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

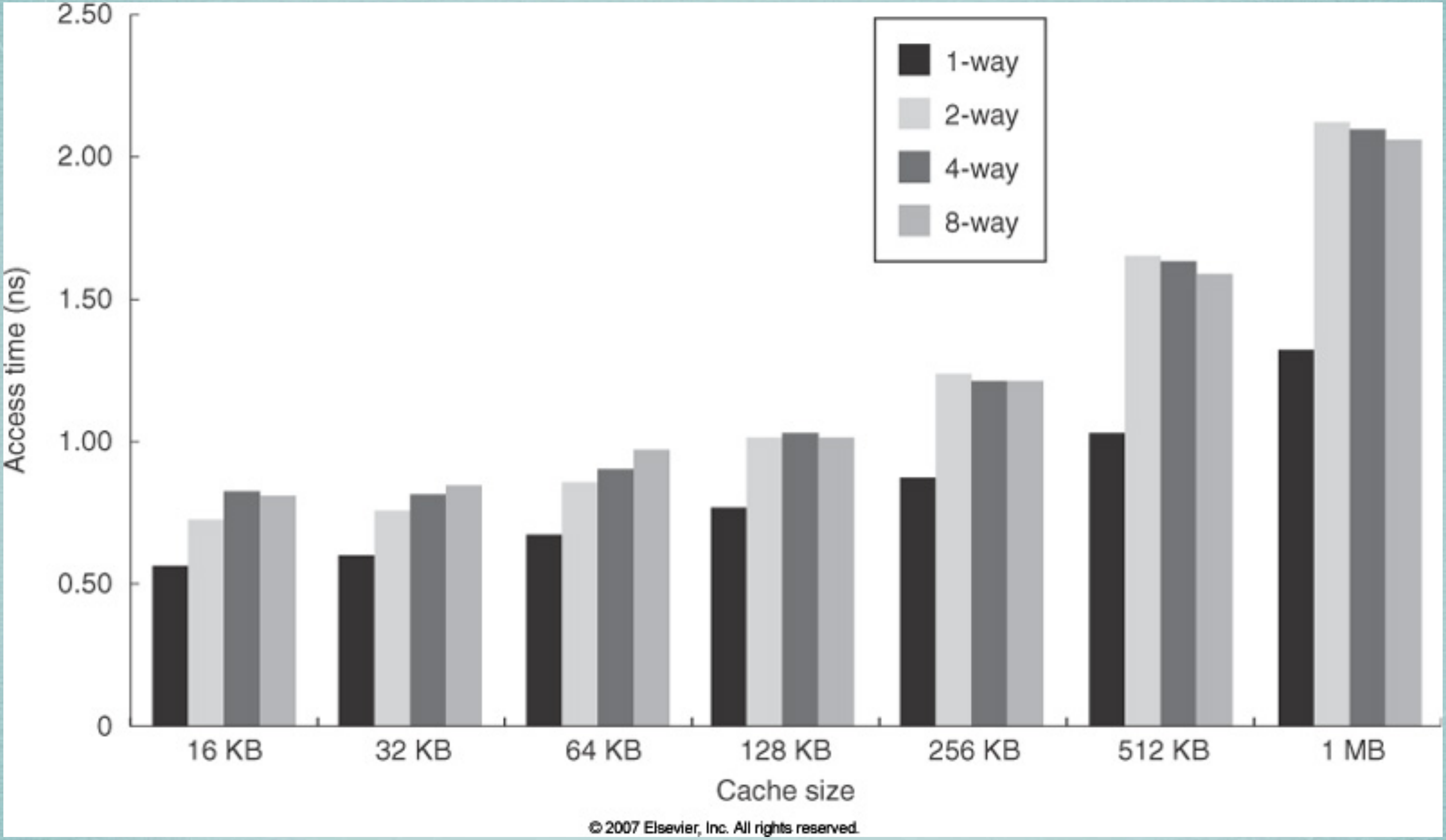
$$\boxed{\text{Average memory access time} = \text{Hit time} + \text{Miss rate} \times \text{Miss penalty}}$$

- 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)

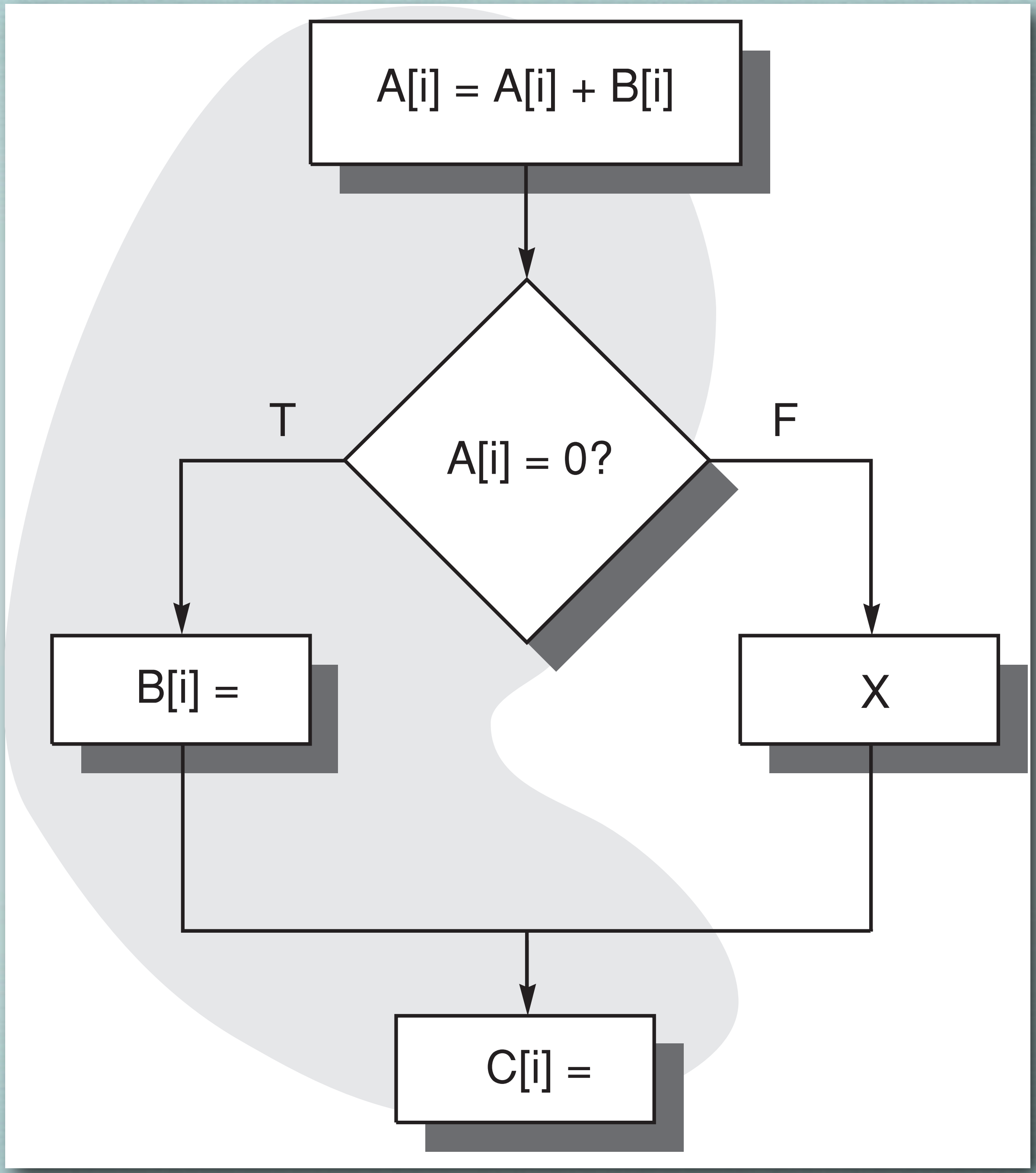


#2: Way Prediction
(To Reduce Hit Time)

- 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)

