MEMORY HIERARCHY DESIGN

B649
Parallel Architectures and Programming

Basic Optimizations

Average memory access time = Hit time + Miss rate × Miss penalty

- Larger block size to reduce miss rate
- Larger caches to reduce miss rate
- Higher associativity to reduce miss rate
- Multilevel caches to reduce miss penalty
- Prioritizing read misses over writes to reduce miss penalty
- Avoiding address translation during indexing of the cache to reduce hit time

ADVANCED OPTIMIZATIONS

Eleven Advanced Optimizations

Average memory access time = Hit time + Miss rate × Miss penalty

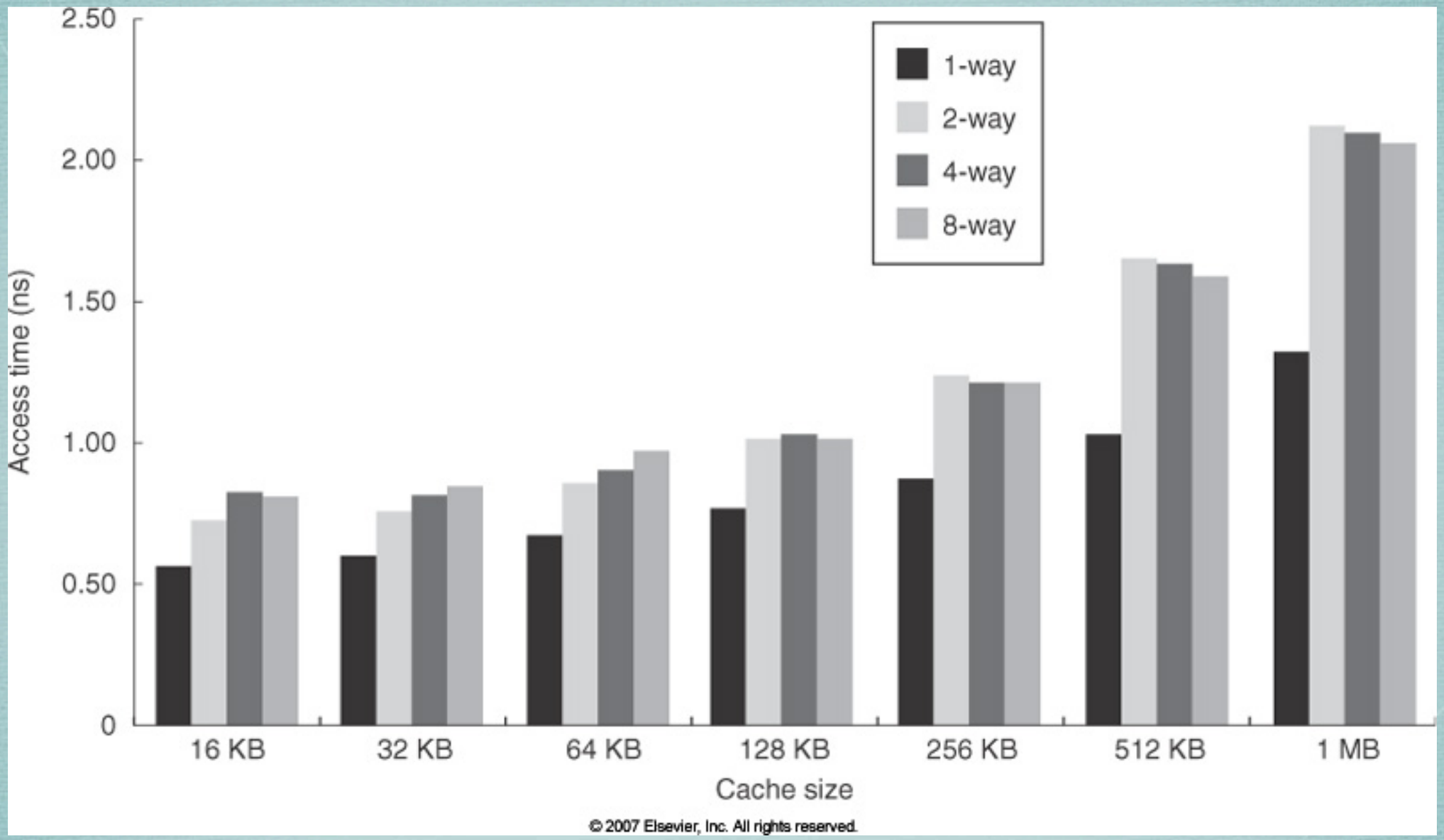
- Reducing the hit time
  - ⋆ small and simple caches, way prediction, trace caches
- Increasing cache bandwidth
  - ⋆ pipelined caches, multibanked caches, non-blocking caches
- Reducing the miss penalty
  - ⋆ critical word first, merging write buffers
- Reducing the miss rate
  - ⋆ compiler optimizations
- Reducing miss penalty / miss rate via parallelism
  - ⋆ hardware prefetching, compiler prefetching

#1: Small and Simple Caches

(To Reduce Hit Time)

- Small caches can be faster
  - ⋆ reading tags and comparing is time-consuming
  - ⋆ L1 should be fast enough to be read in 1-2 cycles
  - ⋆ desirable to keep L2 small enough to fit on chip
    - ✳ could keep data off-chip and tags on-chip
- Simpler caches can be faster
  - ⋆ direct-mapped caches: can overlap tag check and transmission of data
    - ✳ Why is this not possible with set-associative caches?

Access Times on a CMOS Cache (CACTI)



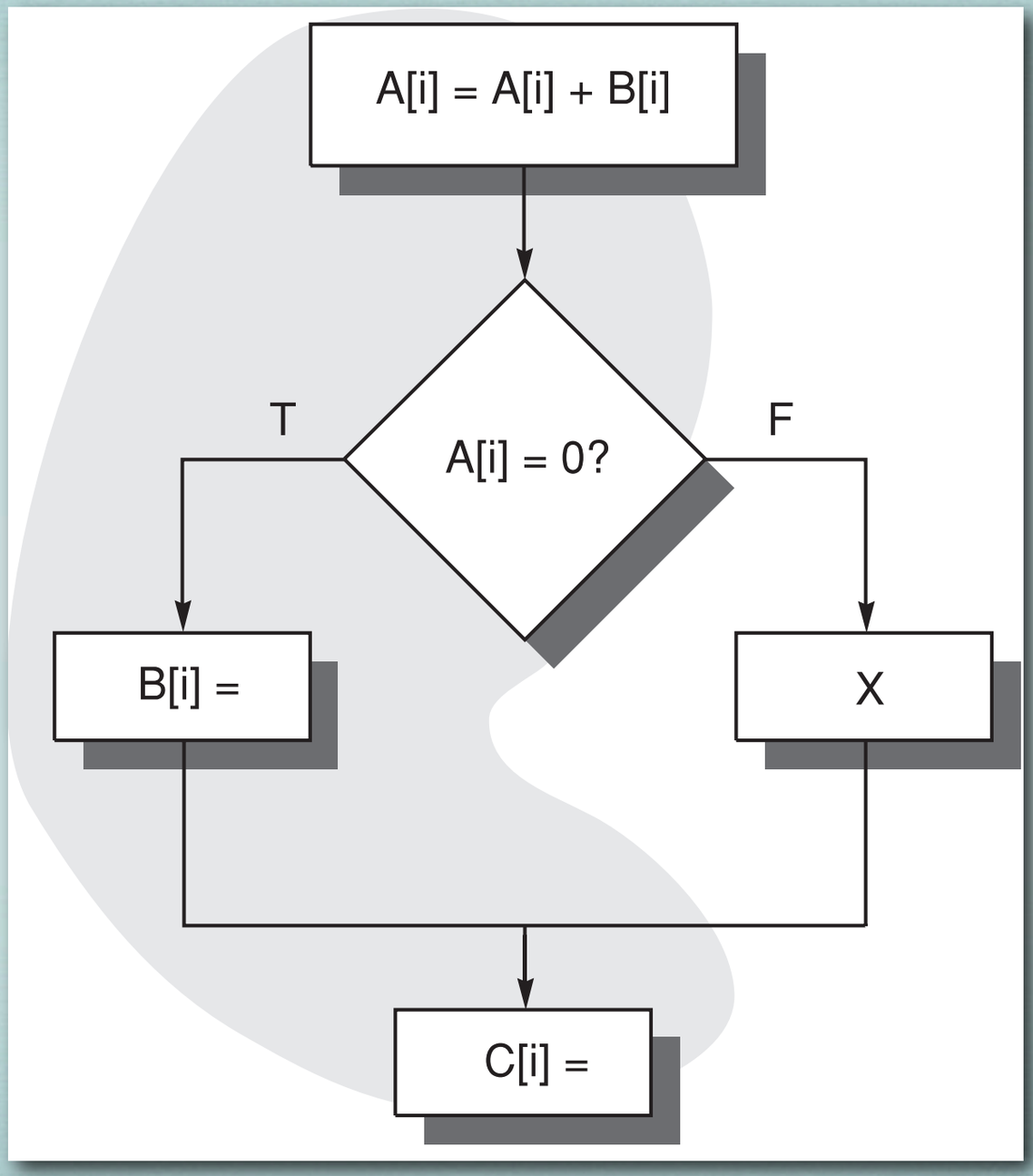
#2: Way Prediction

(To Reduce Hit Time)

- Extra bits per block to predict the *way* (block within the set) of the **next** cache access
  - ⋆ can set the multiplexer early
  - ⋆ can match the tag and read data in parallel
  - ⋆ miss results in matching other blocks in next clock cycle
- Prediction accuracy > 85% suggested by simulations
  - ⋆ good match for speculative processors
  - ⋆ used in Pentium 4

#3: Trace Caches

(To Reduce Hit Time)



#3: Trace Caches

(To Reduce Hit Time)

- Goal: to enhance instruction-level parallelism (find sufficient number of instructions without dependencies)
  - ⋆ trace = dynamic sequence of executed instructions
- Using traces
  - ⋆ branches folded into traces, hence need to be validated
  - ⋆ more complicated address mapping (?)
  - ⋆ better utilize long blocks
  - ⋆ conditional branches cause duplication of instructions across traces
  - ⋆ used in Pentium 4 (in general, benefits not obvious)

#4: Pipelined Cache Access

(To Increase Cache Bandwidth)

- Pipeline results in fast clock cycle time and high bandwidth, but slow hits
  - ⋆ Pentium 1: 1 clock cycle for instruction cache
  - ⋆ Pentium Pro / III: 2 clock cycles
  - ⋆ Pentium 4: 4 clock cycles

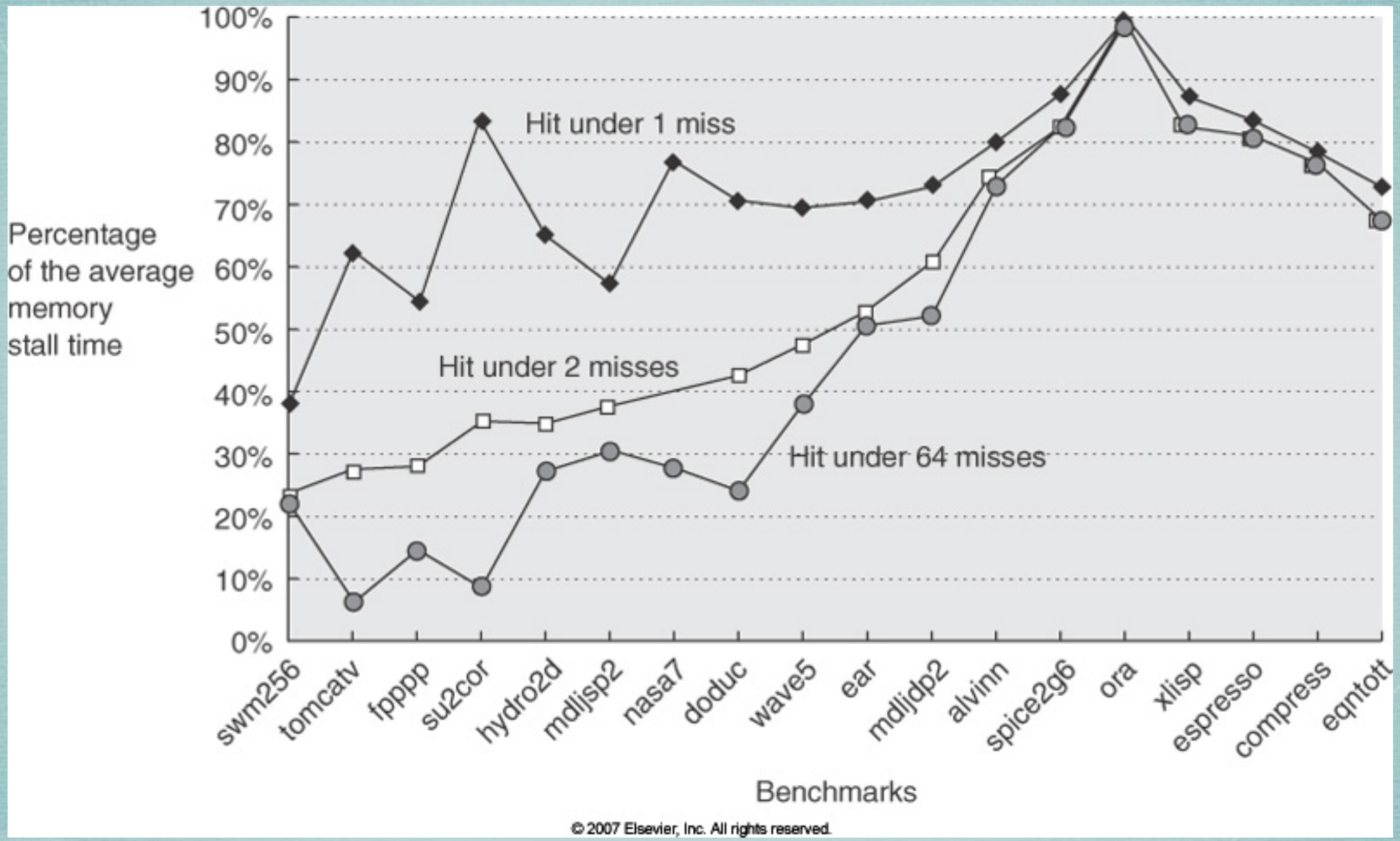
#5: Nonblocking Caches

(To Increase Cache Bandwidth)

- *Nonblocking* or *lockup-free* cache increases the potential benefit of out-of-order processors by continuing to serve hits while a miss is outstanding
  - ⋆ called *hit-under-miss* optimization
- Further optimization if multiple outstanding misses allowed
  - ⋆ *hit-under-miltiple-miss* or *miss-under-miss* optimization
  - ⋆ useful only if memory system can serve multiple misses
  - ⋆ recall that outstanding misses can limit achievable ILP
- In general, L1 misses possible to hide, but L2 misses extremely difficult to hide

#5: Nonblocking Caches

(To Increase Cache Bandwidth)

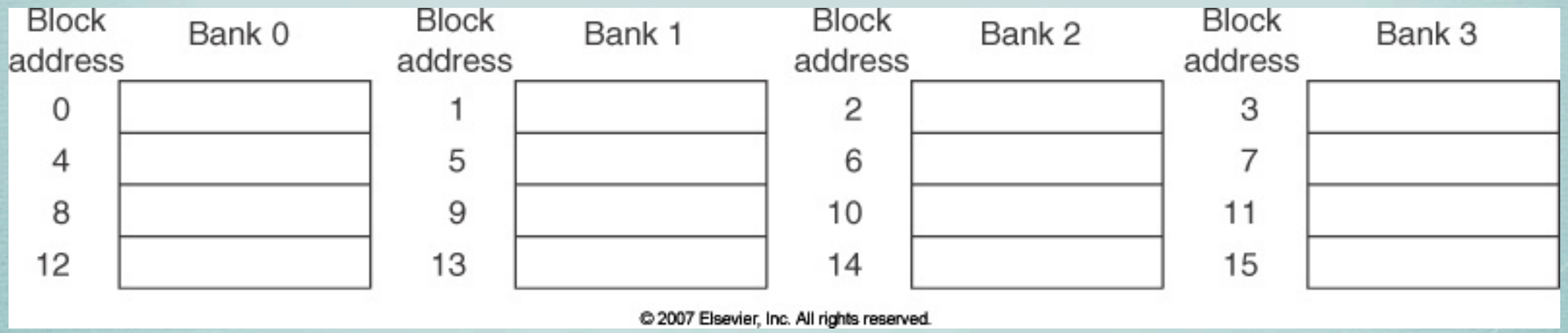


Ratio of average memory stall time for a blocking cache to hit-under-miss schemes for SPEC92 programs

#6: Multibanked Caches

(To Increase Cache Bandwidth)

- Originally used for memory, but also applicable to caches
  - ⋆ L2: Opteron has two banks, Sun Niagara has four banks
- *Sequential interleaving* works well



#7: Critical Word First and Early Restart

(To Reduce Miss Penalty)

- Observation: cache usually needs one word of the block at a time
  - ⋆ show impatience!
- Critical word first
  - ⋆ fetch the missed word from the memory first and sent it to processor as soon as it arrives
- Early restart
  - ⋆ fetch words in normal order, but send the requested word to the processor as soon as it arrives
- Useful for large block sizes

#8: Merging Write Buffers

(To Reduce Miss Penalty)

Write address	V		V		V		V	
100	1	Mem[100]	0		0		0	
108	1	Mem[108]	0		0		0	
116	1	Mem[116]	0		0		0	
124	1	Mem[124]	0		0		0	

Write address	V		V		V		V	
100	1	Mem[100]	1	Mem[108]	1	Mem[116]	1	Mem[124]
	0		0		0		0	
	0		0		0		0	
	0		0		0		0	

#8: Merging Write Buffers

(To Reduce Miss Penalty)

- Write merging
  - ⋆ used in Sun Niagara
- Helps reduce stalls due to write buffers being full
- Uses memory more efficiently
  - ⋆ multi-word writes are faster than writes performed one word at a time
- The block replaced in a cache is called the *victim*
  - ⋆ AMD Opteron calls its write buffer *victim buffer*
  - ⋆ do not confuse with *victim cache*!

