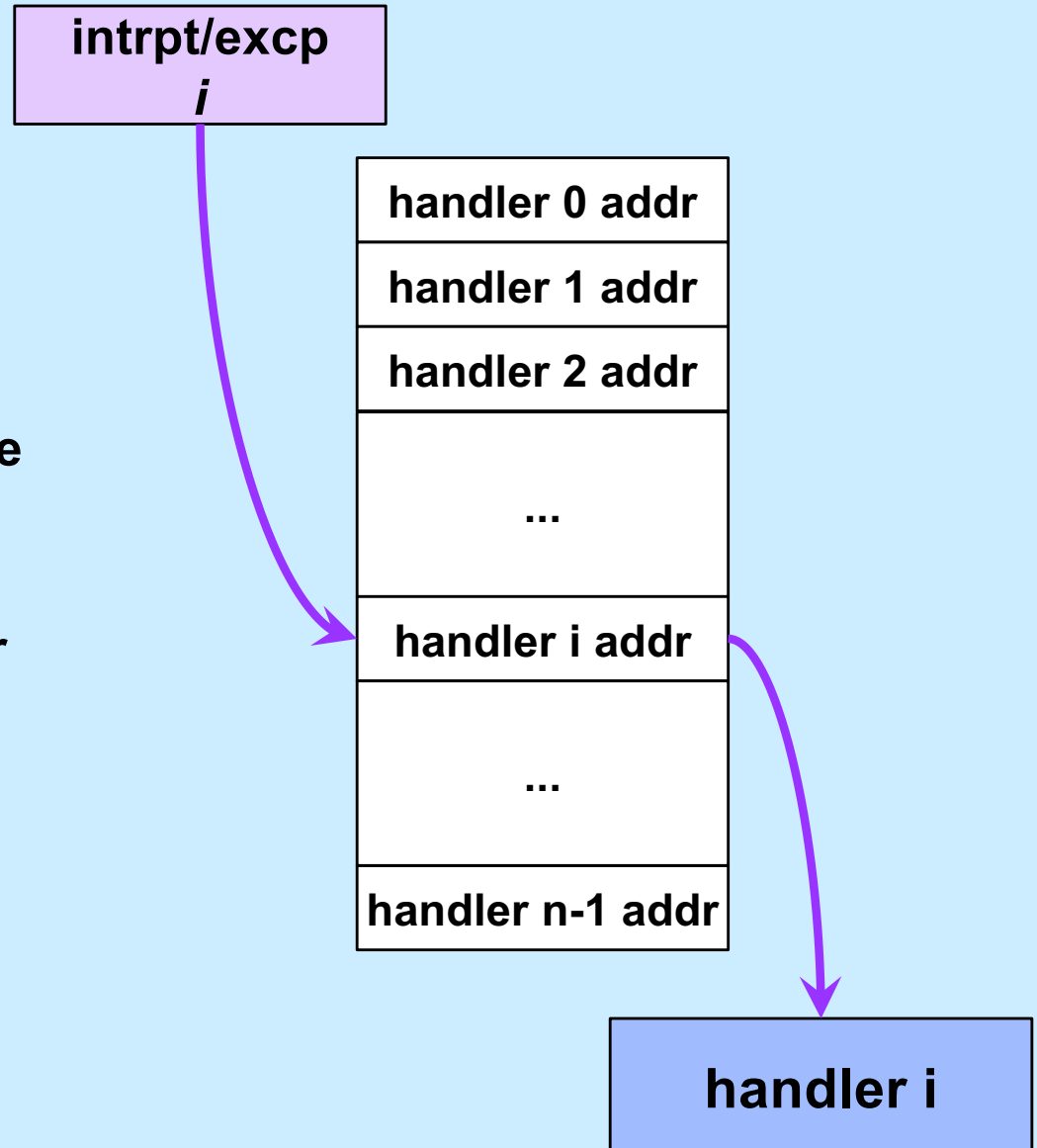
- **Traps**
 - “intentional” exceptions
 - » execution of special instruction to invoke OS
 - after servicing, execution resumes with next instruction
- **Faults**
 - a problem condition that is normally corrected
 - after servicing, instruction is re-tried
- **Aborts**
 - something went dreadfully wrong ...
 - not possible to re-try instruction, nor to go on to next instruction

Actions for Interrupts and Exceptions

- **When interrupt or exception occurs**
 - processor saves state of current thread/process on stack
 - processor switches to privileged mode (if not already there)
 - invokes handler for interrupt/exception
 - if thread/process is to be resumed (typical action after interrupt)
 - » thread/process state is restored from stack
 - if thread/process is to re-execute current instruction
 - » thread/process state is restored, after backing up instruction pointer
 - if thread/process is to terminate
 - » it's terminated