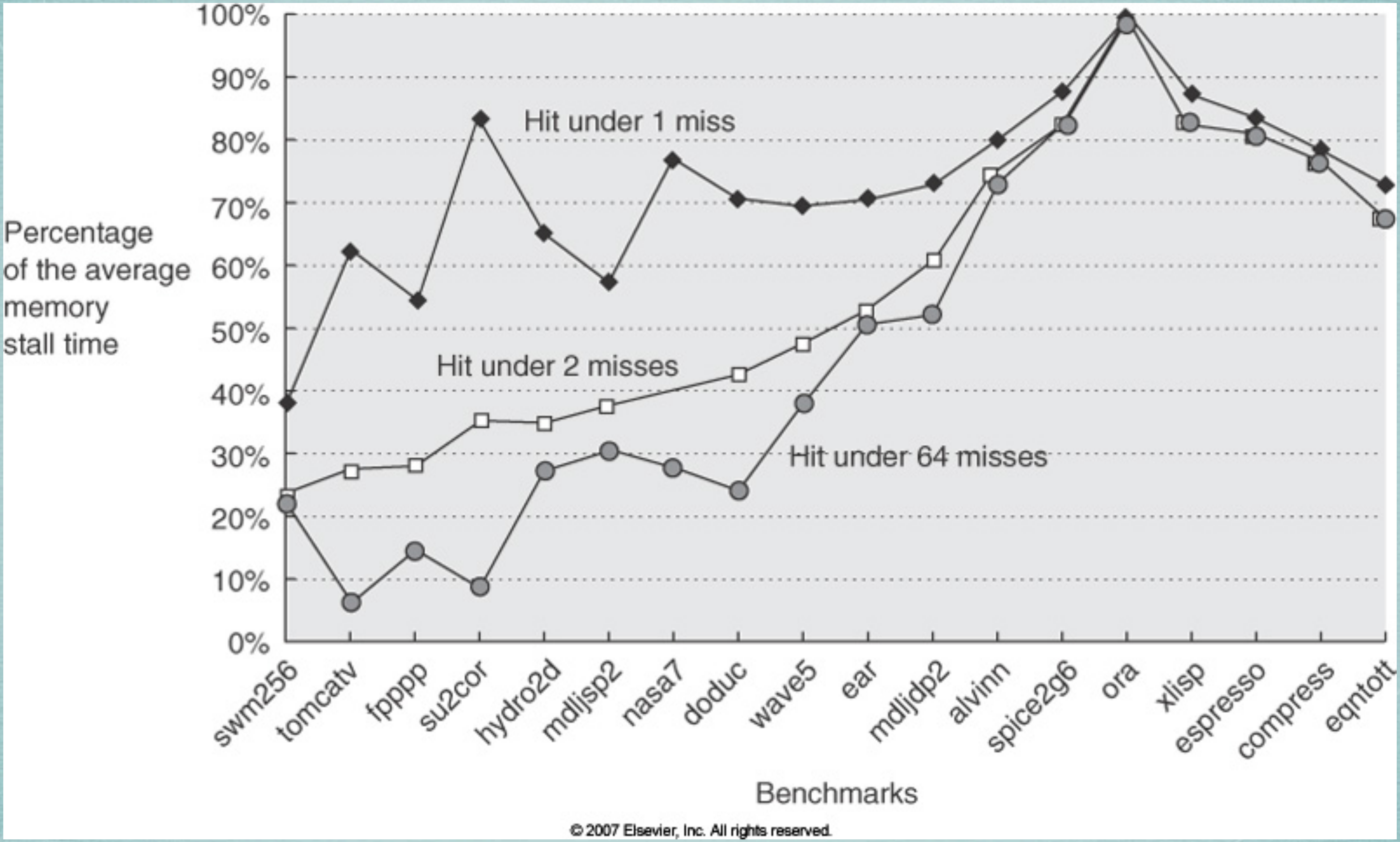
- Pipeline results in fast clock cycle time and high bandwidth, but slow hits
  - ★ Pentium 1: 1 clock cycle for instruction cache
  - ★ Pentium Pro / III: 2 clock cycles
  - ★ Pentium 4: 4 clock cycles

#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

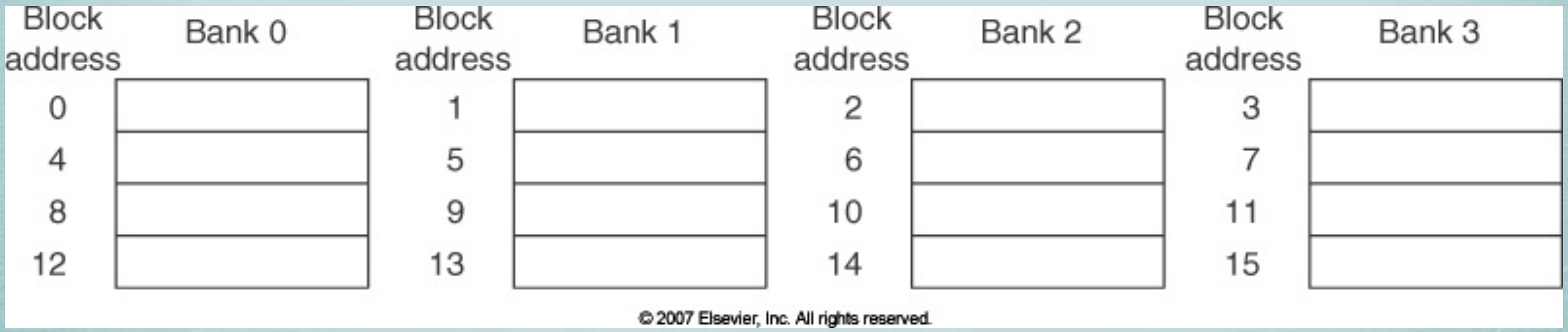
(To Increase Cache Bandwidth)



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



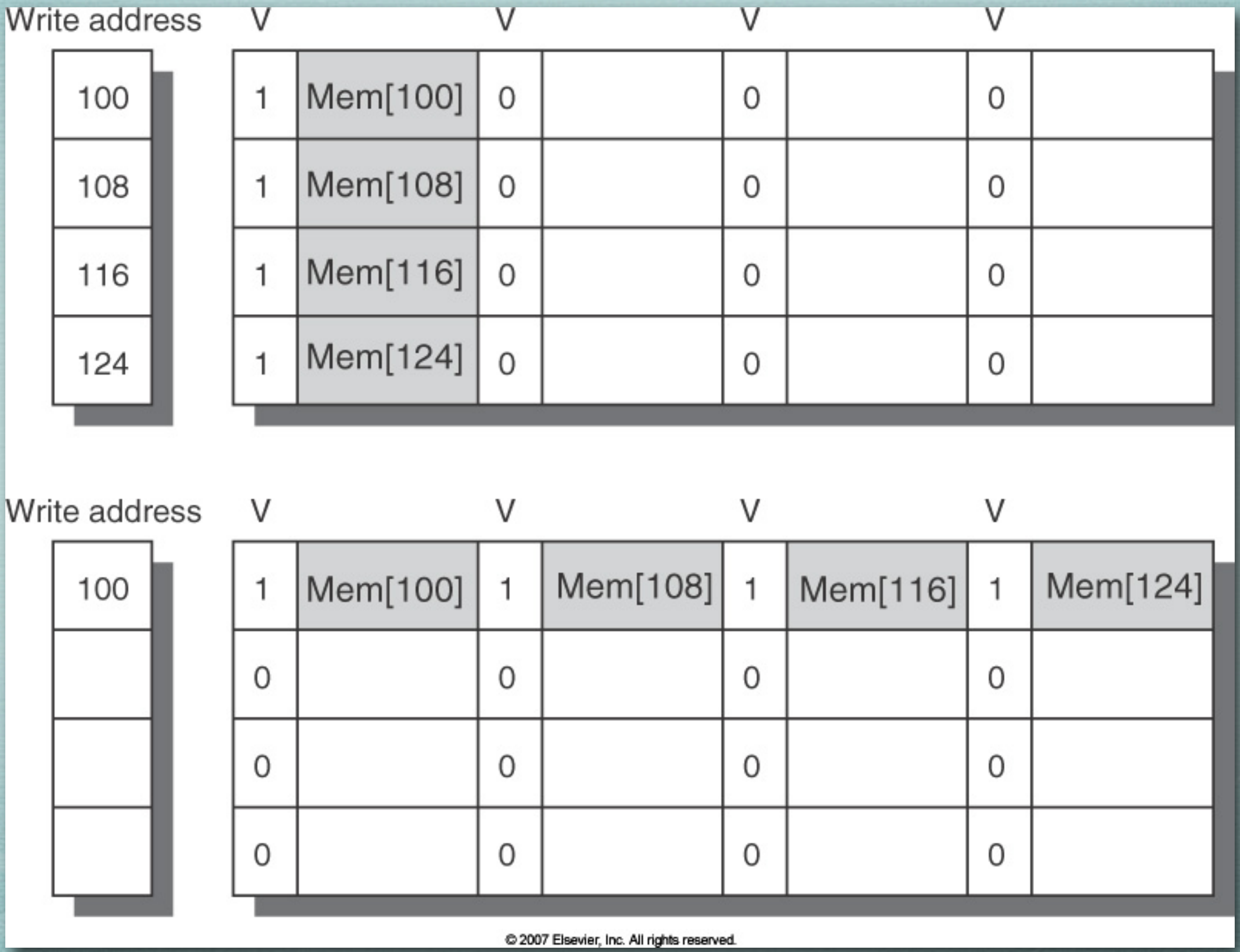
#7: Critical Word First and Early Restart