#9: Compiler Optimizations: Code

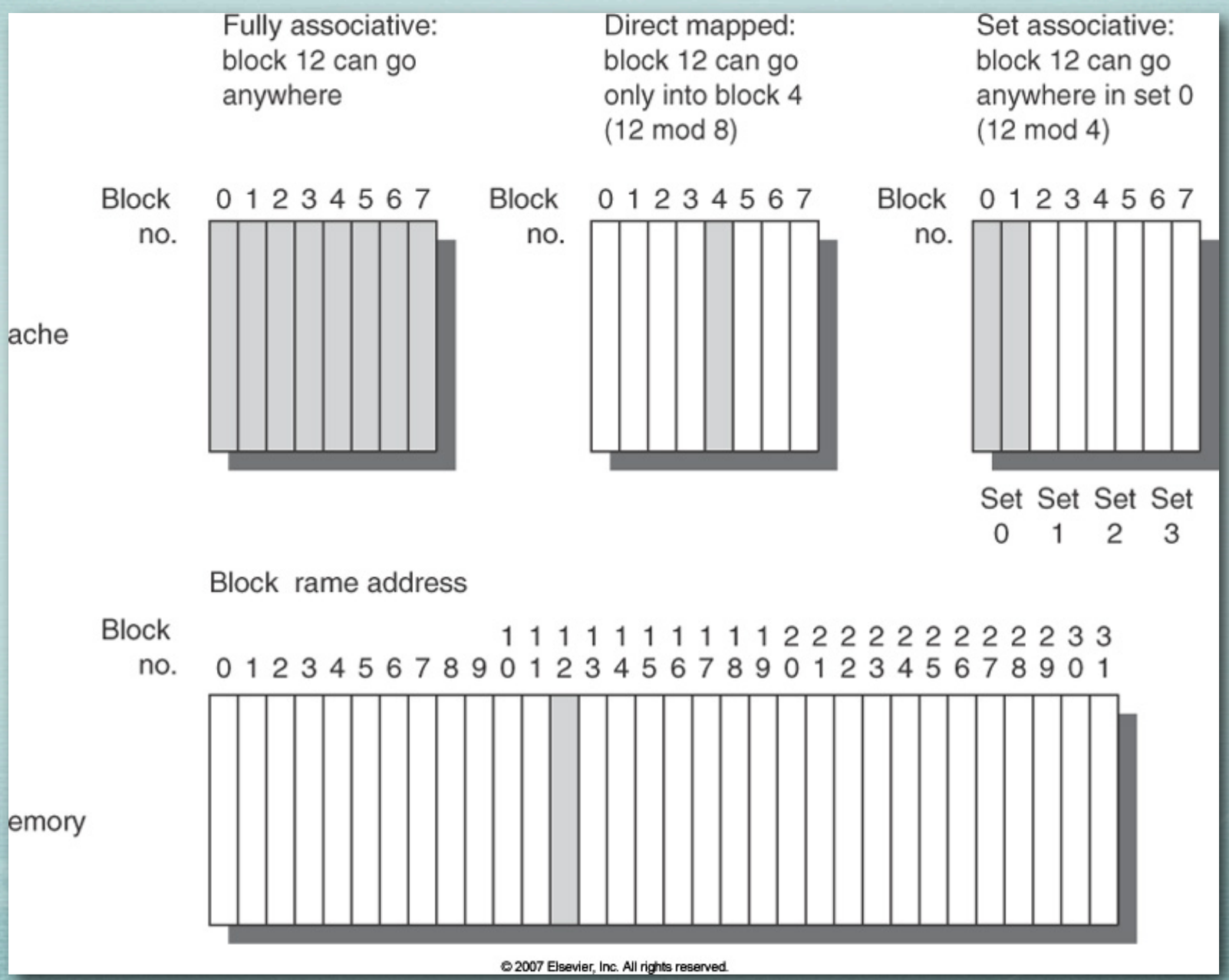
(To Reduce Miss Rate)

- Reordering procedures to reduce conflict misses
- Aligning basic blocks at cache block boundaries
- Branch straightening

#9: Compiler Optimizations: Code

(To Reduce Miss Rate)

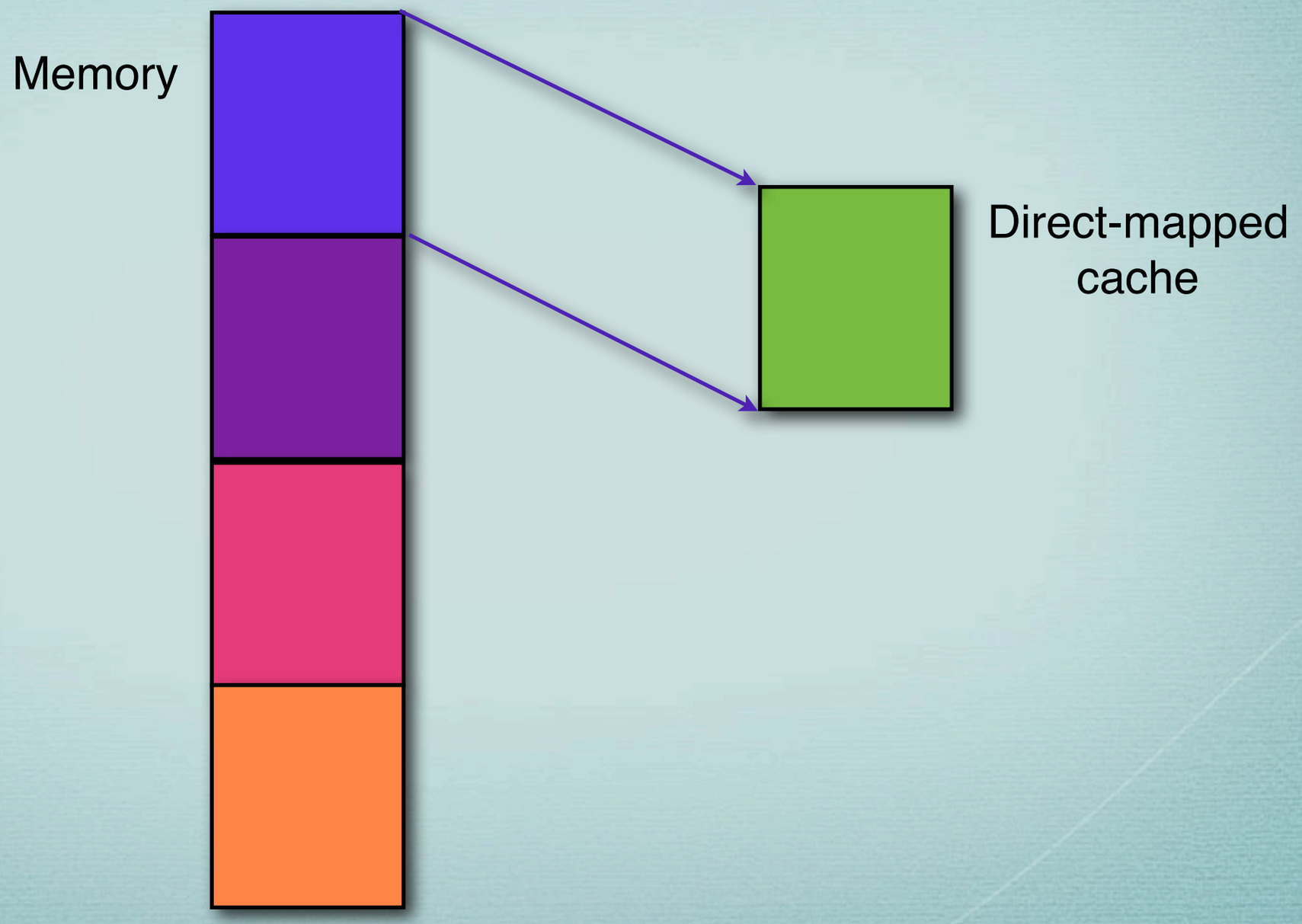
- Reordering procedures to reduce conflict misses



#9: Compiler Optimizations: Code

(To Reduce Miss Rate)

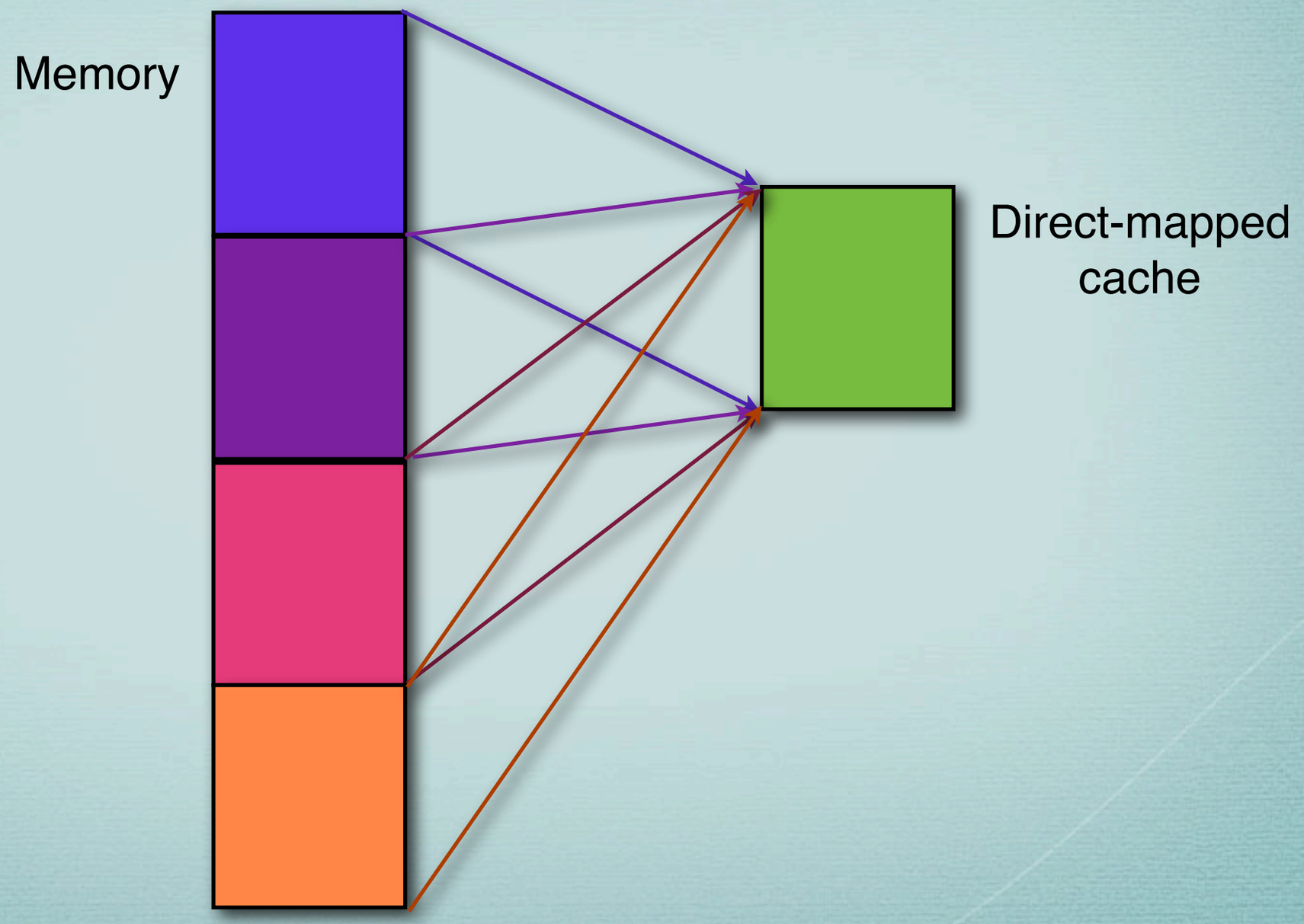
- Reordering procedures to reduce conflict misses



#9: Compiler Optimizations: Code

(To Reduce Miss Rate)

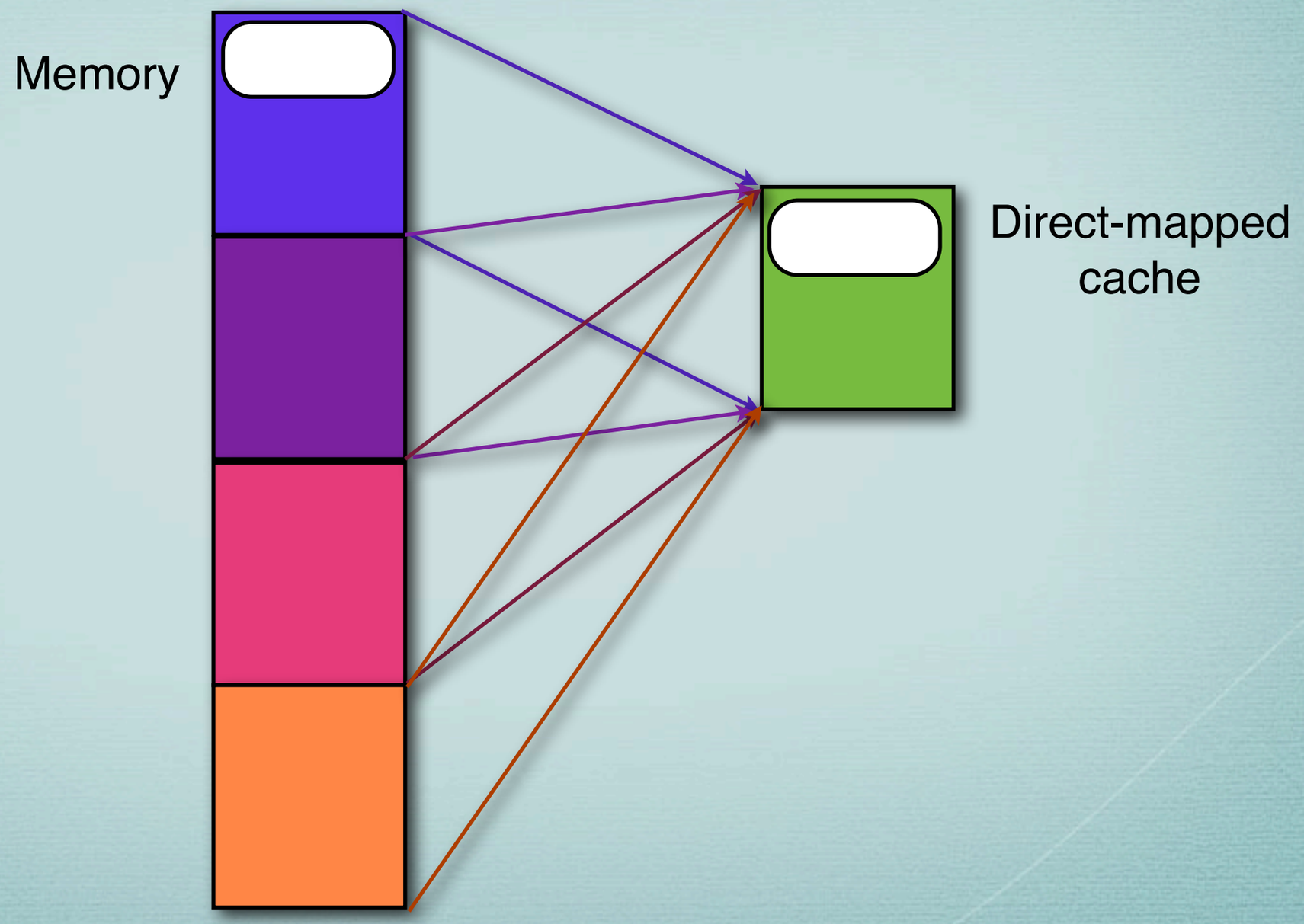
- Reordering procedures to reduce conflict misses