Interrupt and Exception Handlers

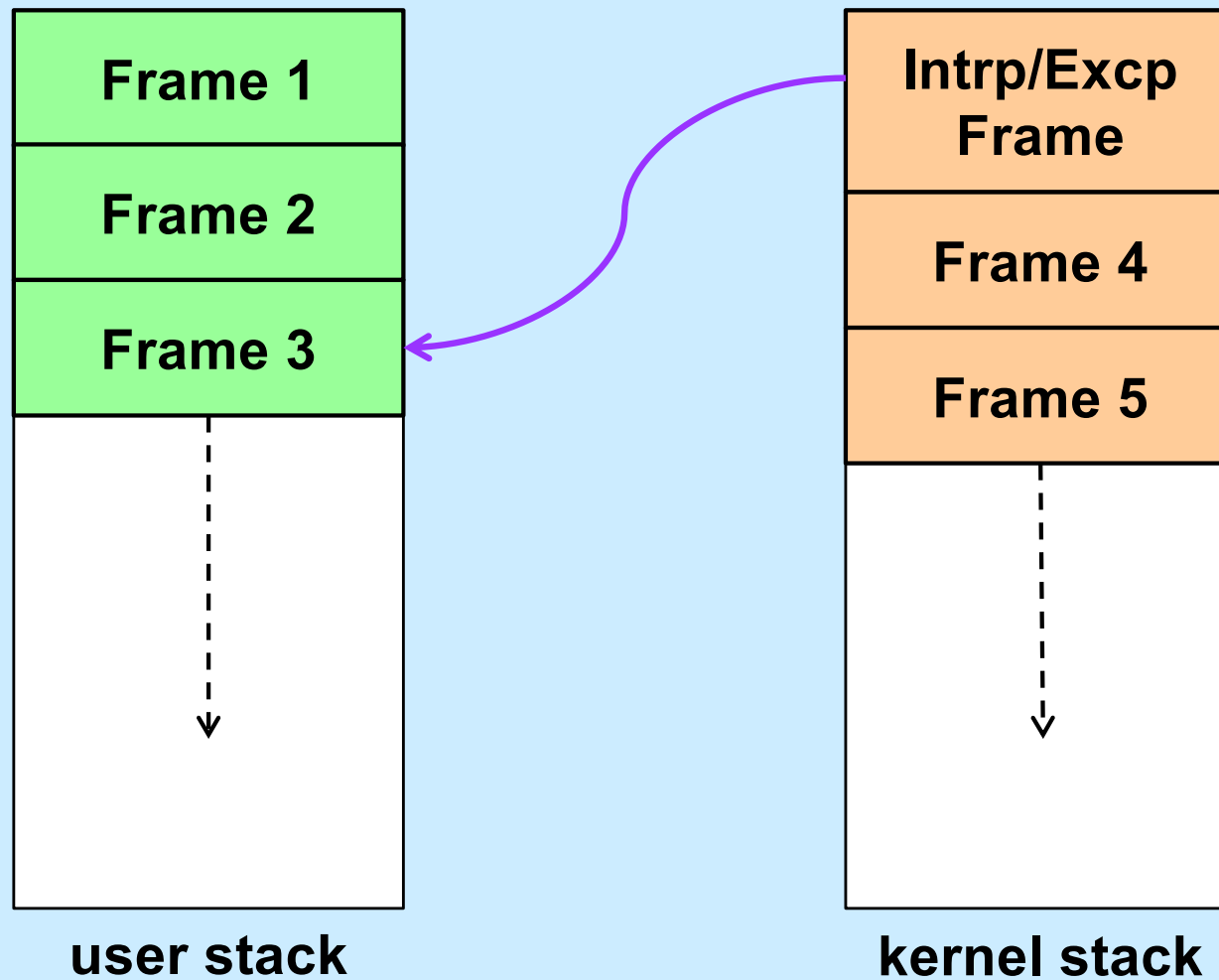
- **Interrupt or exception invokes handler (in OS)**
 - via interrupt and exception vector
 - » one entry for each possible interrupt/exception
 - contains
 - address of handler
 - code executed in privileged mode
 - » but code is part of the OS



Entering and Exiting

- **Entering/exiting interrupt/exception handler more involved than entering/exiting a procedure**
 - **must deal with processor mode**
 - » **switch to privileged mode on entry**
 - » **switch back to previous mode on exit**
 - **interrupted process/thread's state is saved on separate kernel stack**
 - **stack in kernel must be different from stack in user program**
 - » **why?**

One Stack Per Mode



Quiz 3

If an interrupt occurs, which general-purpose registers must be pushed onto the kernel stack?

- a) all
- b) none
- c) callee-save registers
- d) caller-save registers