(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

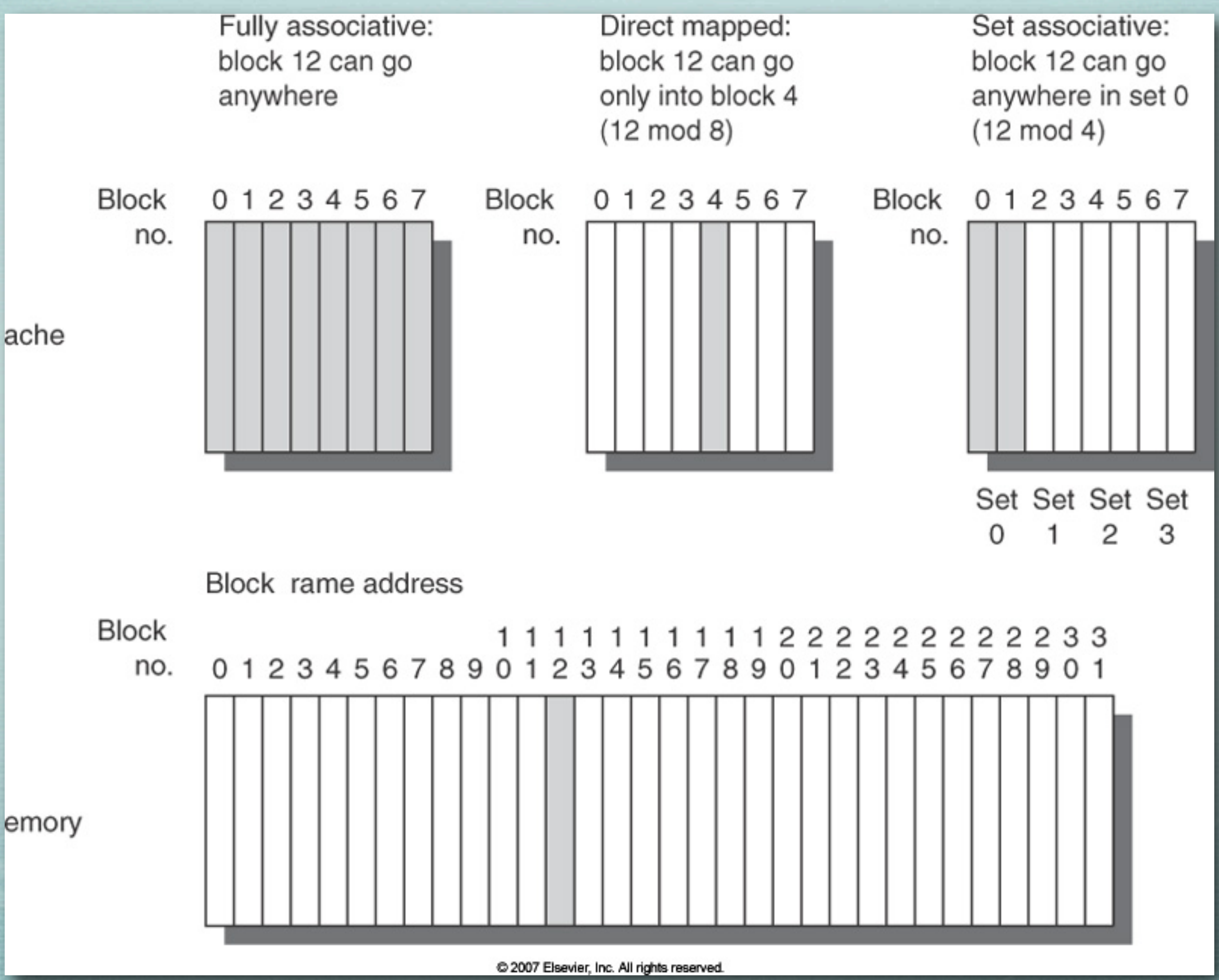
(To Reduce Miss Rate)

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

#9: Compiler Optimizations: Code

(To Reduce Miss Rate)

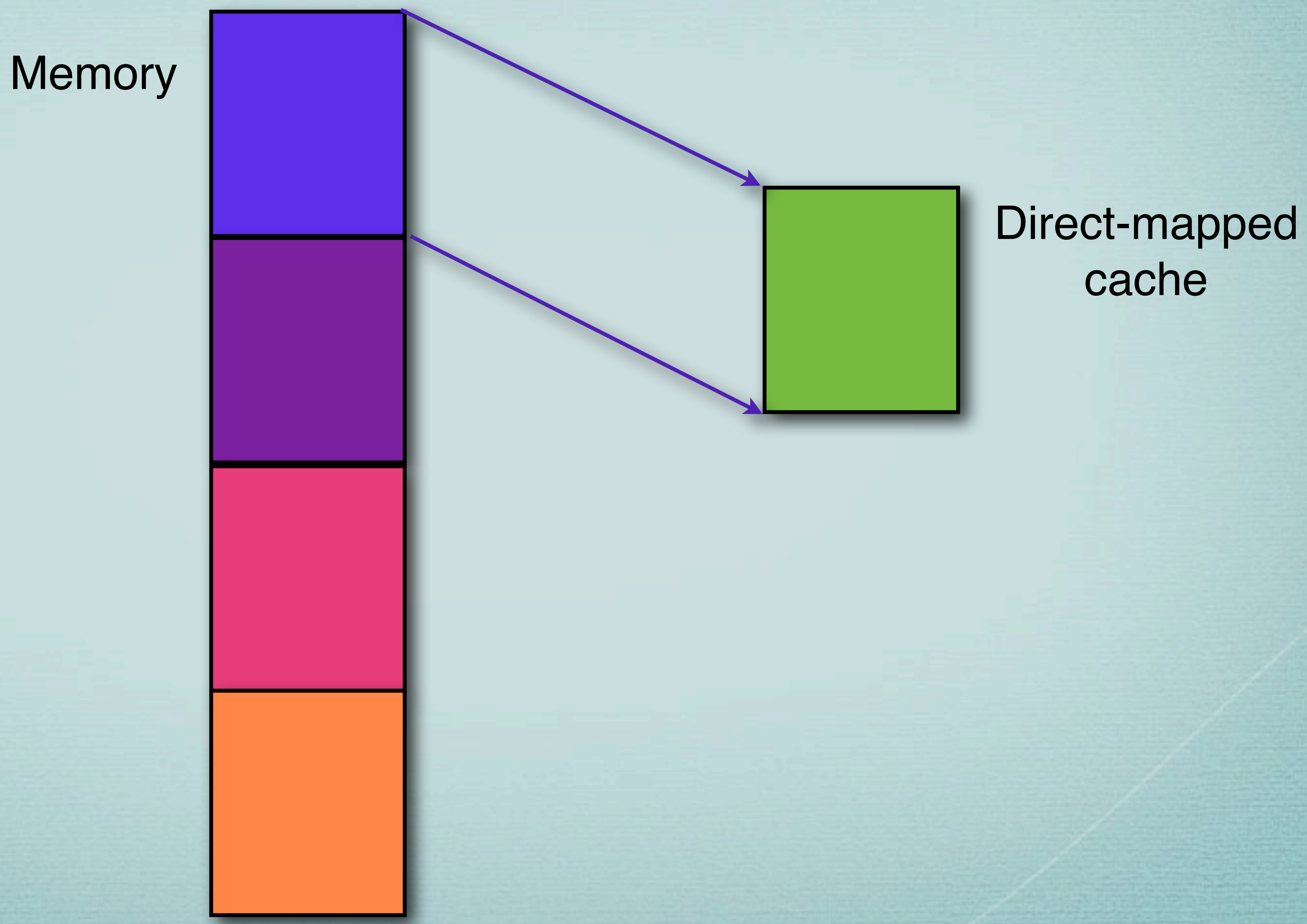
- Reordering procedures to reduce conflict misses