#9: Compiler Optimizations: Code

(To Reduce Miss Rate)

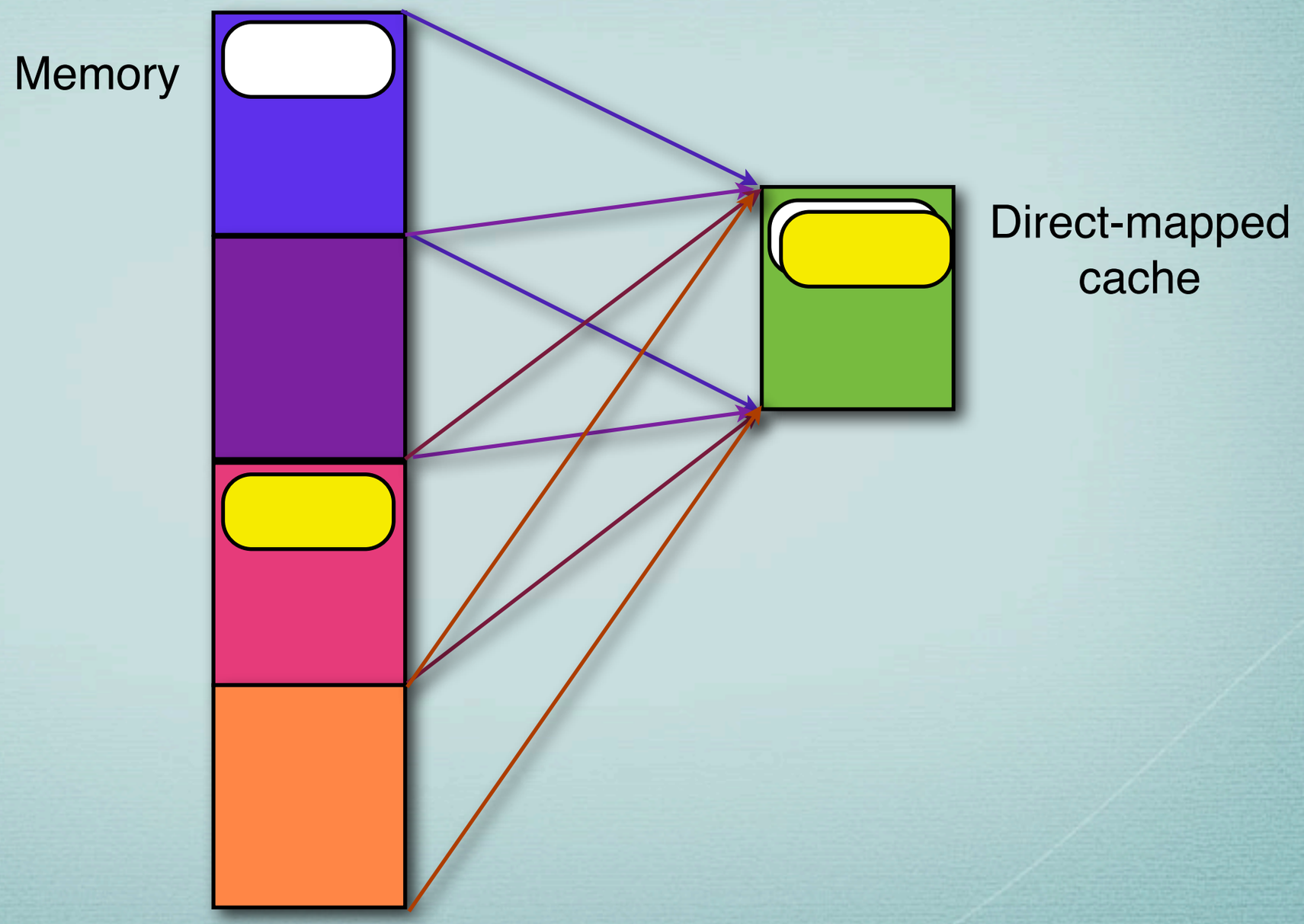
- Reordering procedures to reduce conflict misses



#9: Compiler Optimizations: Code

(To Reduce Miss Rate)

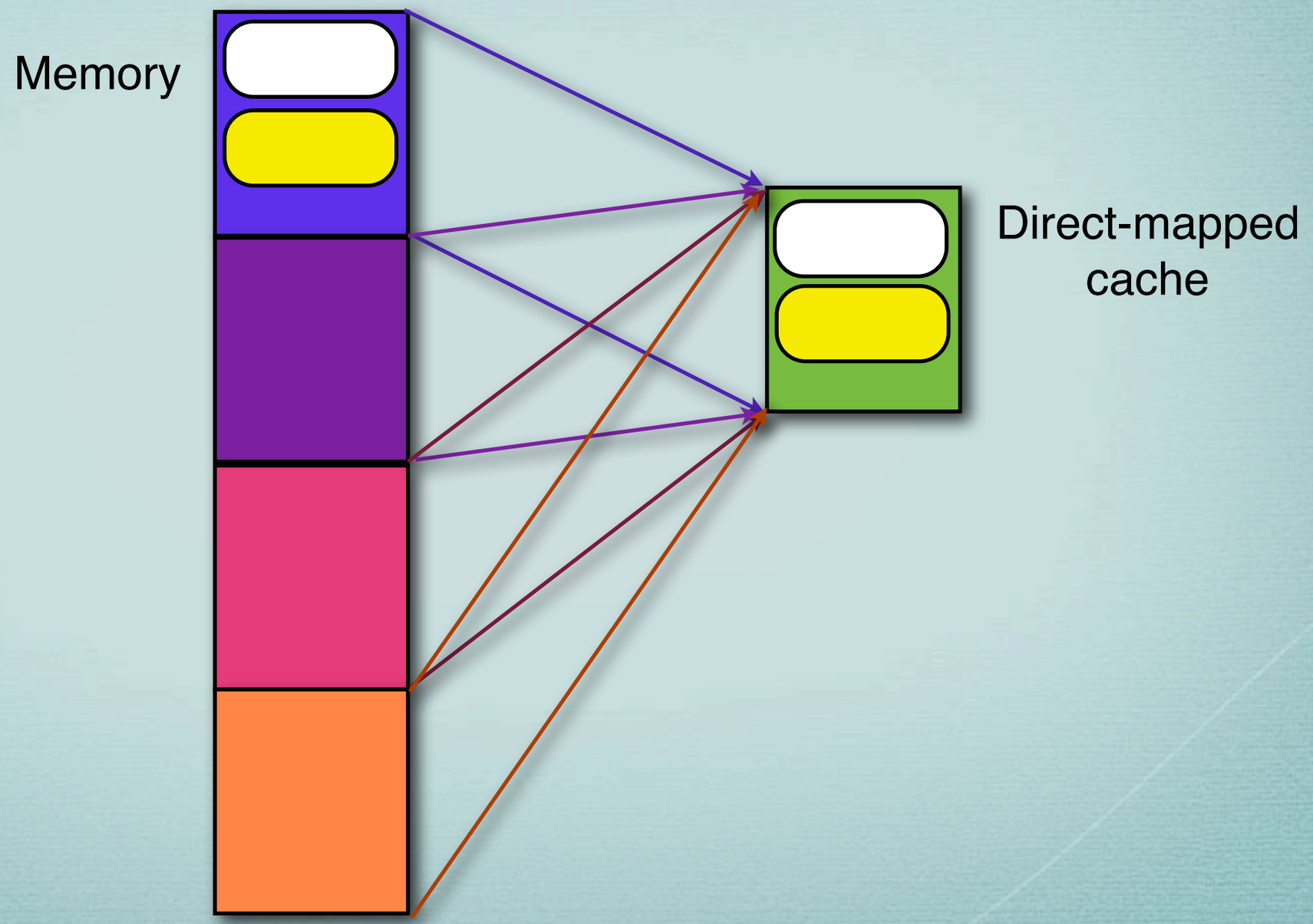
- Reordering procedures to reduce conflict misses



#9: Compiler Optimizations: Code

(To Reduce Miss Rate)

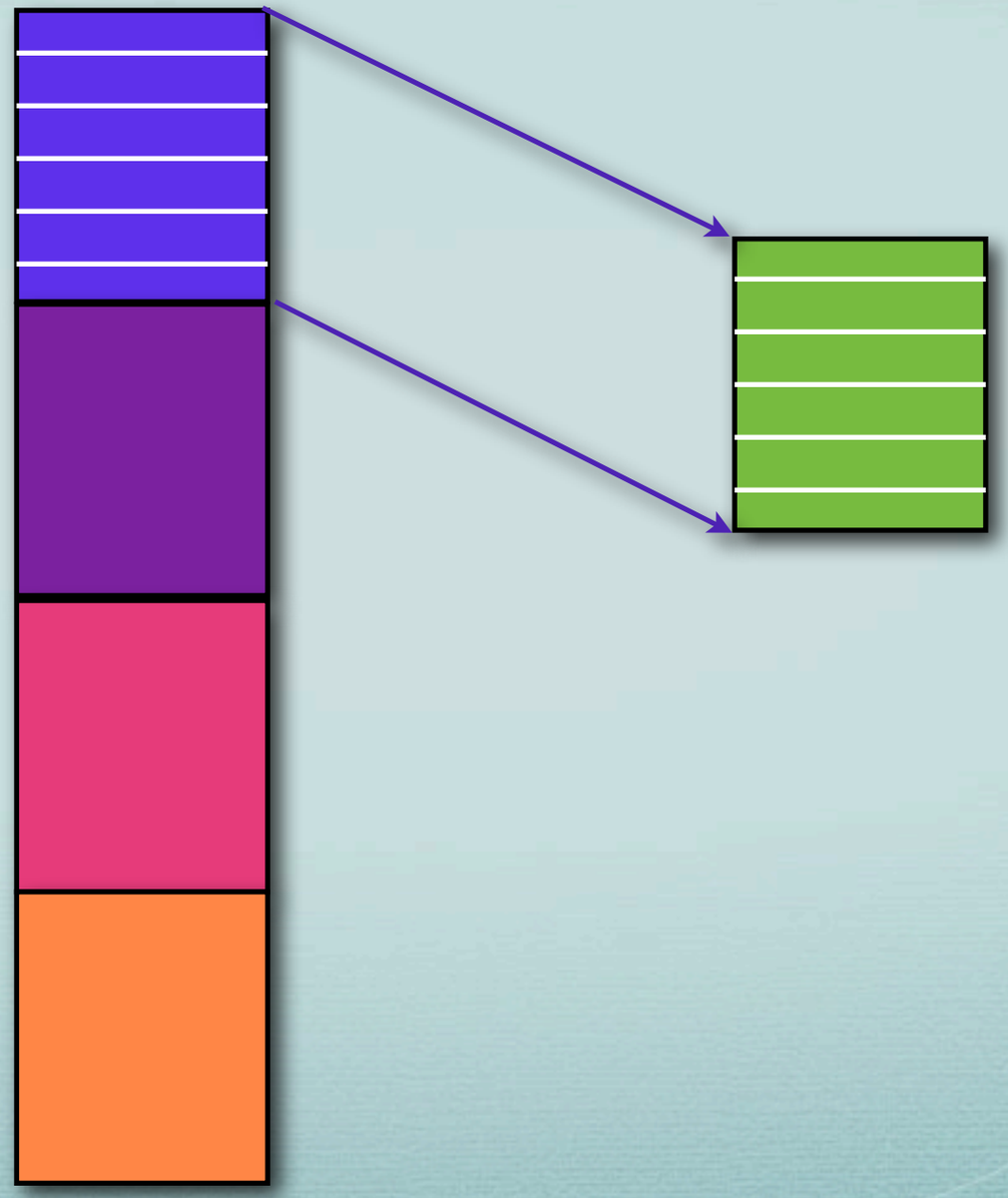
- Reordering procedures to reduce conflict misses



#9: Compiler Optimizations: Code

(To Reduce Miss Rate)

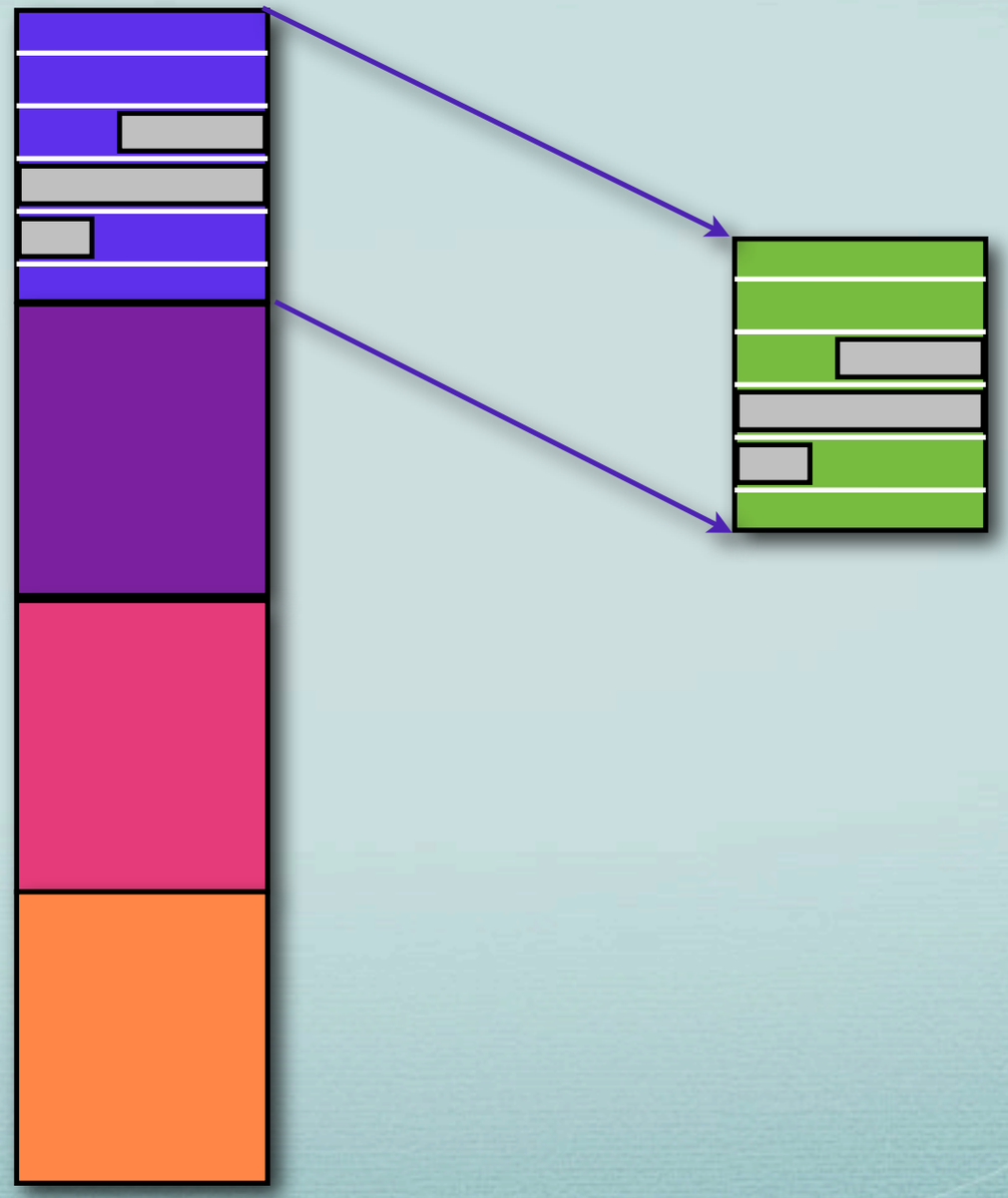
- Reordering procedures to reduce conflict misses
- Aligning basic blocks at cache block boundaries



#9: Compiler Optimizations: Code

(To Reduce Miss Rate)