Fully associative: block 12 can go anywhere
Direct mapped: block 12 can go only into block 4 (12 mod 8)
Set associative: block 12 can go anywhere in set 0 (12 mod 4)
Block no.
0 1 2 3 4 5 6 7
ache
Set 0
Set 1
Set 2
Set 3
Block rame address
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
emory
© 2007 Elsevier, Inc. All rights reserved.

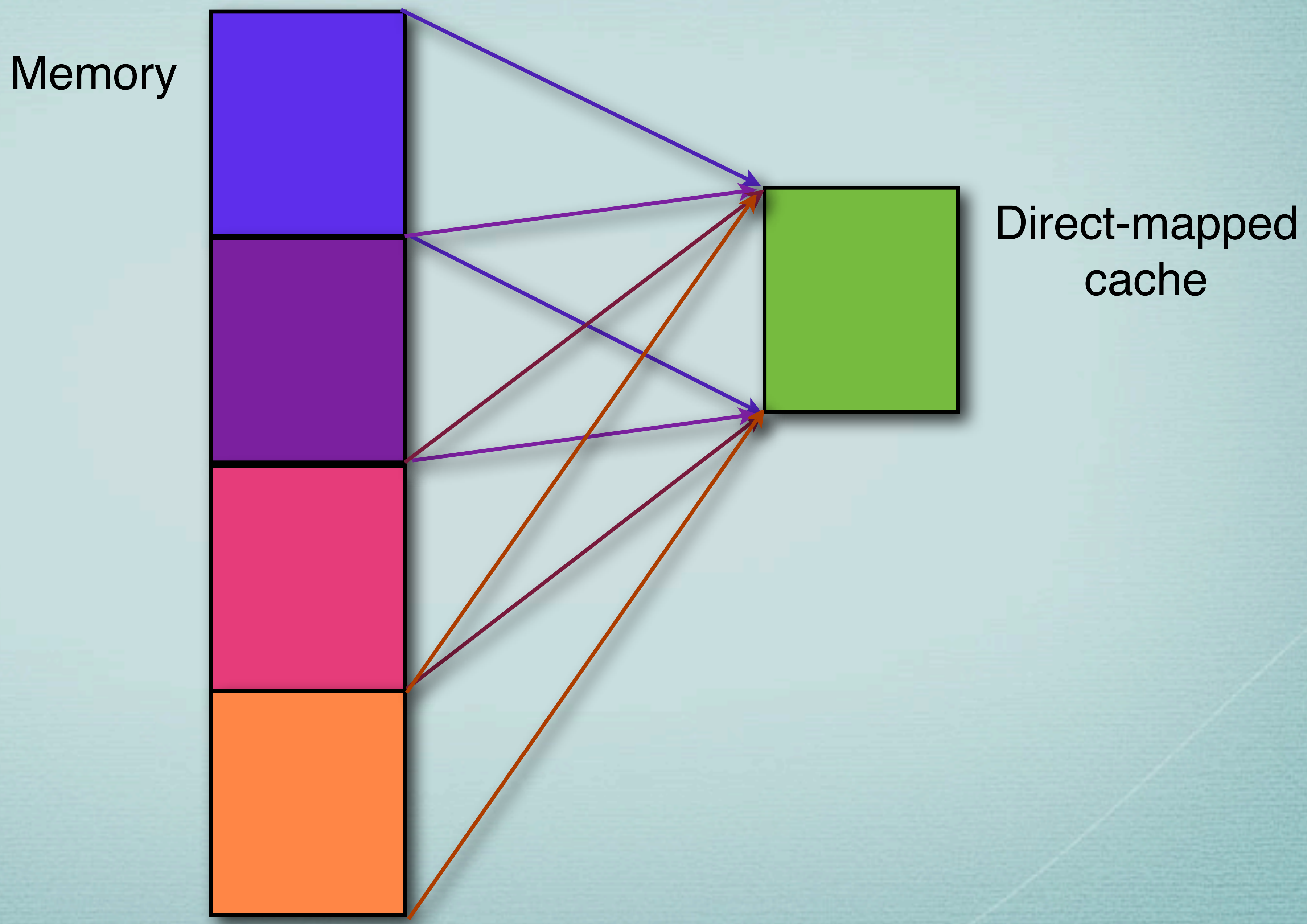
#9: Compiler Optimizations: Code
(To Reduce Miss Rate)

- Reordering procedures to reduce conflict misses



#9: Compiler Optimizations: Code
(To Reduce Miss Rate)

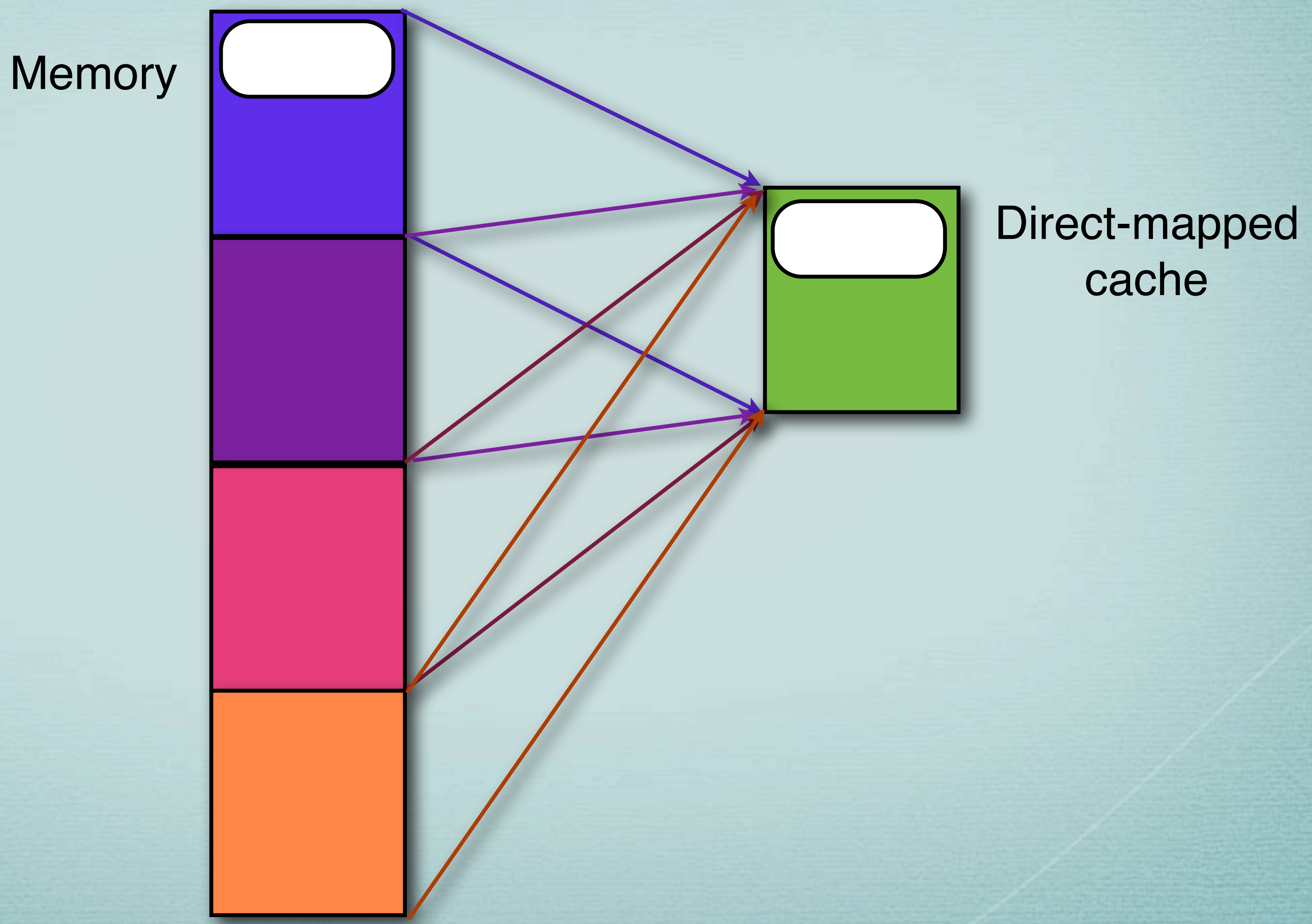
- Reordering procedures to reduce conflict misses



#9: Compiler Optimizations: Code

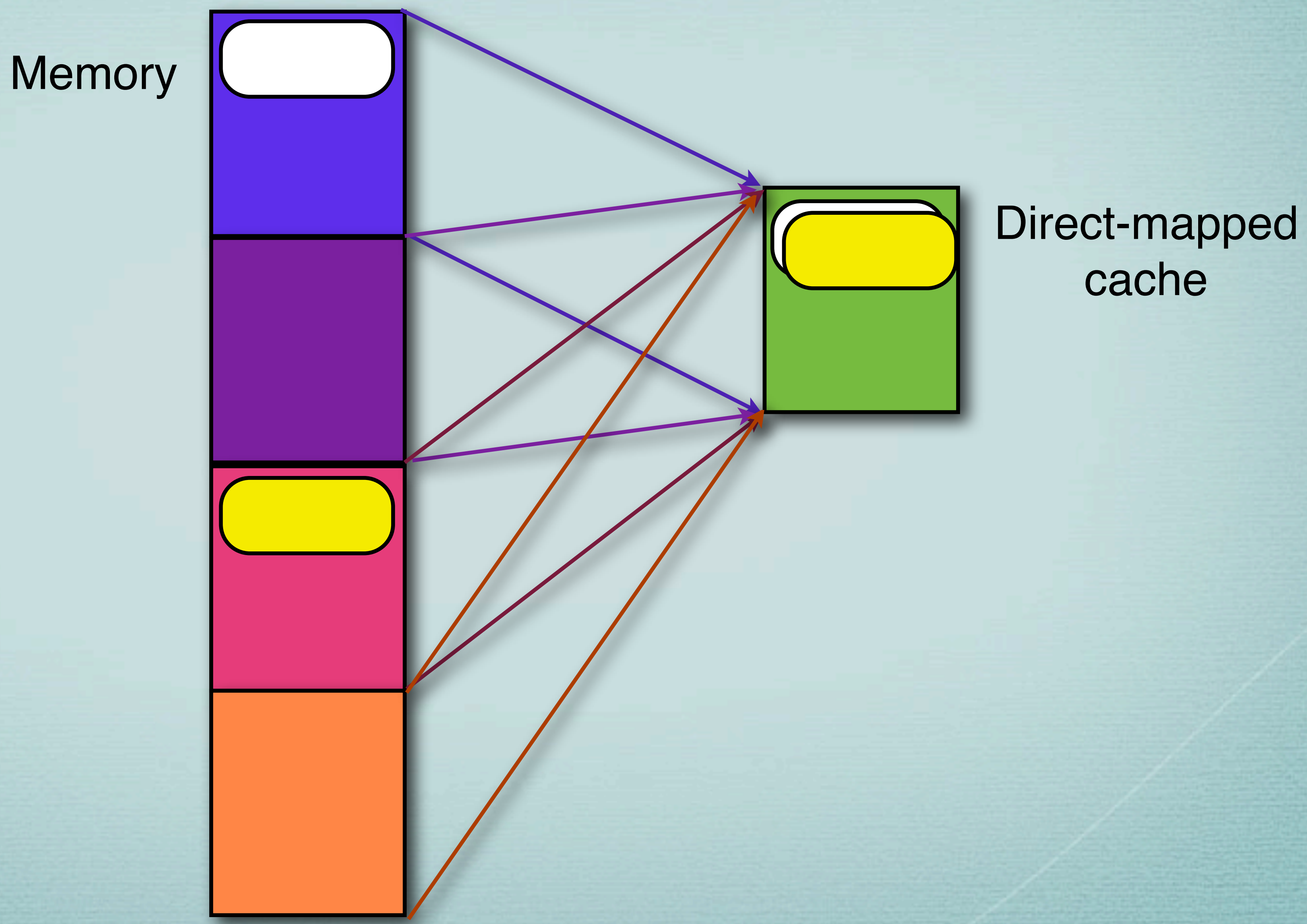
(To Reduce Miss Rate)

- Reordering procedures to reduce conflict misses



#9: Compiler Optimizations: Code
(To Reduce Miss Rate)

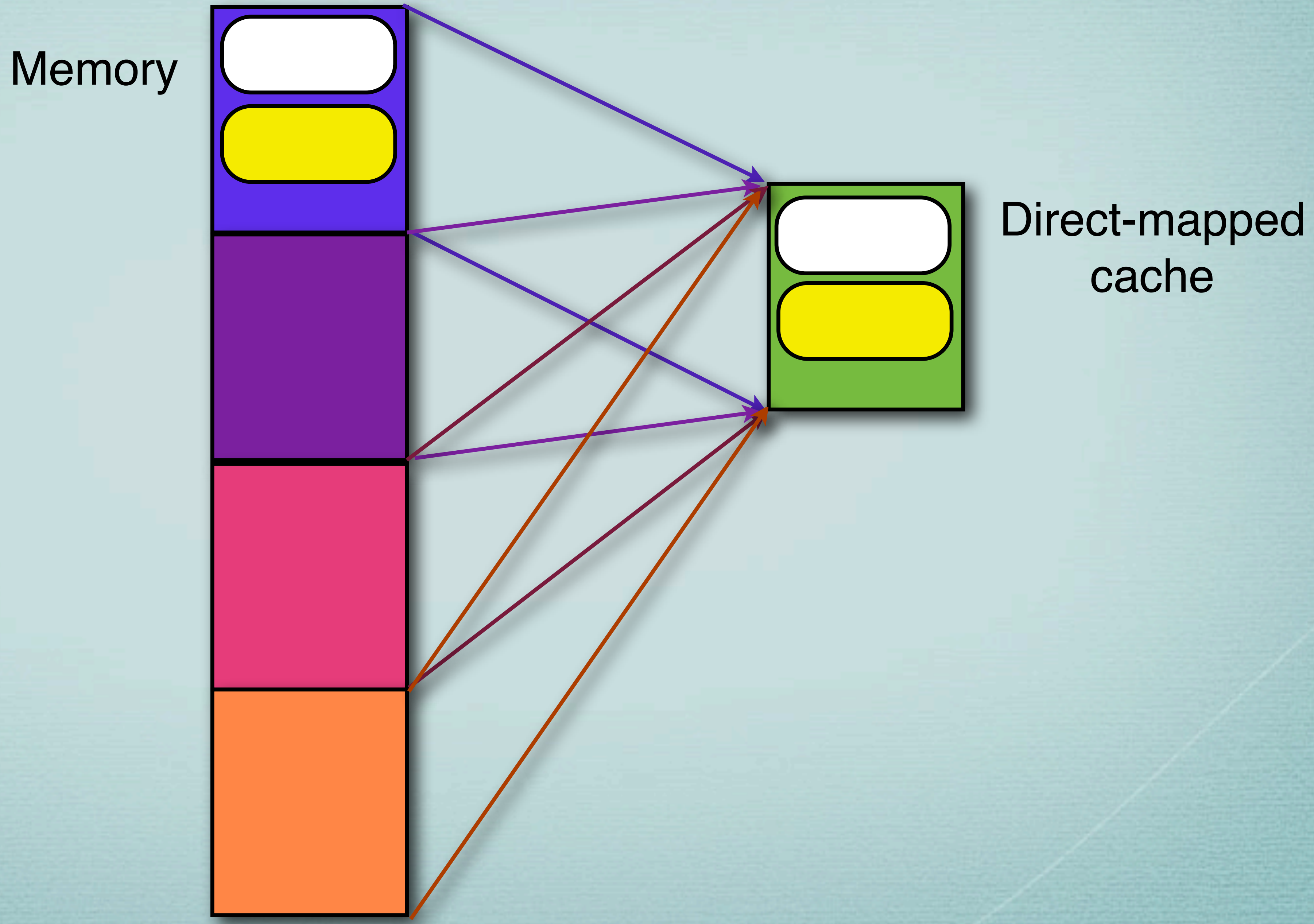
- Reordering procedures to reduce conflict misses