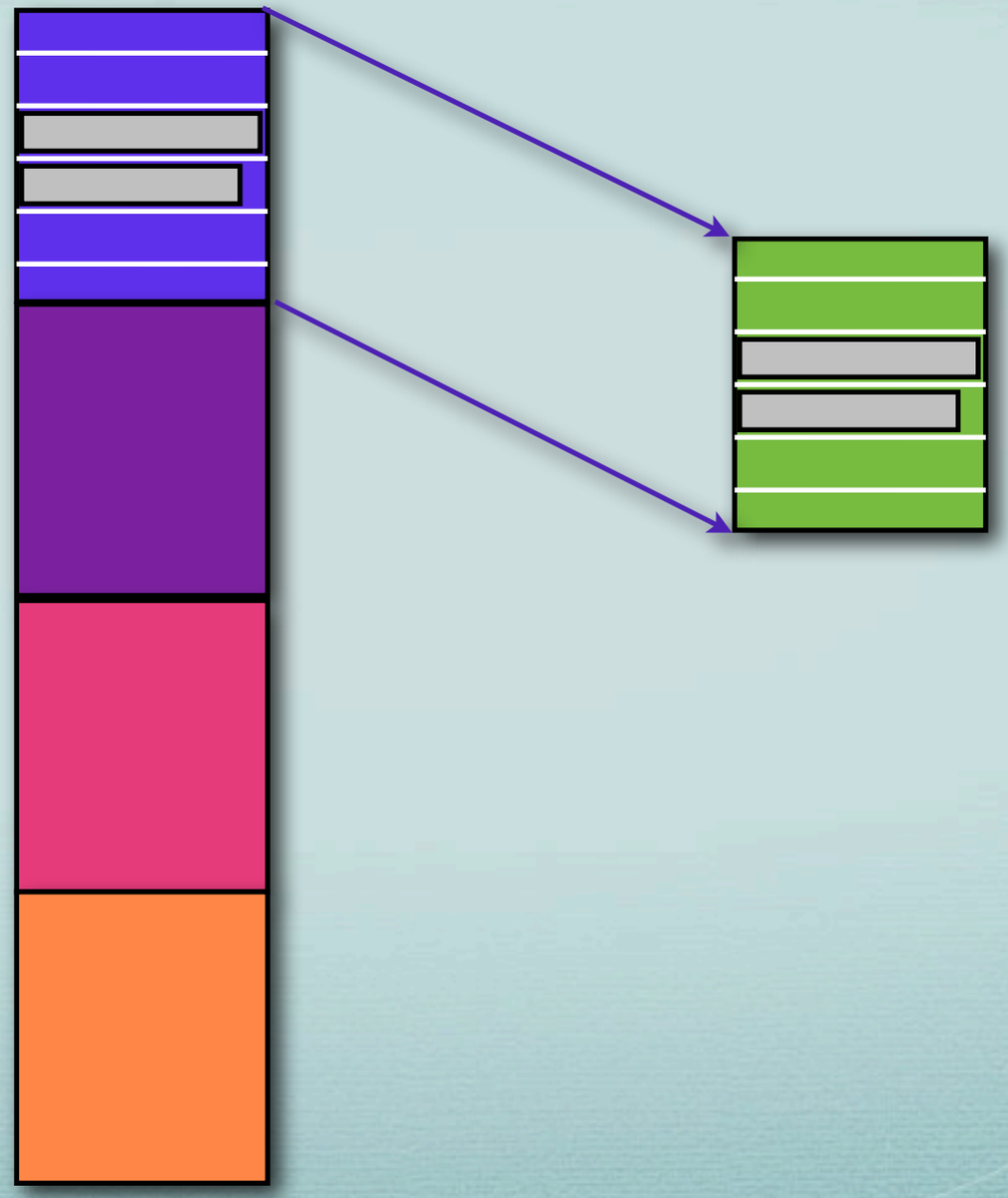
- Reordering procedures to reduce conflict misses
- Aligning basic blocks at cache block boundaries



#9: Compiler Optimizations: Code

(To Reduce Miss Rate)

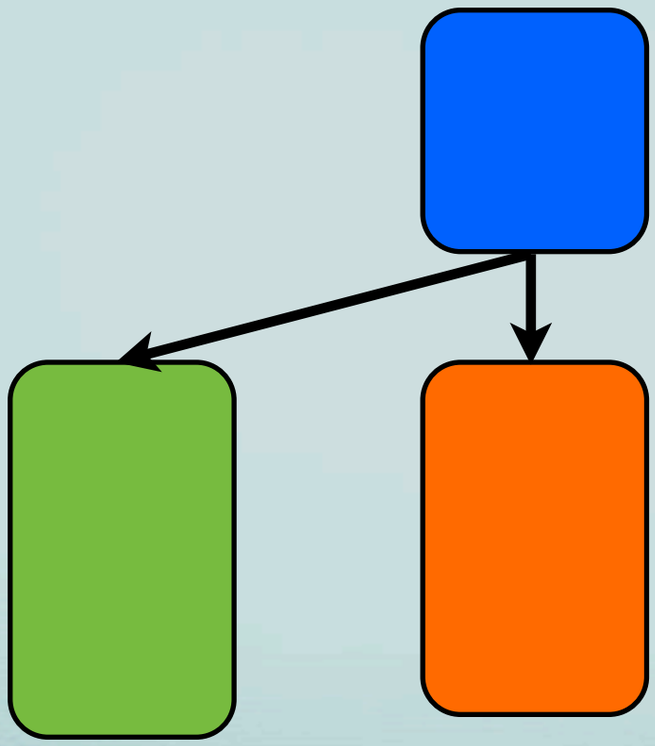
- Reordering procedures to reduce conflict misses
- Aligning basic blocks at cache block boundaries



#9: Compiler Optimizations: Code

(To Reduce Miss Rate)

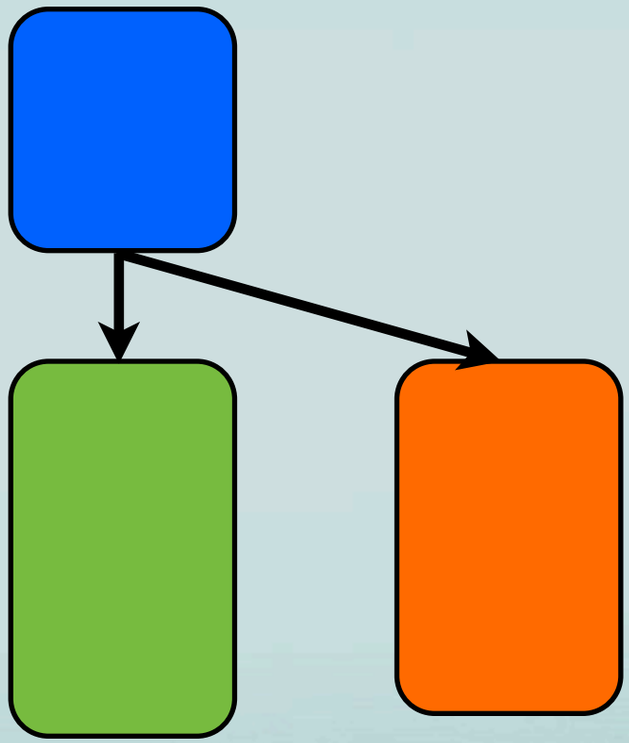
- Reordering procedures to reduce conflict misses
- Aligning basic blocks at cache block boundaries
- Branch straightening



#9: Compiler Optimizations: Code

(To Reduce Miss Rate)

- Reordering procedures to reduce conflict misses
- Aligning basic blocks at cache block boundaries
- Branch straightening



#9: Compiler Optimizations: Data

(To Reduce Miss Rate)

- Loop interchange
  - ⋆ to effectively leverage spatial locality
- Blocking
  - ⋆ to improve temporal locality

#9: Compiler Optimizations: Loop Interchange

(To Reduce Miss Rate)

```
for (j=0; j < 100; j++)
    for (i=0; i < 5000; i++)
        x[i][j] = 2*x[i][j];
```

#9: Compiler Optimizations: Loop Interchange

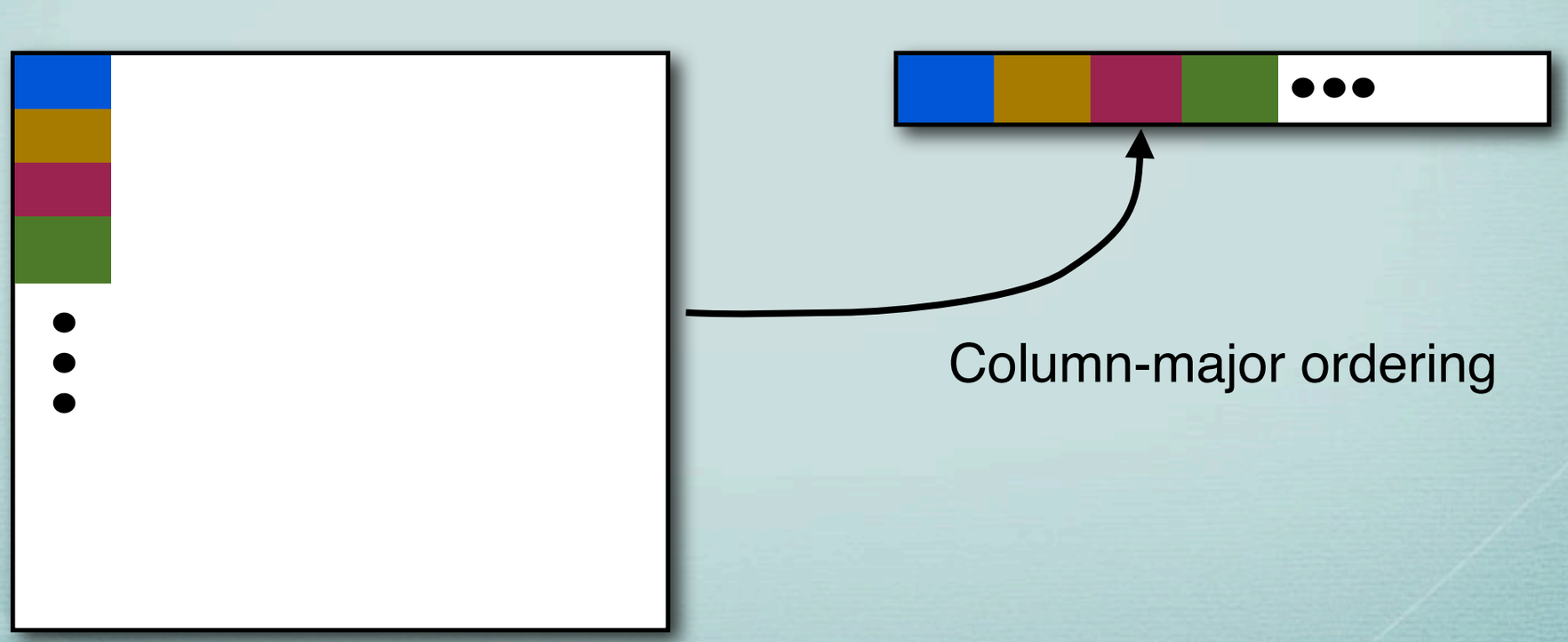
(To Reduce Miss Rate)

```
for (j=0; j < 100; j++)
    for (i=0; i < 5000; i++)
        x[i][j] = 2*x[i][j];
```

#9: Compiler Optimizations: Loop Interchange

(To Reduce Miss Rate)

```
for (j=0; j < 100; j++)
    for (i=0; i < 5000; i++)
        x[i][j] = 2*x[i][j];
```

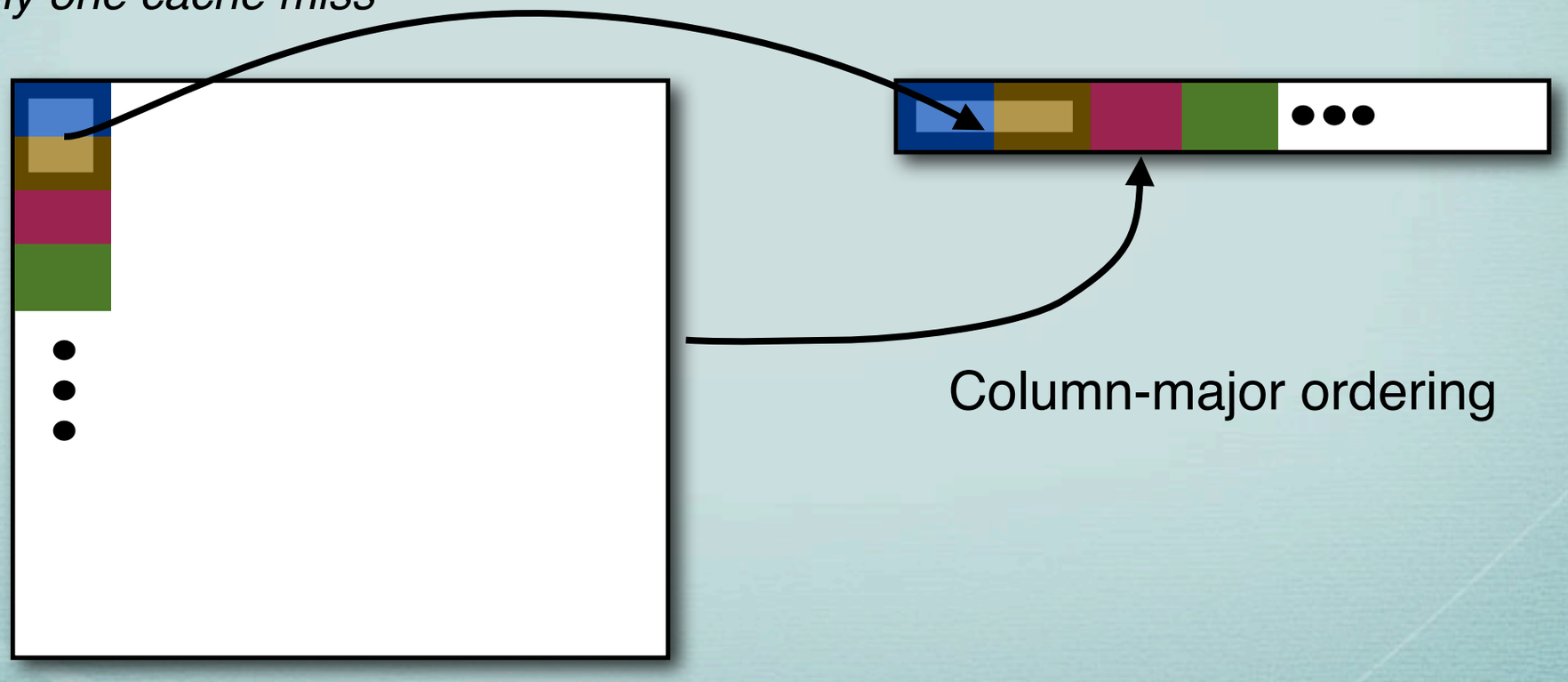


#9: Compiler Optimizations: Loop Interchange

(To Reduce Miss Rate)

```
for (j=0; j < 100; j++)
    for (i=0; i < 5000; i++)
        x[i][j] = 2*x[i][j];
```

Only one cache miss

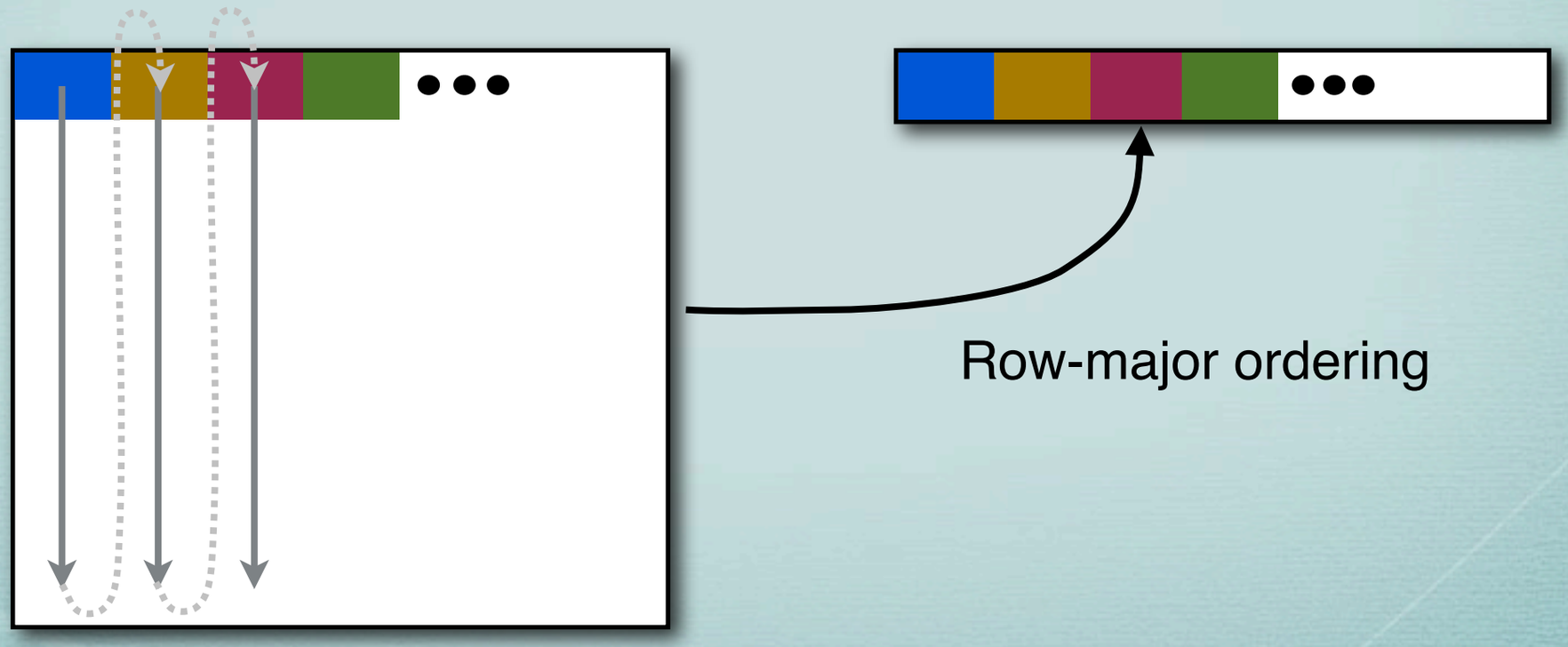


Column-major ordering

#9: Compiler Optimizations: Loop Interchange

(To Reduce Miss Rate)

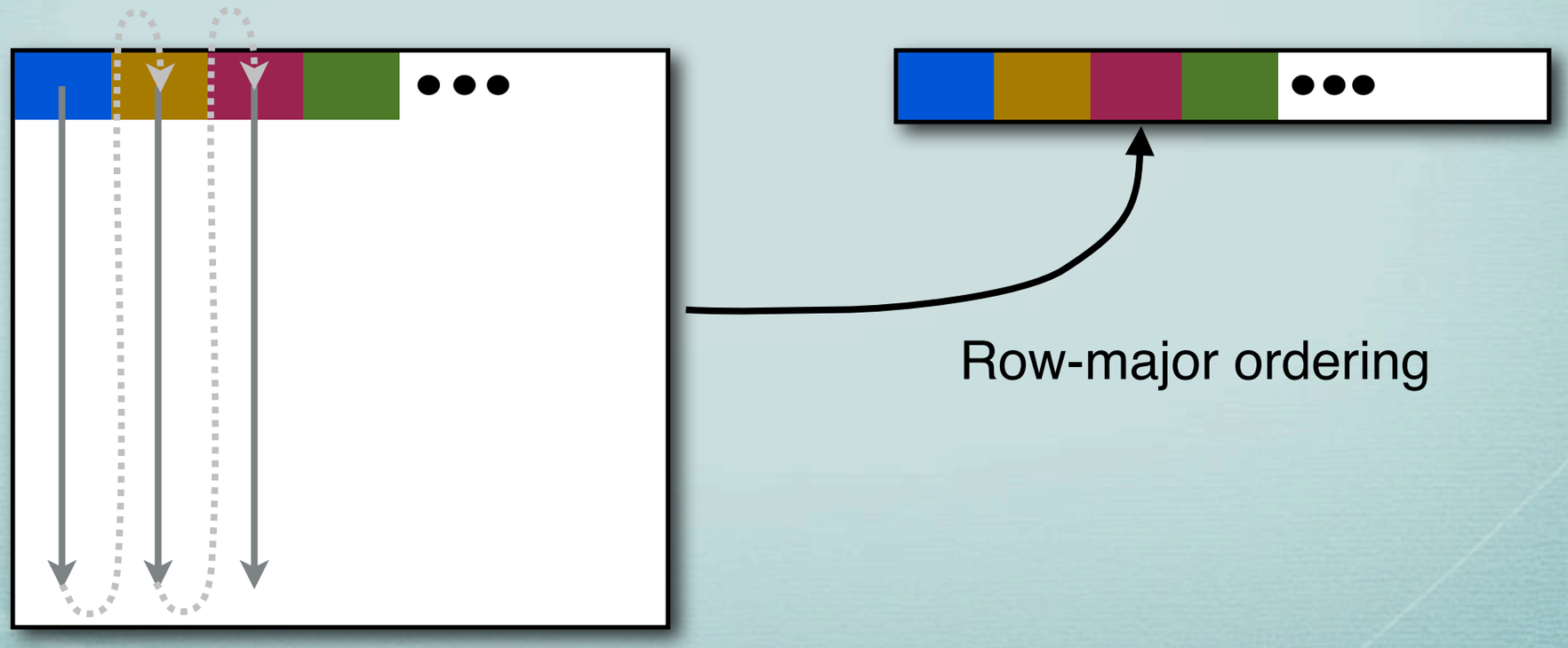
```
for (j=0; j < 100; j++)
    for (i=0; i < 5000; i++)
        x[i][j] = 2*x[i][j];
```



#9: Compiler Optimizations: Loop Interchange

(To Reduce Miss Rate)

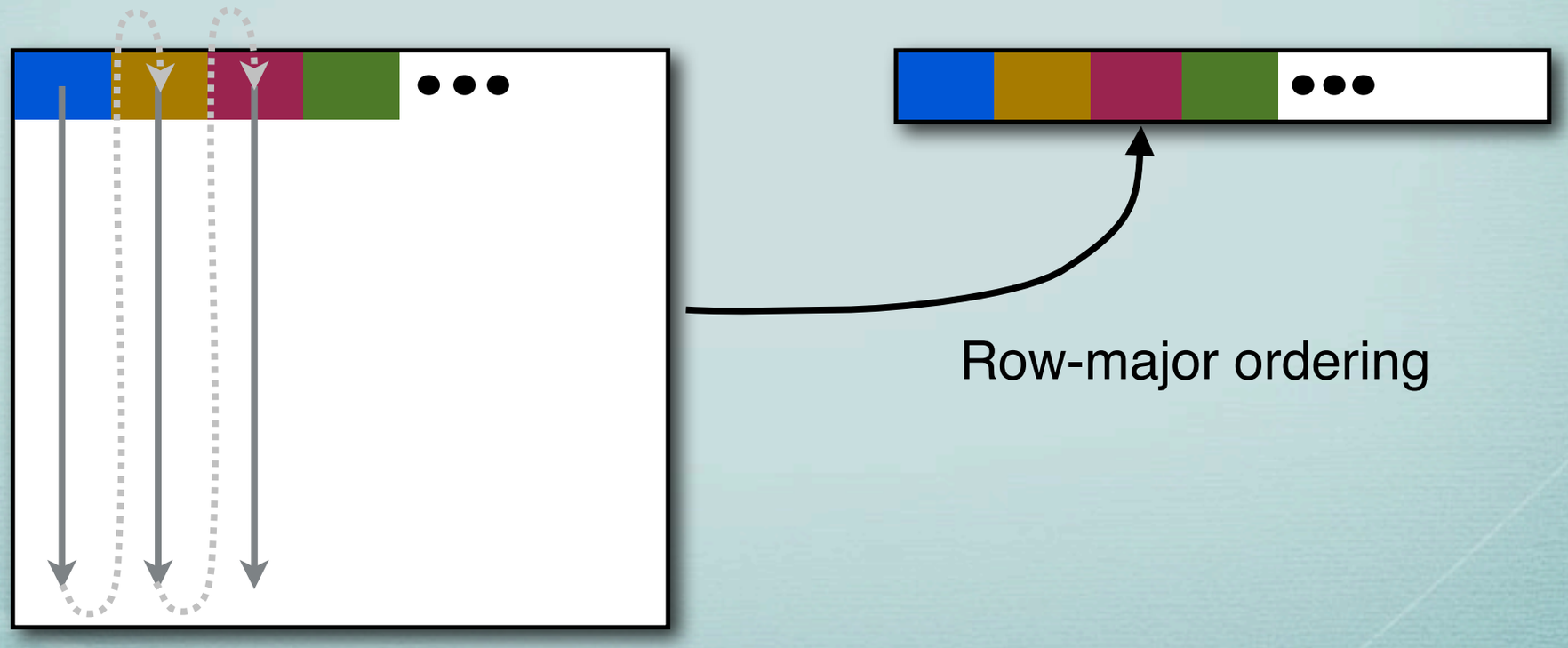
```
for (j=0; j < 100; j++)
    for (i=0; i < 5000; i++)
        x[i][j] = 2*x[i][j];
```



#9: Compiler Optimizations: Loop Interchange

(To Reduce Miss Rate)

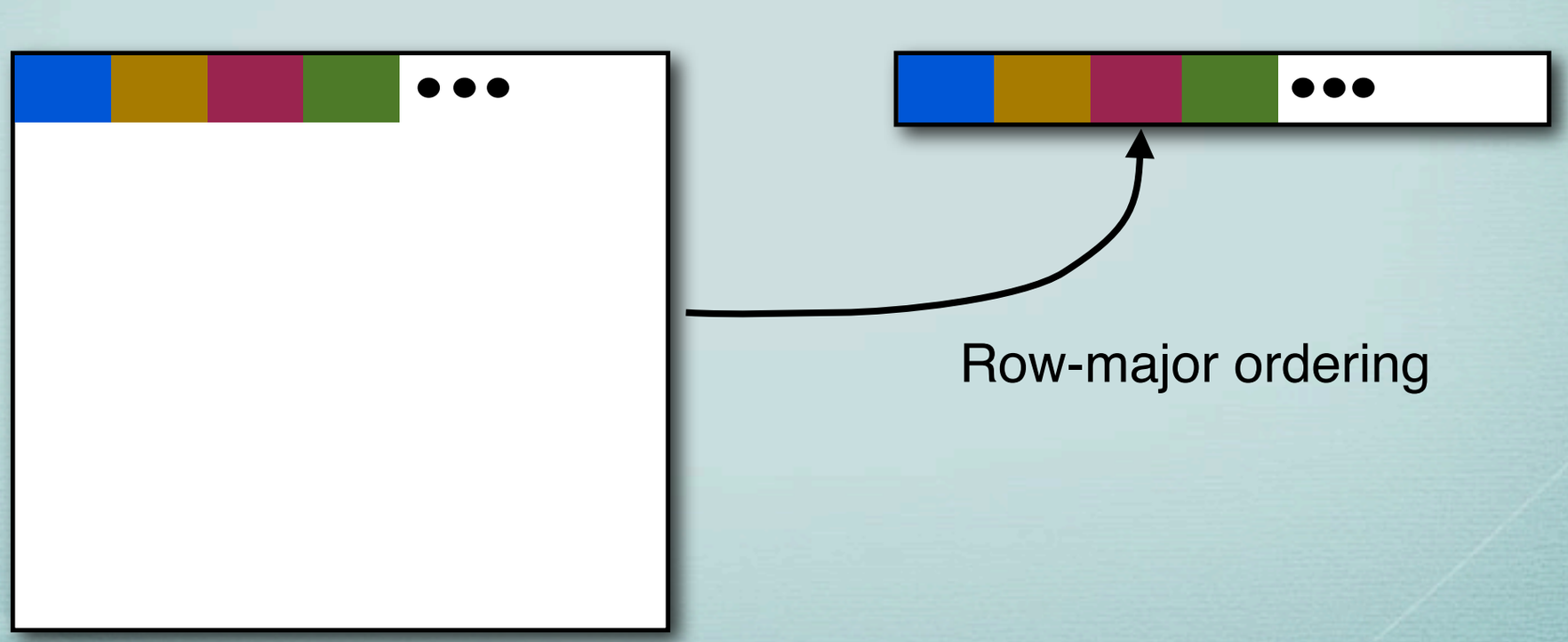
```
for (j=0; j < 100; j++)
    for (i=0; i < 5000; i++)
        x[i][j] = 2*x[i][j];
```



#9: Compiler Optimizations: Loop Interchange

(To Reduce Miss Rate)

```
for (i=0; i < 5000; i++)
    for (j=0; j < 100; j++)
        x[i][j] = 2*x[i][j];
```



#9: Compiler Optimizations: Blocking

(To Reduce Miss Rate)

```
for (i=0; i < N; i++)
    for (j=0; j < N; j++)
    {
     r = 0.0;
     for (k=0; k < N; k++)
         r = r + y[i][k]*z[k][j];
     x[i][j] = r;
    }
```

#9: Compiler Optimizations: Blocking

(To Reduce Miss Rate)

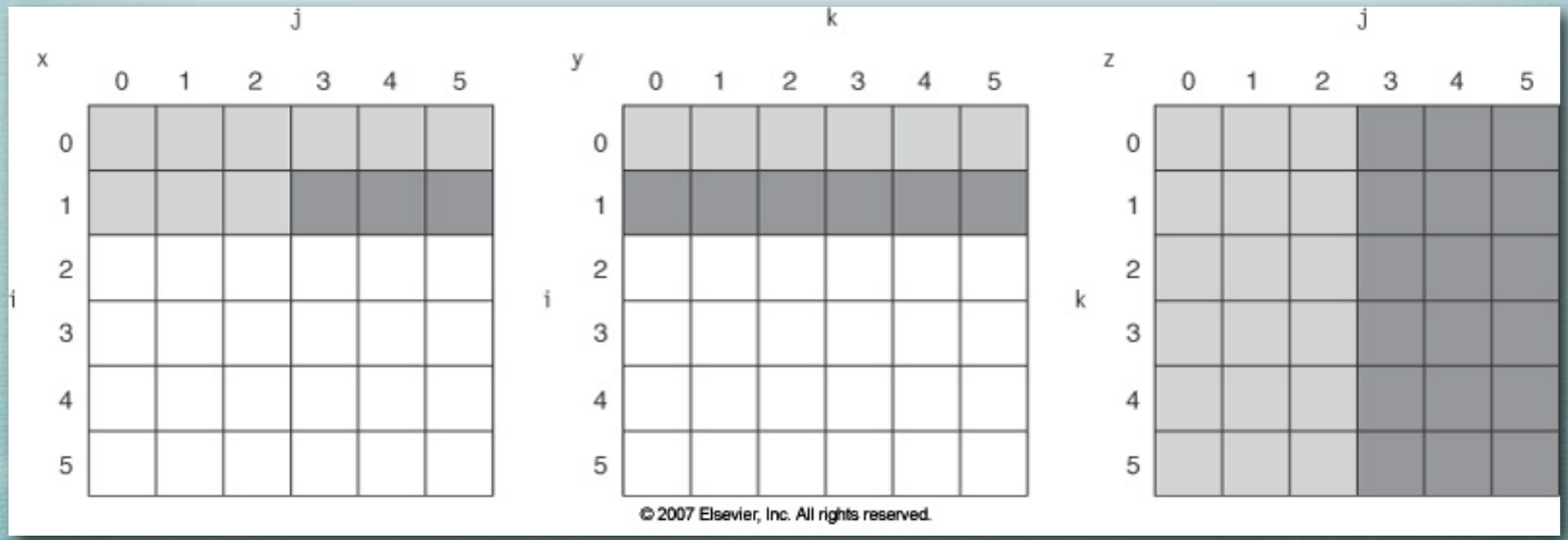
```
for (i=0; i < N; i++)
    for (j=0; j < N; j++)
    {
     r = 0.0;
     for (k=0; k < N; k++)
         r = r + y[i][k]*z[k][j];
     x[i][j] = r;
    }
```

Accesses = anywhere between $3N^2$ and $(2N^3+N^2)$

#9: Compiler Optimizations: Blocking

(To Reduce Miss Rate)

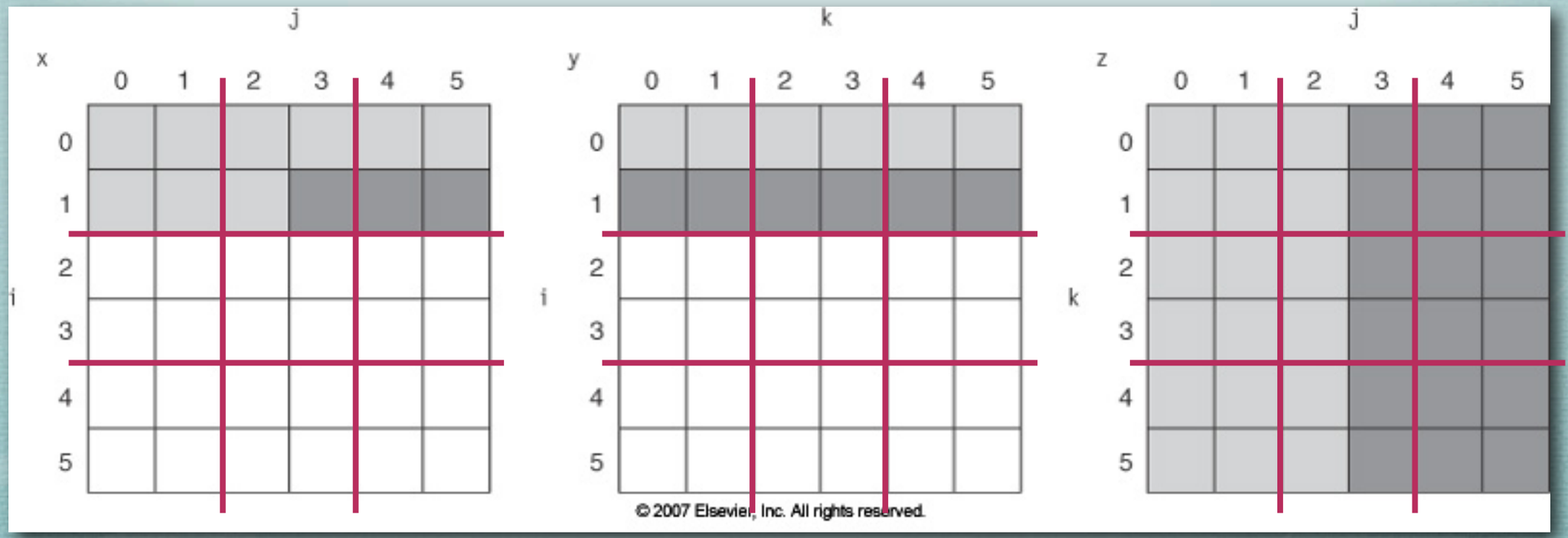
```
for (i=0; i < N; i++)
    for (j=0; j < N; j++)
    {
     r = 0.0;
     for (k=0; k < N; k++)
         r = r + y[i][k]*z[k][j];
     x[i][j] = r;
    }
```