#9: Compiler Optimizations: Code

(To Reduce Miss Rate)

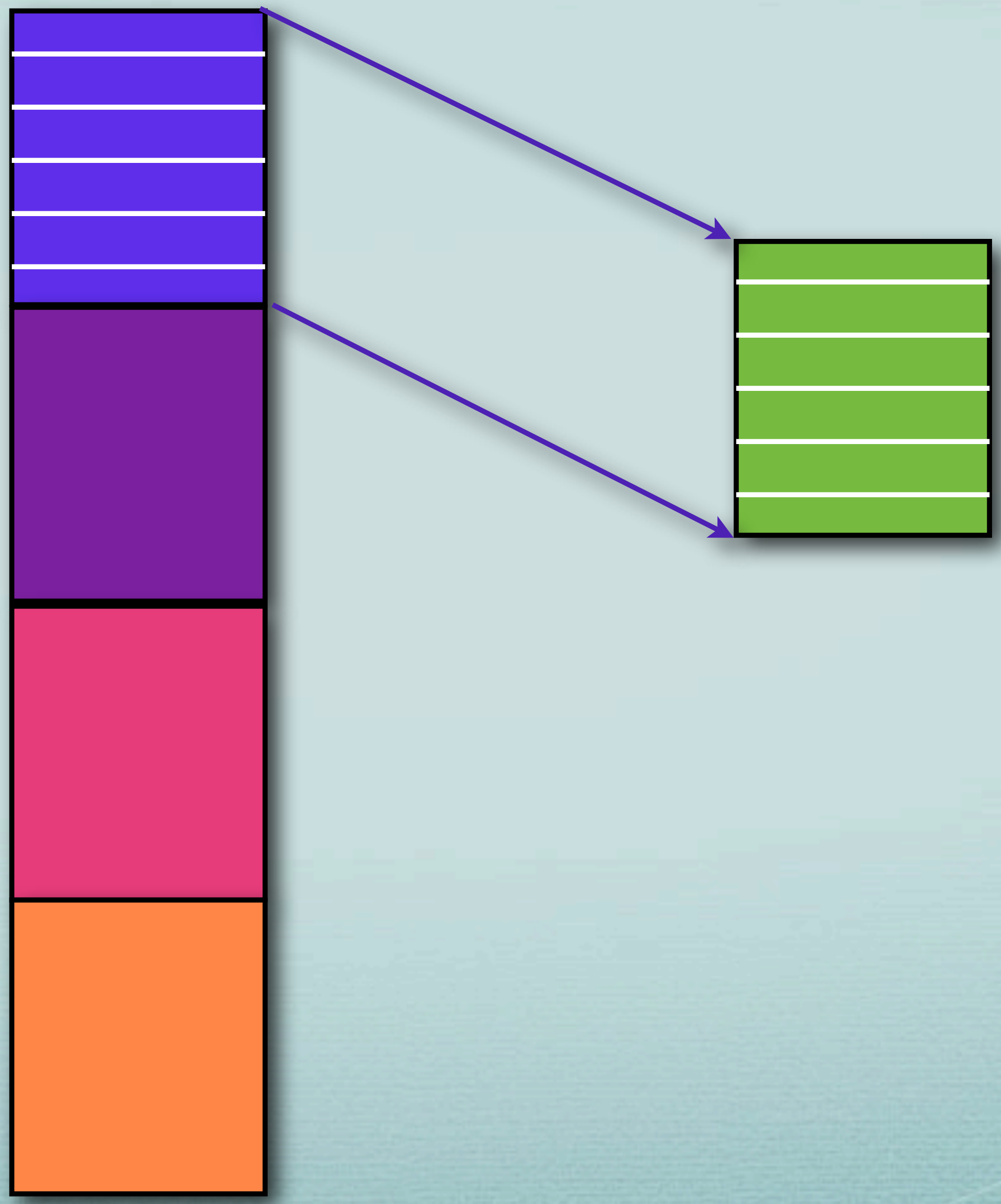
- Reordering procedures to reduce conflict misses



#9: Compiler Optimizations: Code

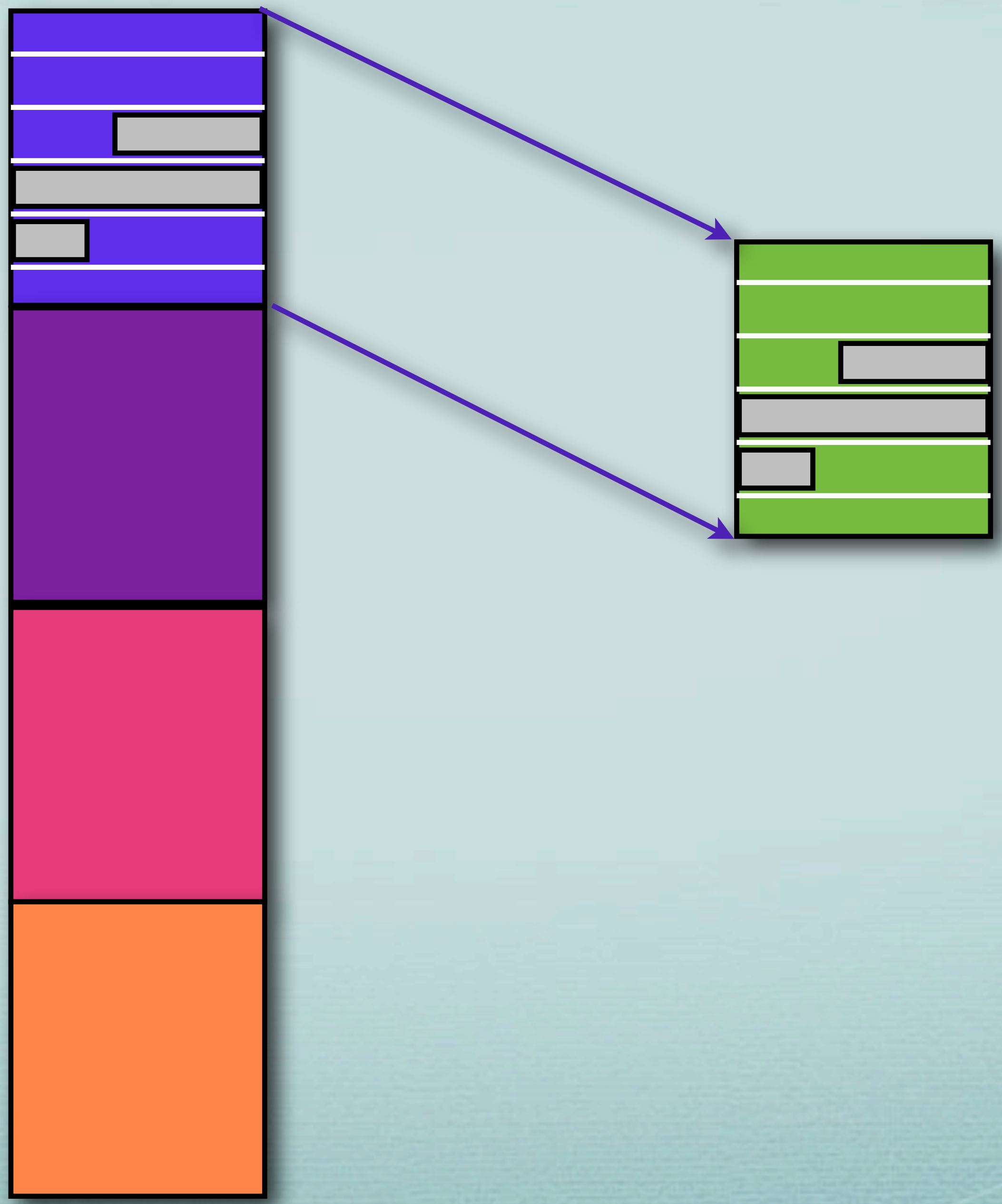
(To Reduce Miss Rate)