#9: Compiler Optimizations: Blocking

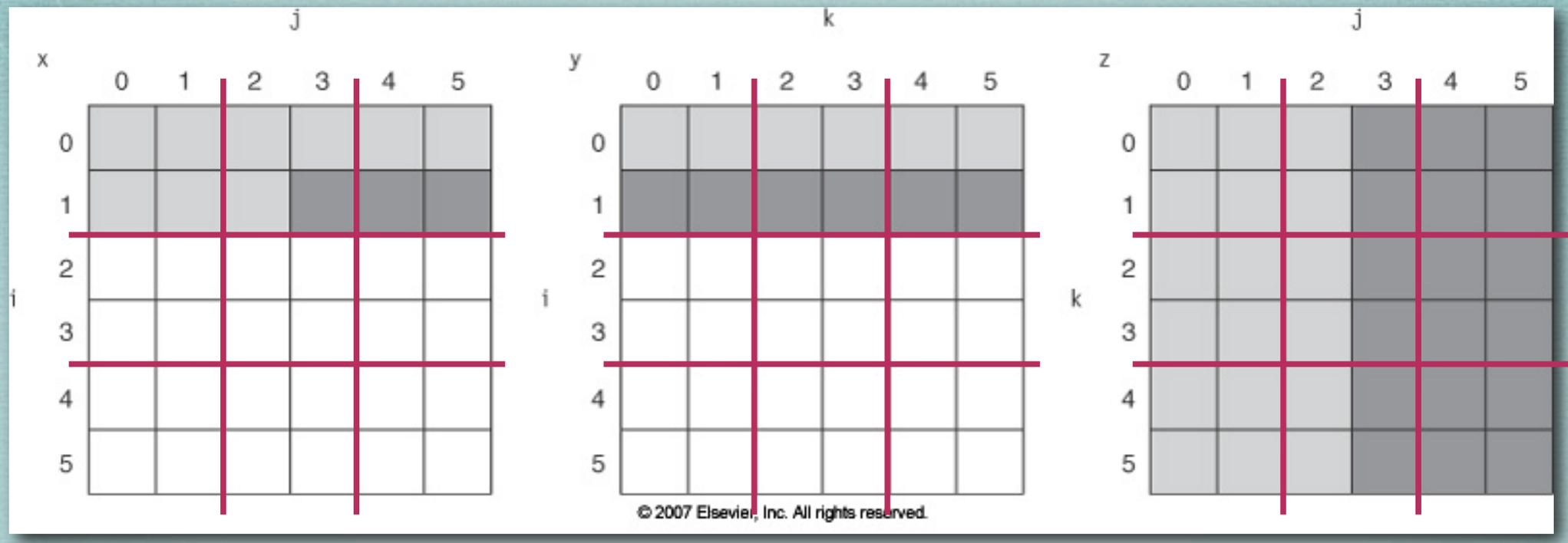
(To Reduce Miss Rate)

```
for (i=0; i < N; i++)
    for (j=0; j < N; j++)
    {
     r = 0.0;
     for (k=0; k < N; k++)
         r = r + y[i][k]*z[k][j];
     x[i][j] = r;
    }
```



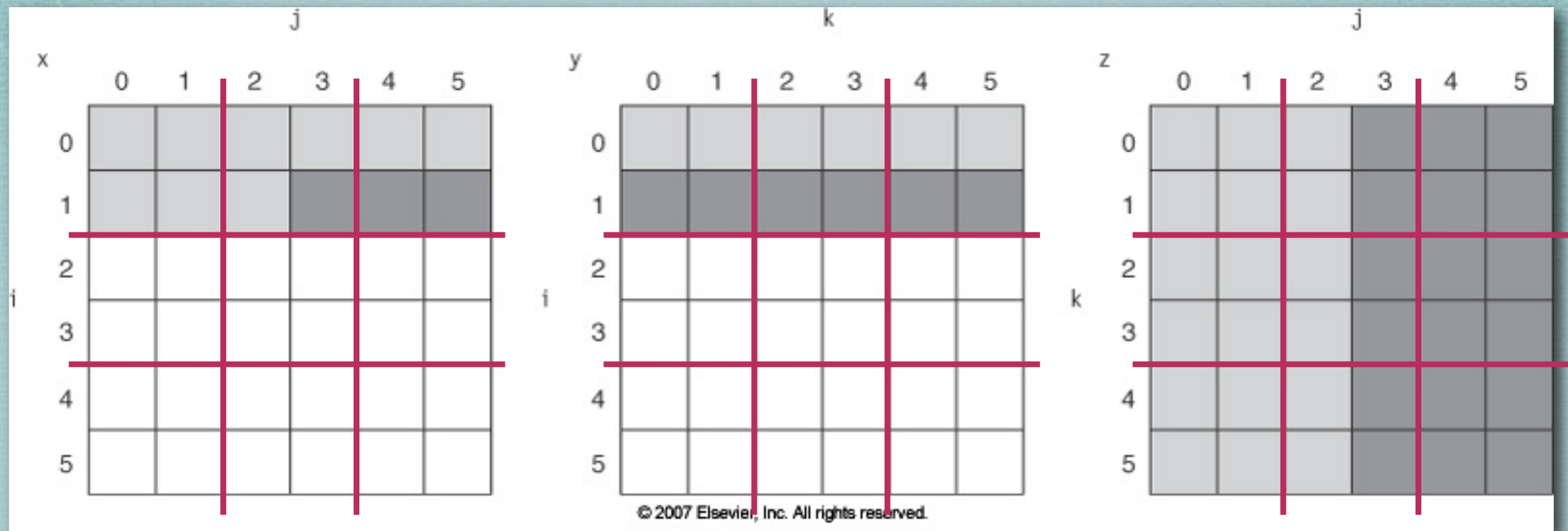
#9: Compiler Optimizations: Blocking

(To Reduce Miss Rate)



#9: Compiler Optimizations: Blocking

(To Reduce Miss Rate)




```
for (ii=0; ii < N/B; ii++)
    for (jj=0; jj < N/B; jj++)
    {
     R = ZEROS(B,B);
     for (kk=0; kk < N/B; kk++)
         R = R ⊞ Y[ii][kk]⊠Z[kk][jj];
     X[ii][jj] = R;
    }
```

#9: Compiler Optimizations: Blocking

(To Reduce Miss Rate)

```
for (ii=0; ii < N/B; ii++)
   for (jj=0; jj < N/B; j++)
   {
    R = ZEROS(B,B);
    for (kk=0; kk < N/B; kk++)
       R = R ⊞ Y[ii][kk]⊠Z[kk][jj];
    X[ii][jj] = R;
   }
```

```
for (ii=0; ii < N/B; ii++)
   for (jj=0; jj < N/B; jj++)
   {
    R = ZEROS(B,B);
    for (kk=0; kk < N/B; kk++)
    {
     for (i=ii; i < ii+B; i++)
        for (j=jj; j < jj+B; j++)
           for (k=kk; k < kk+B; k++)
              R[i][j] = R[i][j] + y[i][k]*z[k][j]
    }
    X[ii][jj] = R;
  }
```

#9: Compiler Optimizations: Blocking

(To Reduce Miss Rate)

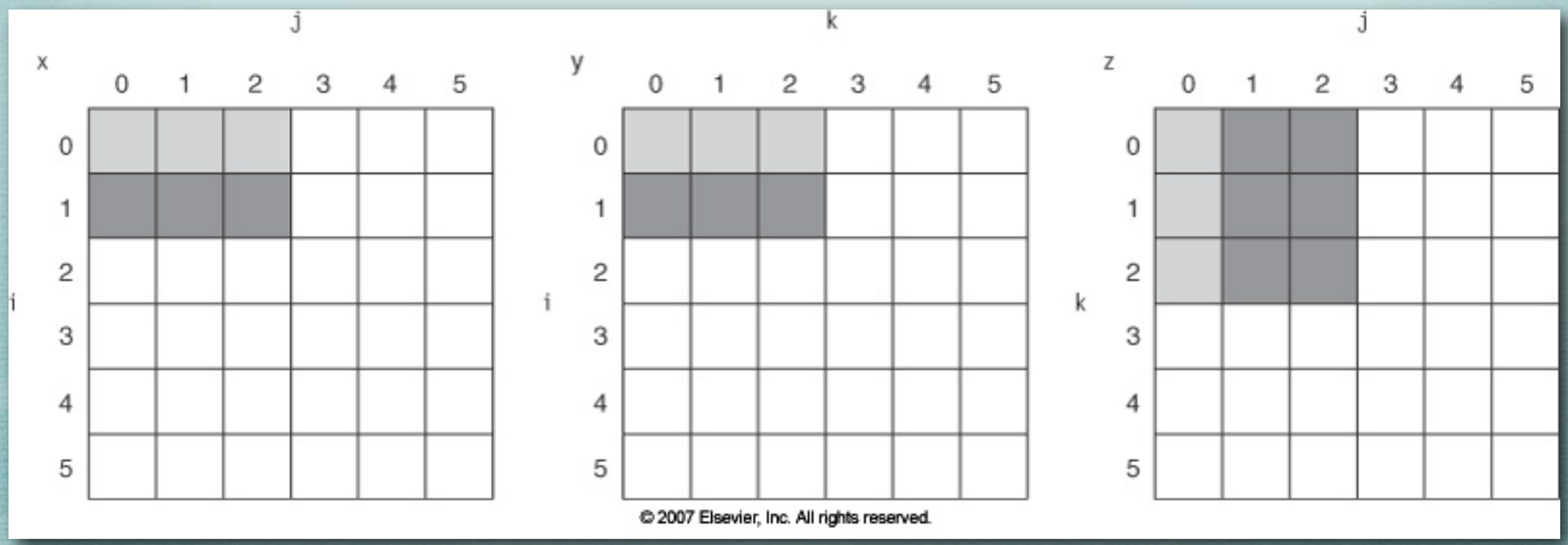
```
for (ii=0; ii < N/B; ii++)
   for (jj=0; jj < N/B; jj++)
   {
    R = ZEROS(B,B);
    for (kk=0; kk < N/B; kk++)
    {
     for (i=ii; i < ii+B; i++)
        for (j=jj; j < jj+B; j++)
           for (k=kk; k < kk+B; k++)
              R[i][j] = R[i][j] + y[i][k]*z[k][j]
    }
    X[ii][jj] = R;
   }
```

```
for (ii=0; ii < N/B; ii++)
   for (jj=0; jj < N/B; jj++)
      for (kk=0; kk < N/B; kk++)
         for (i=ii; i < ii+B; i++)
            for (j=jj; j < jj+B; j++)
            {
             r = 0.0;
             for (k=kk; k < kk+B; k++)
                r = r + y[i][k]*z[k][j];
             x[i][j] = r;
            }
```

#9: Compiler Optimizations: Blocking

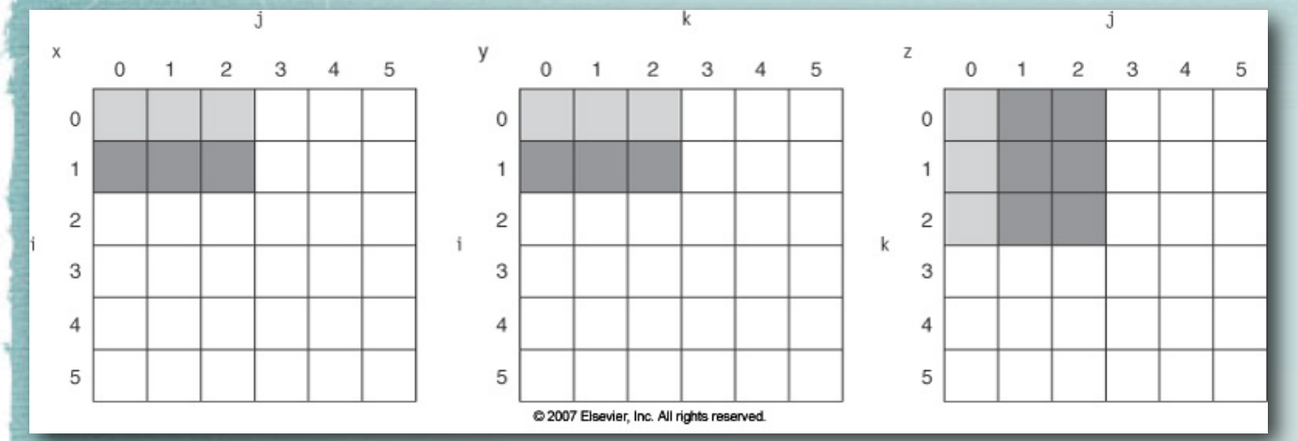
(To Reduce Miss Rate)

```
for (jj=0; jj < N; jj = jj+B)
    for (kk=0; kk < N; kk = kk+B)
        for (i=0; i < N; i++)
            for (j=jj; j < min(jj+B,N); j++)
            {
             r = 0.0;
             for (k=kk; k < min(kk+B,N); k++)
                 r = r + y[i][k]*z[k][j];
             x[i][j] = x[i][j]+r;
            }
```



#9: Compiler Optimizations: Blocking

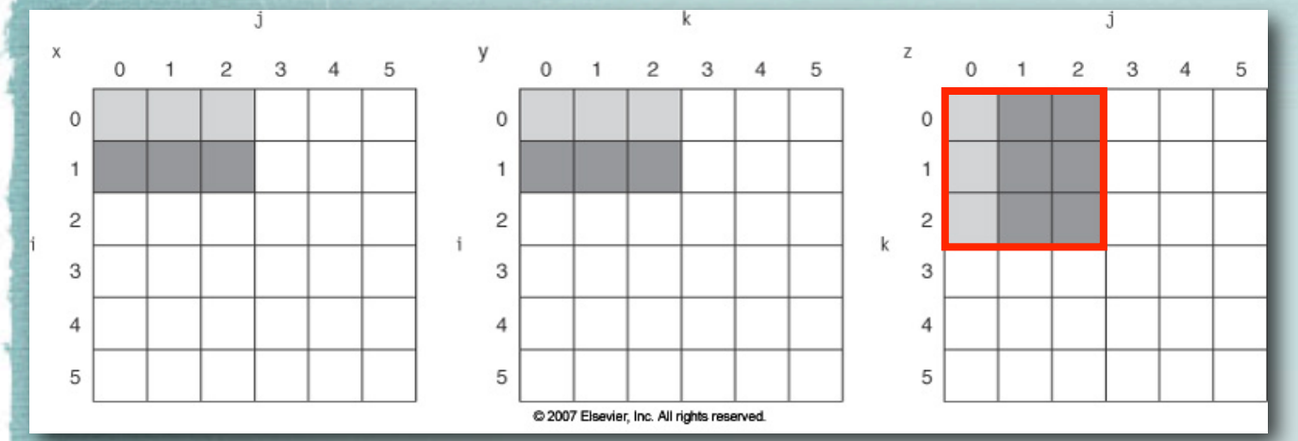
(To Reduce Miss Rate)




```
for (jj=0; jj < N; jj = jj+B)
   for (kk=0; kk < N; kk = kk+B)
      for (i=0; i < N; i++)
         for (j=jj; j < min(jj+B,N); j++)
         {
          r = 0.0;
          for (k=kk; k < min(kk+B,N); k++)
             r = r + y[i][k]*z[k][j];
          x[i][j] = x[i][j]+r;
         }
```

#9: Compiler Optimizations: Blocking

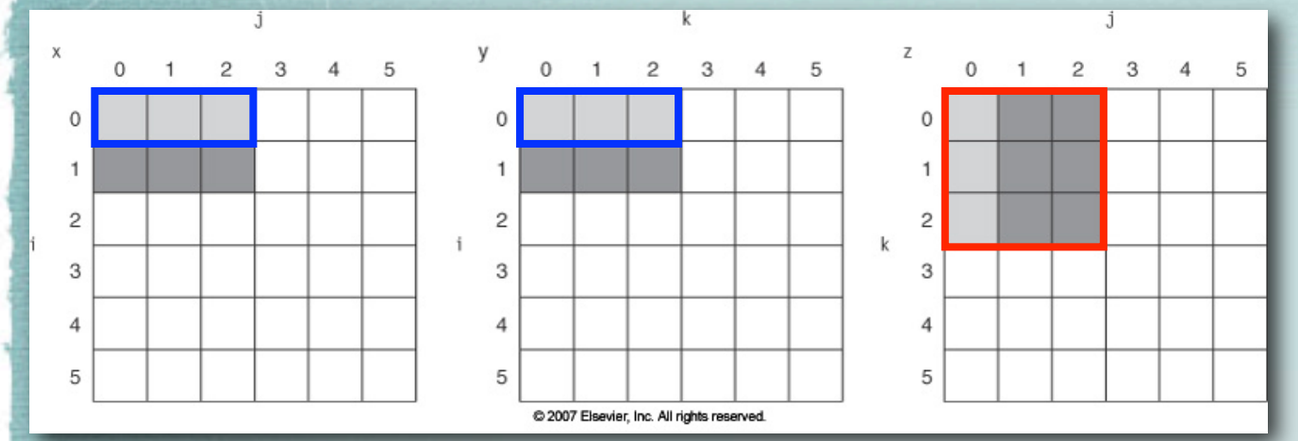
(To Reduce Miss Rate)




```
for (jj=0; jj < N; jj = jj+B)
   for (kk=0; kk < N; kk = kk+B)
      for (i=0; i < N; i++)
         for (j=jj; j < min(jj+B,N); j++)
         {
          r = 0.0;
          for (k=kk; k < min(kk+B,N); k++)
             r = r + y[i][k]*z[k][j];
          x[i][j] = x[i][j]+r;
         }
```

#9: Compiler Optimizations: Blocking

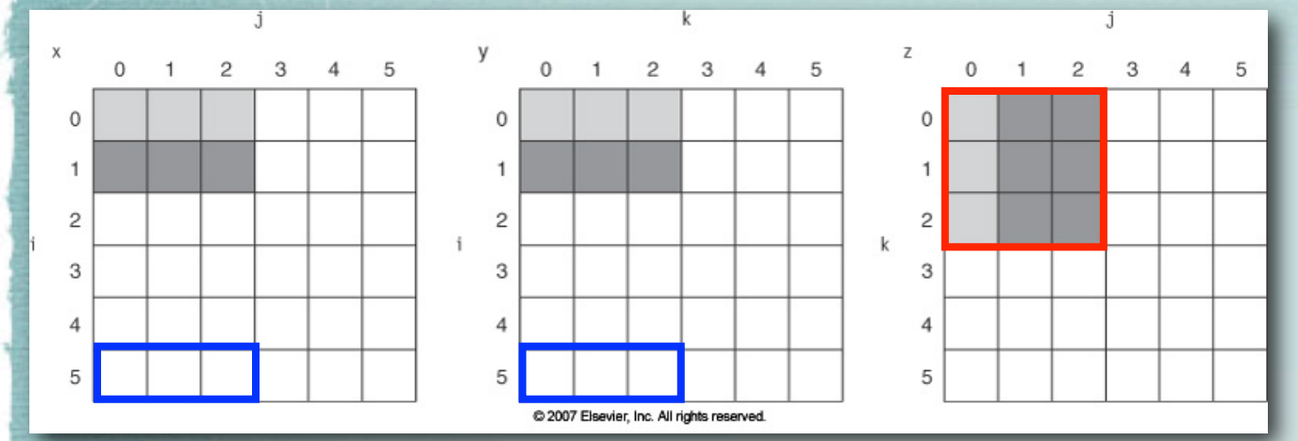
(To Reduce Miss Rate)




```
for (jj=0; jj < N; jj = jj+B)
   for (kk=0; kk < N; kk = kk+B)
      for (i=0; i < N; i++)
         for (j=jj; j < min(jj+B,N); j++)
         {
          r = 0.0;
          for (k=kk; k < min(kk+B,N); k++)
             r = r + y[i][k]*z[k][j];
          x[i][j] = x[i][j]+r;
         }
```

#9: Compiler Optimizations: Blocking

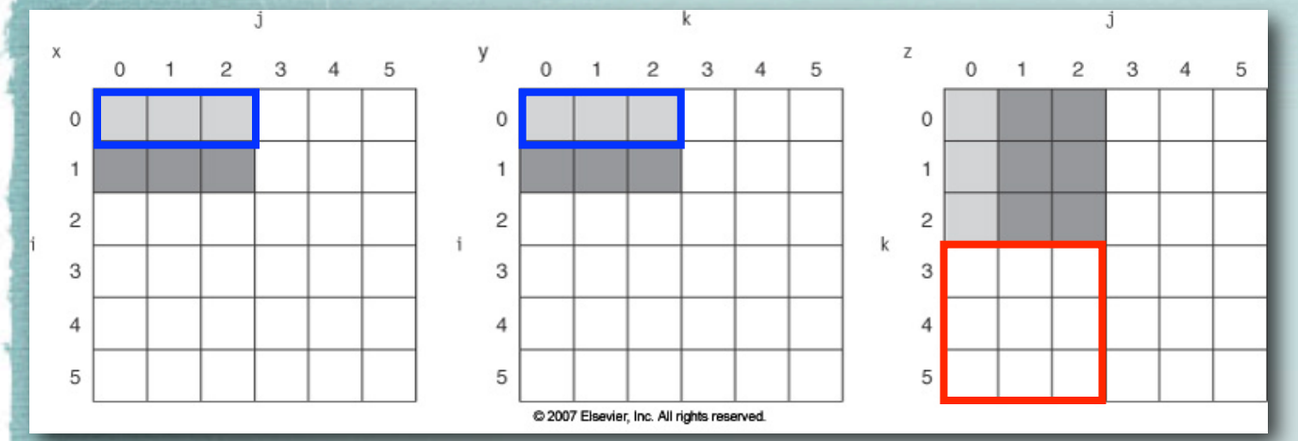
(To Reduce Miss Rate)




```
for (jj=0; jj < N; jj = jj+B)
   for (kk=0; kk < N; kk = kk+B)
      for (i=0; i < N; i++)
         for (j=jj; j < min(jj+B,N); j++)
         {
          r = 0.0;
          for (k=kk; k < min(kk+B,N); k++)
             r = r + y[i][k]*z[k][j];
          x[i][j] = x[i][j]+r;
         }
```

#9: Compiler Optimizations: Blocking

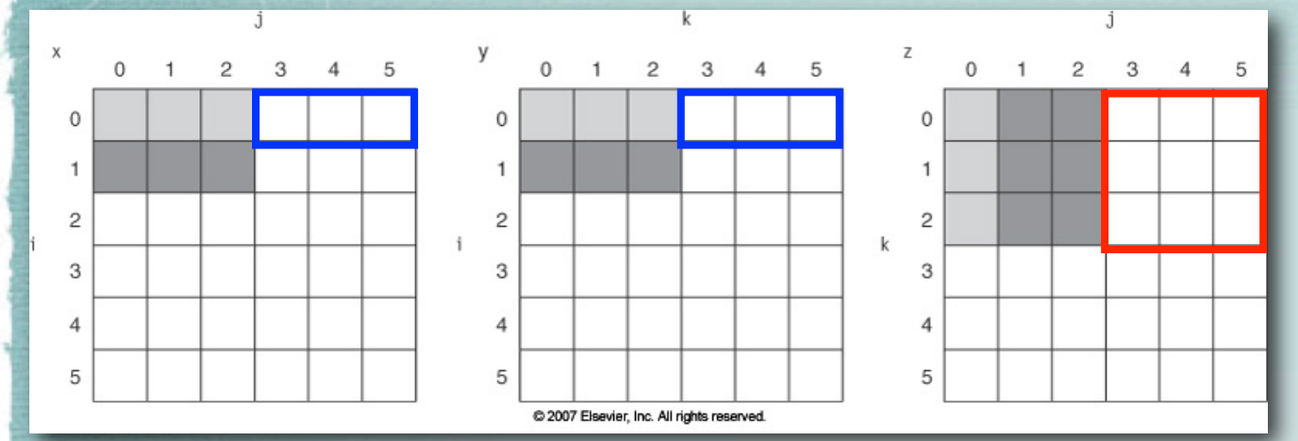
(To Reduce Miss Rate)




```
for (jj=0; jj < N; jj = jj+B)
   for (kk=0; kk < N; kk = kk+B)
      for (i=0; i < N; i++)
         for (j=jj; j < min(jj+B,N); j++)
         {
          r = 0.0;
          for (k=kk; k < min(kk+B,N); k++)
             r = r + y[i][k]*z[k][j];
          x[i][j] = x[i][j]+r;
         }
```

#9: Compiler Optimizations: Blocking

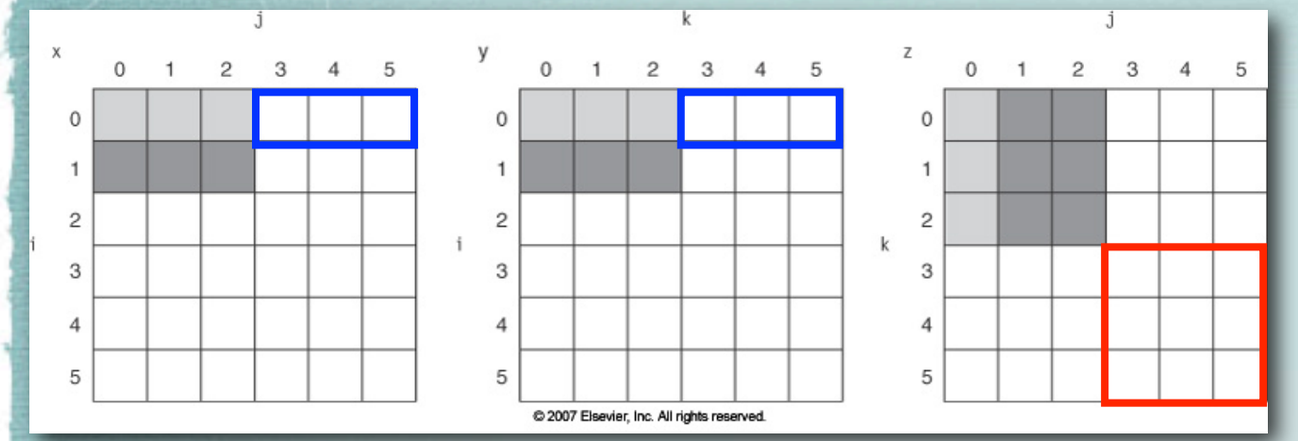
(To Reduce Miss Rate)




```
for (jj=0; jj < N; jj = jj+B)
   for (kk=0; kk < N; kk = kk+B)
      for (i=0; i < N; i++)
         for (j=jj; j < min(jj+B,N); j++)
         {
          r = 0.0;
          for (k=kk; k < min(kk+B,N); k++)
             r = r + y[i][k]*z[k][j];
          x[i][j] = x[i][j]+r;
         }
```

#9: Compiler Optimizations: Blocking

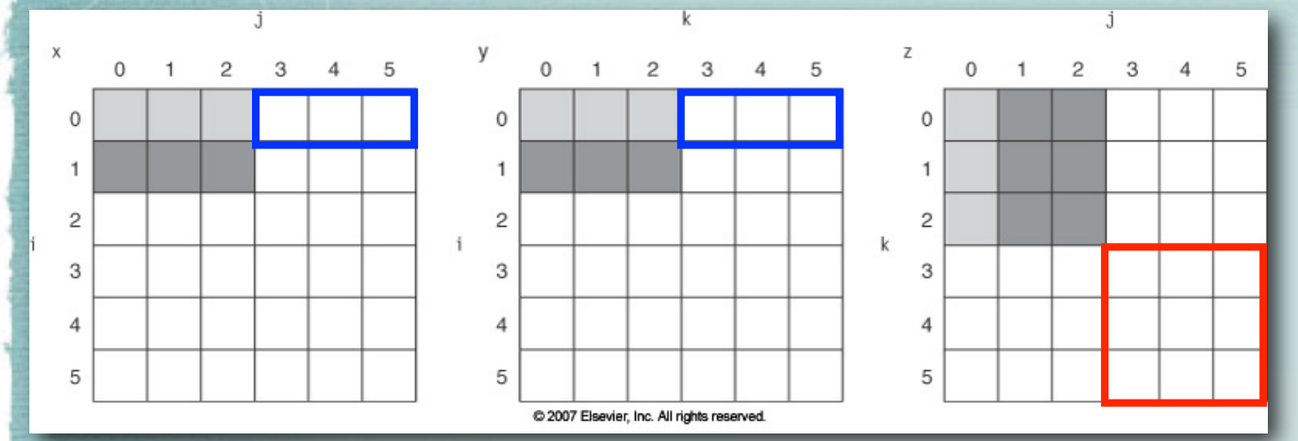
(To Reduce Miss Rate)




```
for (jj=0; jj < N; jj = jj+B)
   for (kk=0; kk < N; kk = kk+B)
      for (i=0; i < N; i++)
         for (j=jj; j < min(jj+B,N); j++)
         {
          r = 0.0;
          for (k=kk; k < min(kk+B,N); k++)
             r = r + y[i][k]*z[k][j];
          x[i][j] = x[i][j]+r;
         }
```

#9: Compiler Optimizations: Blocking

(To Reduce Miss Rate)



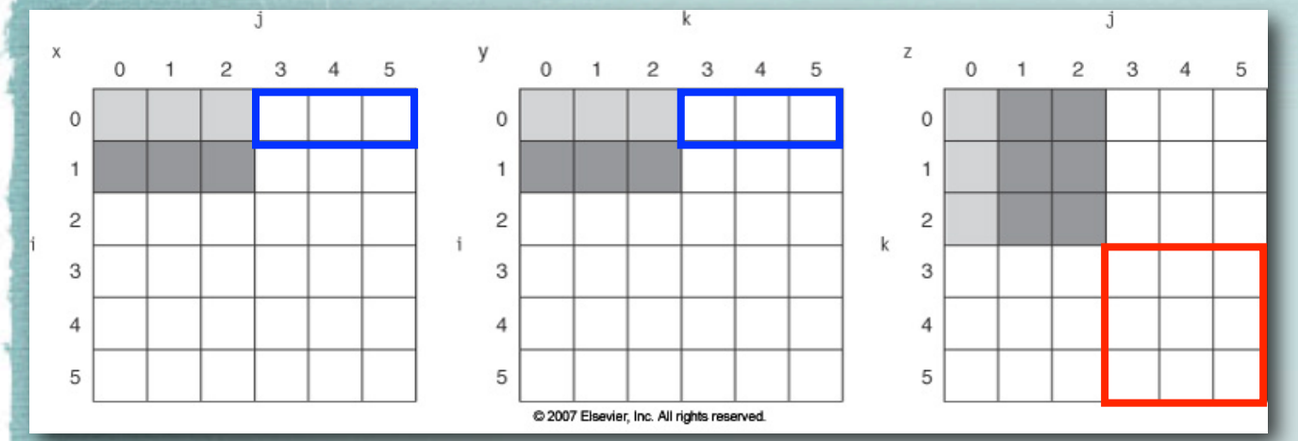

```
for (jj=0; jj < N; jj = jj+B)
   for (kk=0; kk < N; kk = kk+B)
      for (i=0; i < N; i++)
         for (j=jj; j < min(jj+B,N); j++)
         {
          r = 0.0;
          for (k=kk; k < min(kk+B,N); k++)
             r = r + y[i][k]*z[k][j];
          x[i][j] = x[i][j]+r;
         }
```

Accesses:

z : N^2

#9: Compiler Optimizations: Blocking

(To Reduce Miss Rate)




```
for (jj=0; jj < N; jj = jj+B)
   for (kk=0; kk < N; kk = kk+B)
      for (i=0; i < N; i++)
         for (j=jj; j < min(jj+B,N); j++)
         {
          r = 0.0;
          for (k=kk; k < min(kk+B,N); k++)
             r = r + y[i][k]*z[k][j];
          x[i][j] = x[i][j]+r;
         }
```