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



#9: Compiler Optimizations: Code
(To Reduce Miss Rate)

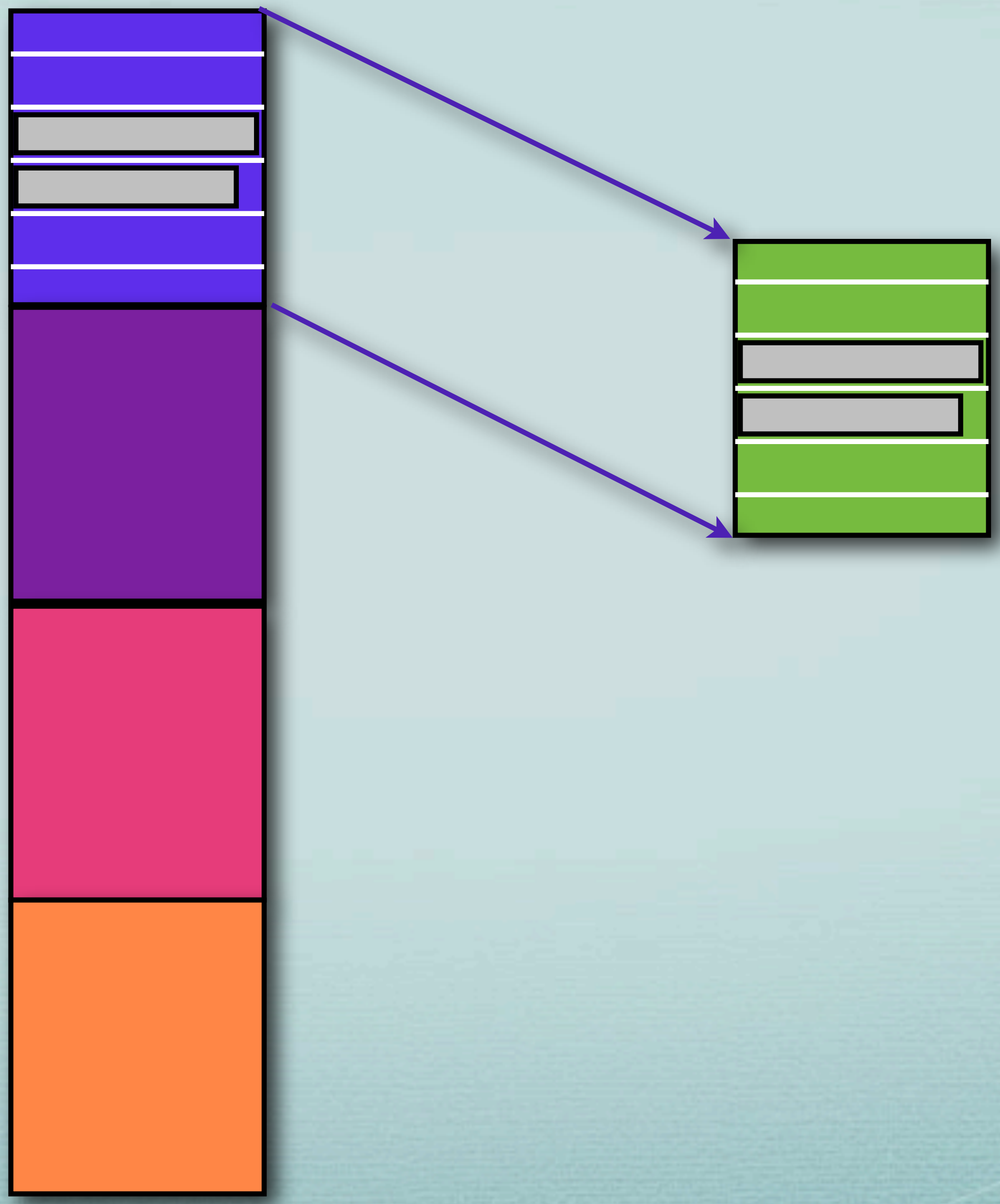
- Reordering procedures to reduce conflict misses
- Aligning basic blocks at cache block boundaries



#9: Compiler Optimizations: Code

(To Reduce Miss Rate)

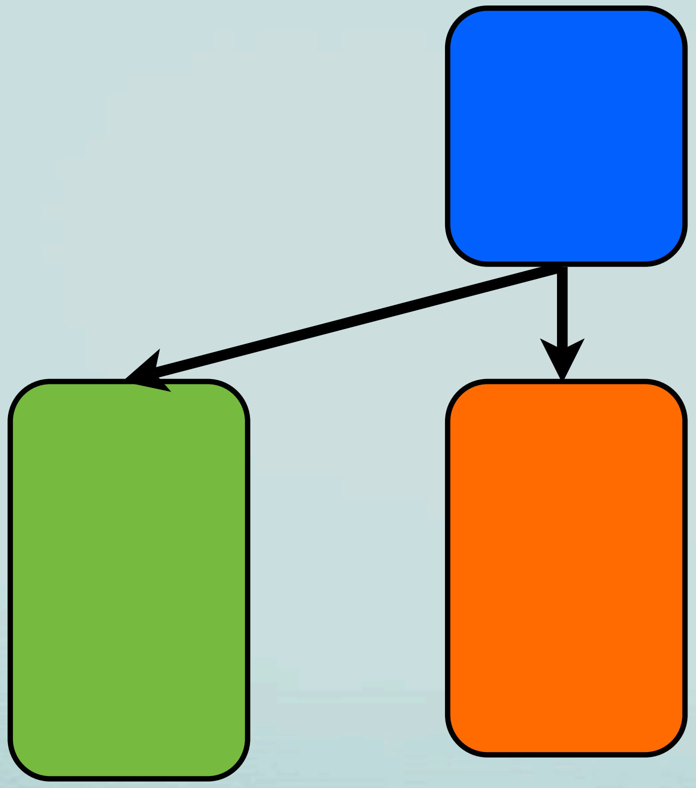
- Reordering procedures to reduce conflict misses
- Aligning basic blocks at cache block boundaries