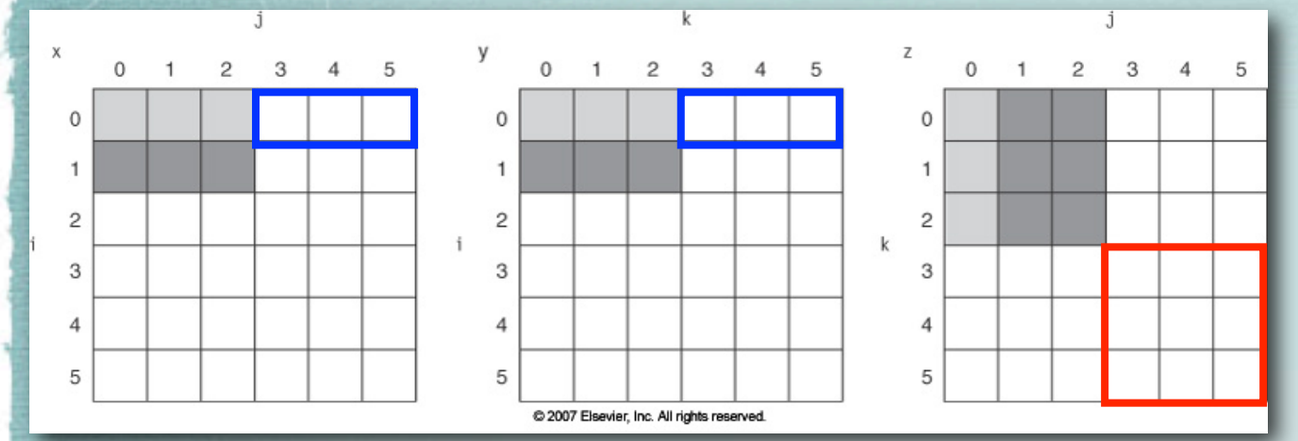
Accesses:

z : N^2

Compulsory misses

#9: Compiler Optimizations: Blocking

(To Reduce Miss Rate)




```
for (jj=0; jj < N; jj = jj+B)
   for (kk=0; kk < N; kk = kk+B)
      for (i=0; i < N; i++)
         for (j=jj; j < min(jj+B,N); j++)
         {
          r = 0.0;
          for (k=kk; k < min(kk+B,N); k++)
             r = r + y[i][k]*z[k][j];
          x[i][j] = x[i][j]+r;
         }
```

Accesses:

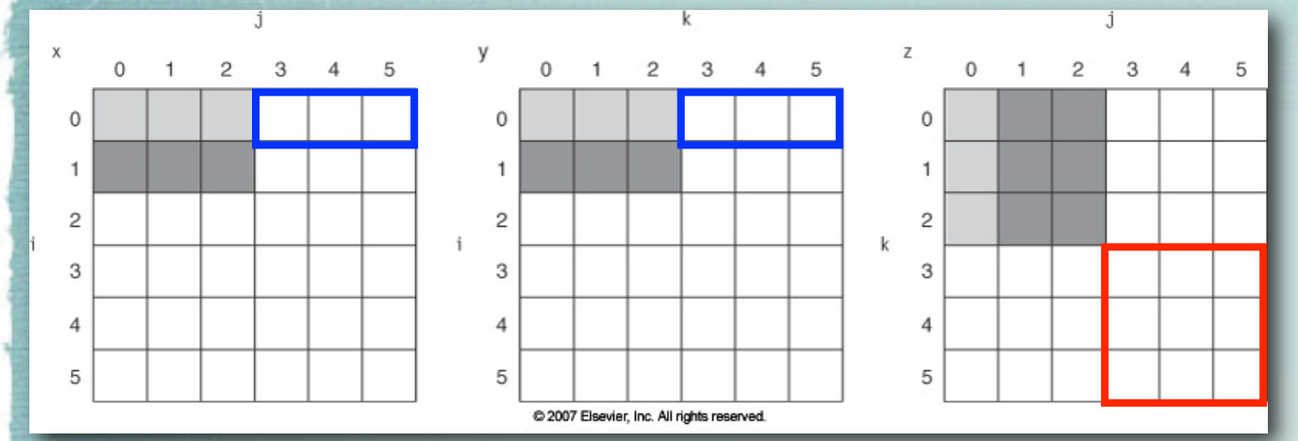
Compulsory misses

z : N^2

x : $B^2 \times N/B \times \#blocks = N \times B \times (N/B)^2 = N^3/B$

#9: Compiler Optimizations: Blocking

(To Reduce Miss Rate)




```
for (jj=0; jj < N; jj = jj+B)
   for (kk=0; kk < N; kk = kk+B)
      for (i=0; i < N; i++)
         for (j=jj; j < min(jj+B,N); j++)
         {
          r = 0.0;
          for (k=kk; k < min(kk+B,N); k++)
             r = r + y[i][k]*z[k][j];
          x[i][j] = x[i][j]+r;
         }
```

Elements in a block

Accesses:

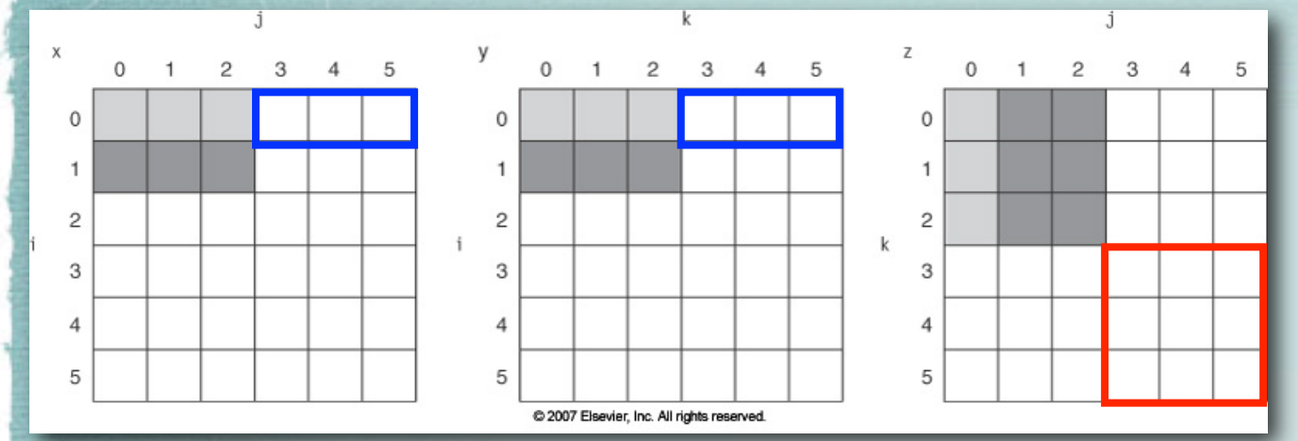
Compulsory misses

z : N^2

x : $B^2 \times N/B \times \#blocks = N \times B \times (N/B)^2 = N^3/B$

#9: Compiler Optimizations: Blocking

(To Reduce Miss Rate)




```
for (jj=0; jj < N; jj = jj+B)
   for (kk=0; kk < N; kk = kk+B)
      for (i=0; i < N; i++)
         for (j=jj; j < min(jj+B,N); j++)
         {
          r = 0.0;
          for (k=kk; k < min(kk+B,N); k++)
             r = r + y[i][k]*z[k][j];
          x[i][j] = x[i][j]+r;
         }
```

Accesses:

z : N^2 — Compulsory misses

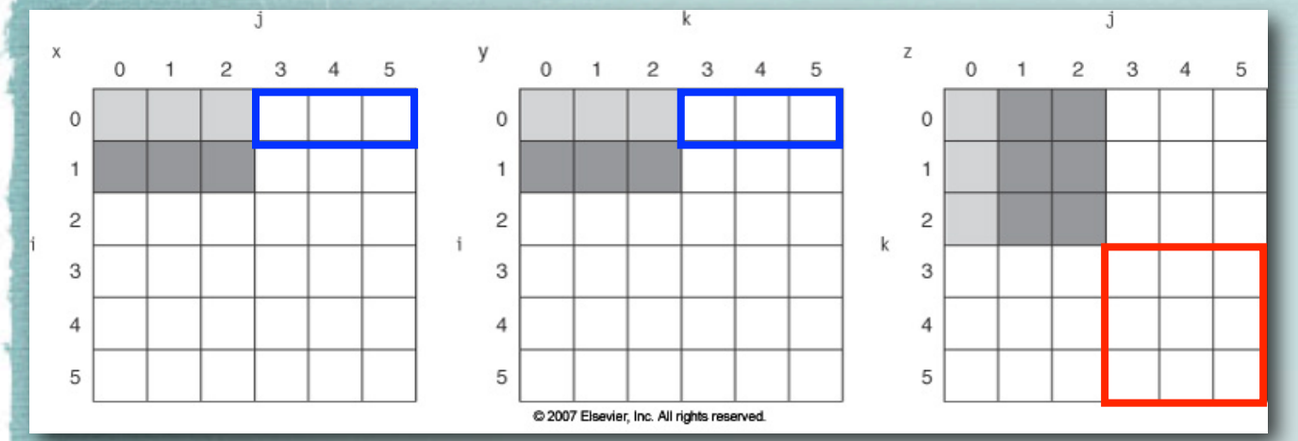
x : $B^2 \times N/B \times \#blocks = N \times B \times (N/B)^2 = N^3/B$

Elements in a block

Number of sweeps

#9: Compiler Optimizations: Blocking

(To Reduce Miss Rate)




```
for (jj=0; jj < N; jj = jj+B)
   for (kk=0; kk < N; kk = kk+B)
      for (i=0; i < N; i++)
         for (j=jj; j < min(jj+B,N); j++)
         {
          r = 0.0;
          for (k=kk; k < min(kk+B,N); k++)
             r = r + y[i][k]*z[k][j];
          x[i][j] = x[i][j]+r;
         }
```

Elements in a block

Compulsory misses

Number of sweeps

Accesses:

z : N^2

x : $B^2 \times N/B \times \#blocks = N \times B \times (N/B)^2 = N^3/B$

y : N^3/B

Total: $2N^3/B+N^2$ (against $2N^3+N^2$)

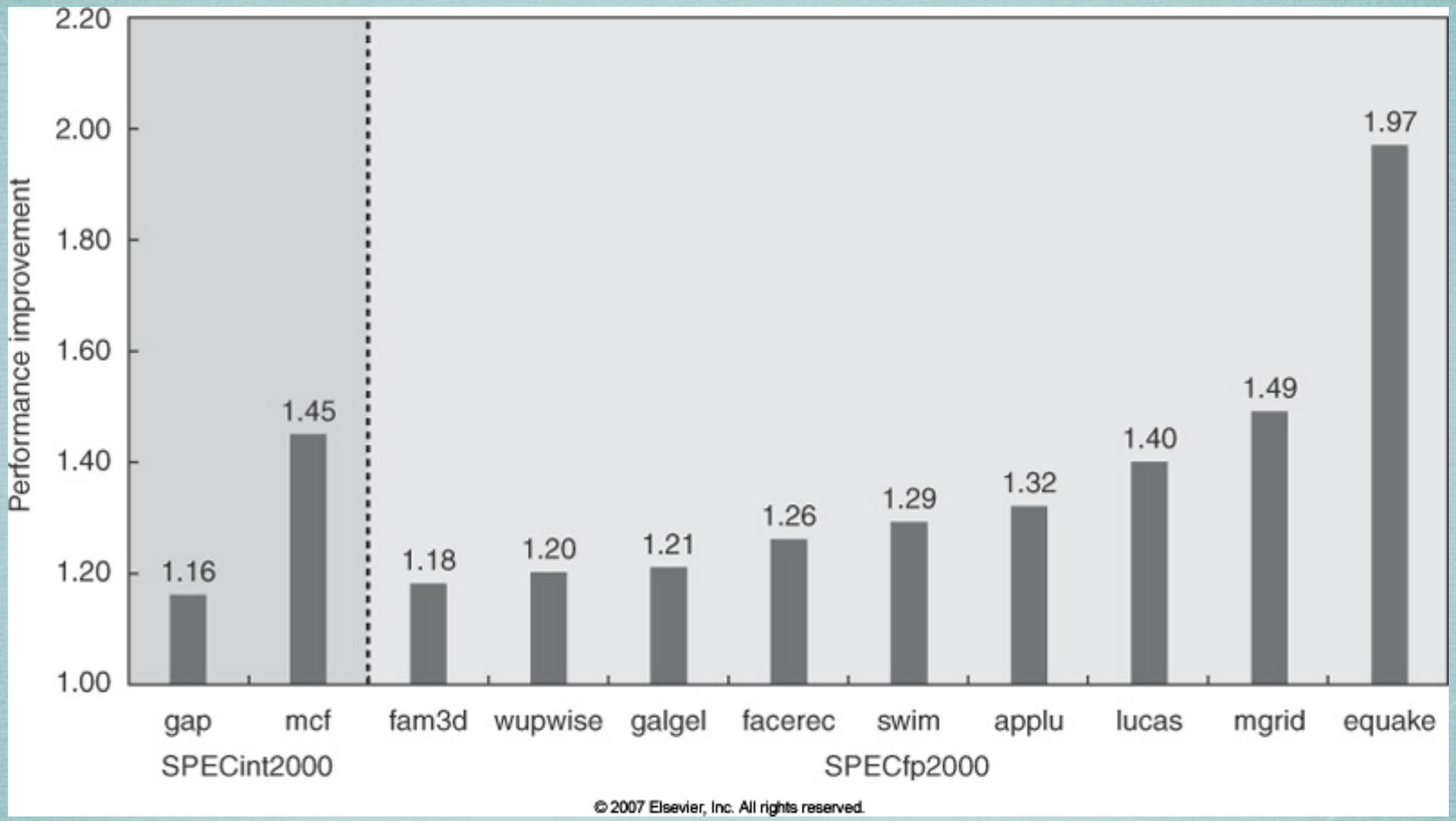
#10: Hardware Prefetching

(To Reduce Miss Penalty or Miss Rate)

- Instruction prefetch
  - ⋆ prefetch two blocks, instead of one, on miss
- Data prefetch
  - ⋆ extend the same idea to data
  - ⋆ an older study found 50% to 70% misses could be captured with 8 stream buffers (one for instruction, 7 for data)
  - ⋆ Pentium 4 can prefetch into L2 cache from up to 8 streams
    - ✳ invokes prefetch upon two successive misses to a page
    - ✳ won't prefetch across 4KB page boundary

#10: Hardware Prefetching

(To Reduce Miss Penalty or Miss Rate)



Speedup due to hardware prefetching on Pentium 4

#11: Compiler-Controlled Prefetching

(To Reduce Miss Penalty or Miss Rate)

- Register prefetch
  - ⋆ preload register
- Cache prefetch
  - ⋆ load into the cache, but not register
- Either could be *faulting* or *non-faulting*
  - ⋆ normal load is *faulting register prefetch*
  - ⋆ non-faulting prefetches turn into no-ops
- Usually need non-blocking caches to be effective

Cache Optimization Summary

Technique	Hit time	Band-width	Miss pen-alty	Miss rate	Hardware cost/ complexity	Comment
Small and simple caches	+			–	0	Trivial; widely used
Way-predicting caches	+				1	Used in Pentium 4
Trace caches	+				3	Used in Pentium 4
Pipelined cache access	–	+			1	Widely used
Nonblocking caches		+	+		3	Widely used
Banked caches		+			1	Used in L2 of Opteron and Niagara
Critical word first and early restart			+		2	Widely used
Merging write buffer			+		1	Widely used with write through
Compiler techniques to reduce cache misses				+	0	Software is a challenge; some computers have compiler option
Hardware prefetching of instructions and data			+	+	2 instr., 3 data	Many prefetch instructions; Opteron and Pentium 4 prefetch data
Compiler-controlled prefetching			+	+	3	Needs nonblocking cache; possible instruction overhead; in many CPUs

VIRTUAL MACHINES

Virtual Machines: An Old Idea

A virtual machine is taken to be an efficient, isolated duplicate *of the real machine. We explain these notions through the idea of a virtual machine monitor (VMM) ... a VMM has three essential characteristics. First, the VMM provides an environment for programs which is essentially identical with the original machine; second, programs run in this environment show at worst only minor decreases in speed; and last, the VMM is in complete control of the system resources.*

Gerlad Popek and Robert Goldberg
"Formal requirements for virtualizable third generation architectures,"
Communications of the ACM (July 1974)

Protection via Virtual Machines

- (Operating) System Virtual Machines
  - ⋆ does not include JVM or Microsoft CLR
  - ⋆ VMware ESX, Xen (*hypervisors* or *virtual machine monitors*, can run on bare machines)
  - ⋆ Parallels, VMware Fusion (run on a host OS)
- Regained popularity
  - ⋆ increased importance of isolation and security
  - ⋆ failures in security and reliability of standard OSes
  - ⋆ sharing of computers among unrelated users
  - ⋆ increased hardware speeds, making VM overheads acceptable

Popularity of Virtual Machines: II

- Protection
  - ⋆ see previous slide
- Software management
  - ⋆ could run legacy operating systems
- Hardware management
  - ⋆ let separate software stacks share hardware
    - ✳ also useful at the end-user level
  - ⋆ some VMMs support migration of a running VM to a different computer, for load-balancing and fault tolerance

Hardware that allows VM to execute directly on hardware is called *virtualizable*

Complications

- "Difficult" instructions
  - ⋆ *paravirtualization*: make some minimal changes to the guest OS to avoid difficult instructions
- Virtual memory
  - ⋆ separate *virtual*, *physical*, and *machine* memory
  - ⋆ maintain shadow page table to avoid double translation; alternatively, need hardware support for multiple indirections
- TLB virtualization
  - ⋆ VMM maintains per-OS TBB copies
  - ⋆ TLBs with process-ID tag can avoid TLB flush on VM context switch through the use of virtual PIDs
- I/O sharing

Xen vs Native Linux