#9: Compiler Optimizations: Code

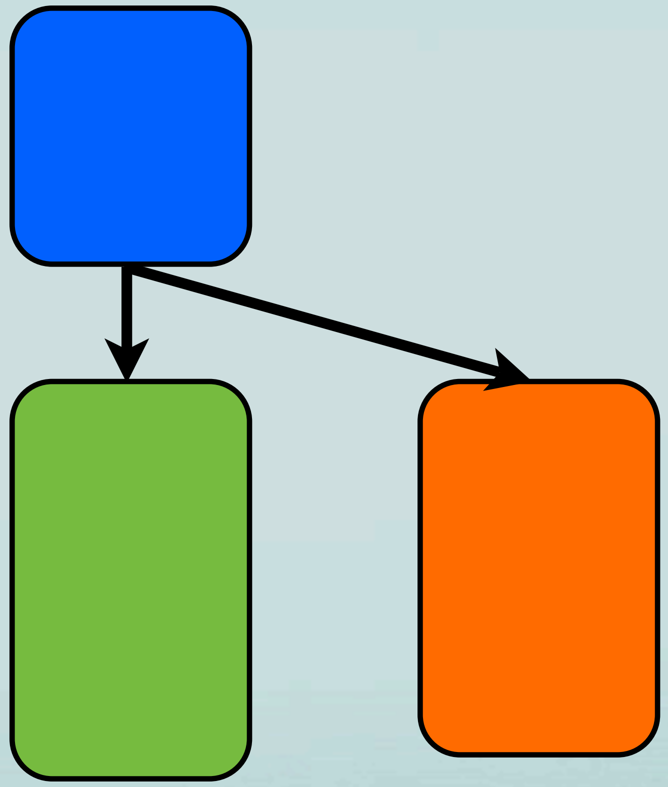
(To Reduce Miss Rate)

- 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

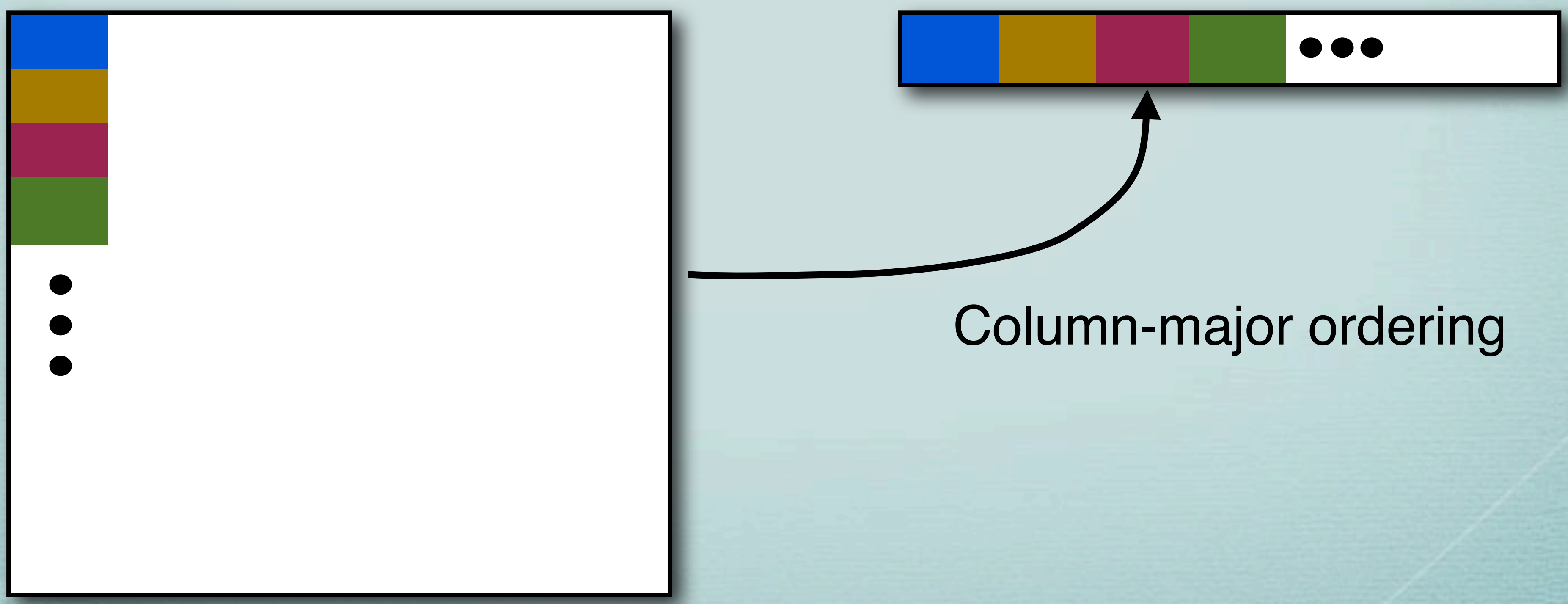
(To Reduce Miss Rate)

```
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)

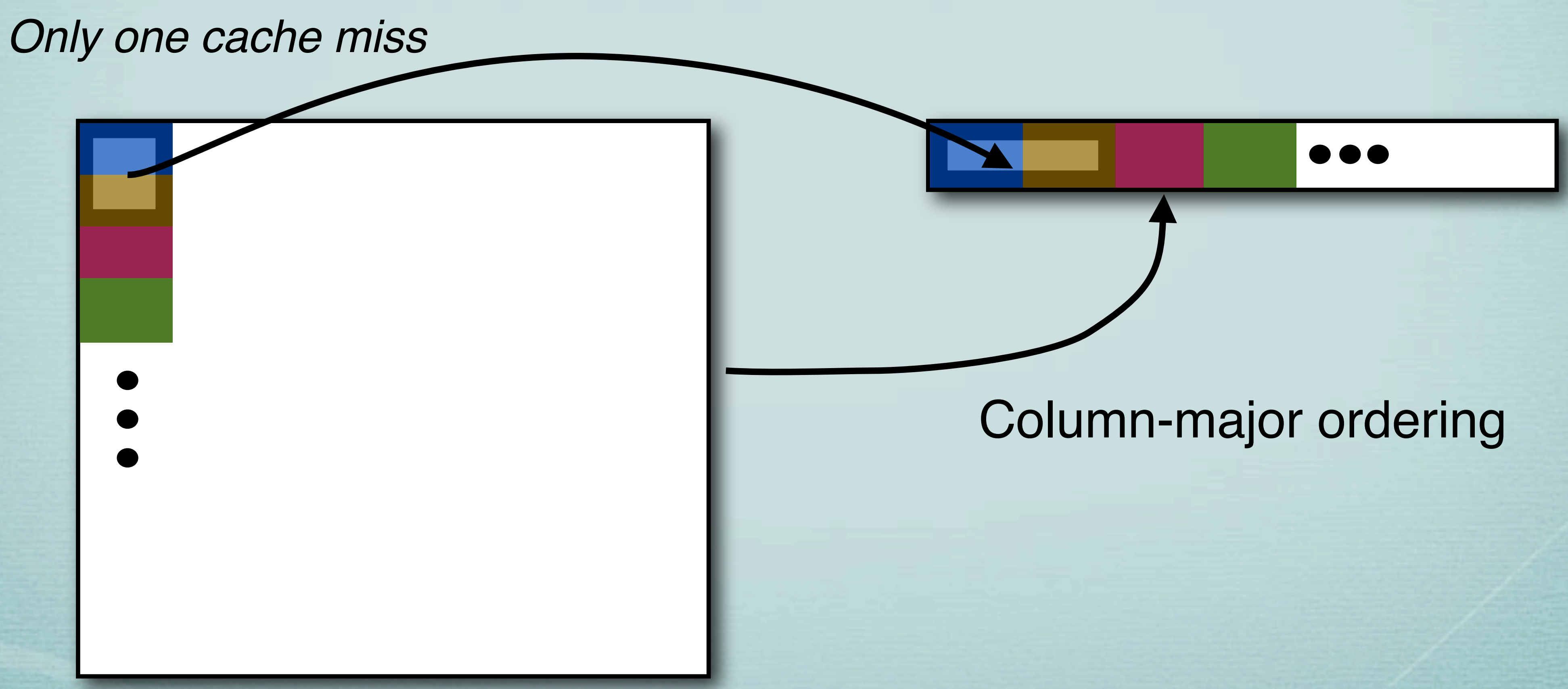
```
for (j=0; j < 100; j++)
    for (i=0; i < 5000; i++)
        x[i][j] = 2*x[i][j];
```



#9: Compiler Optimizations: Loop Interchange

(To Reduce Miss Rate)

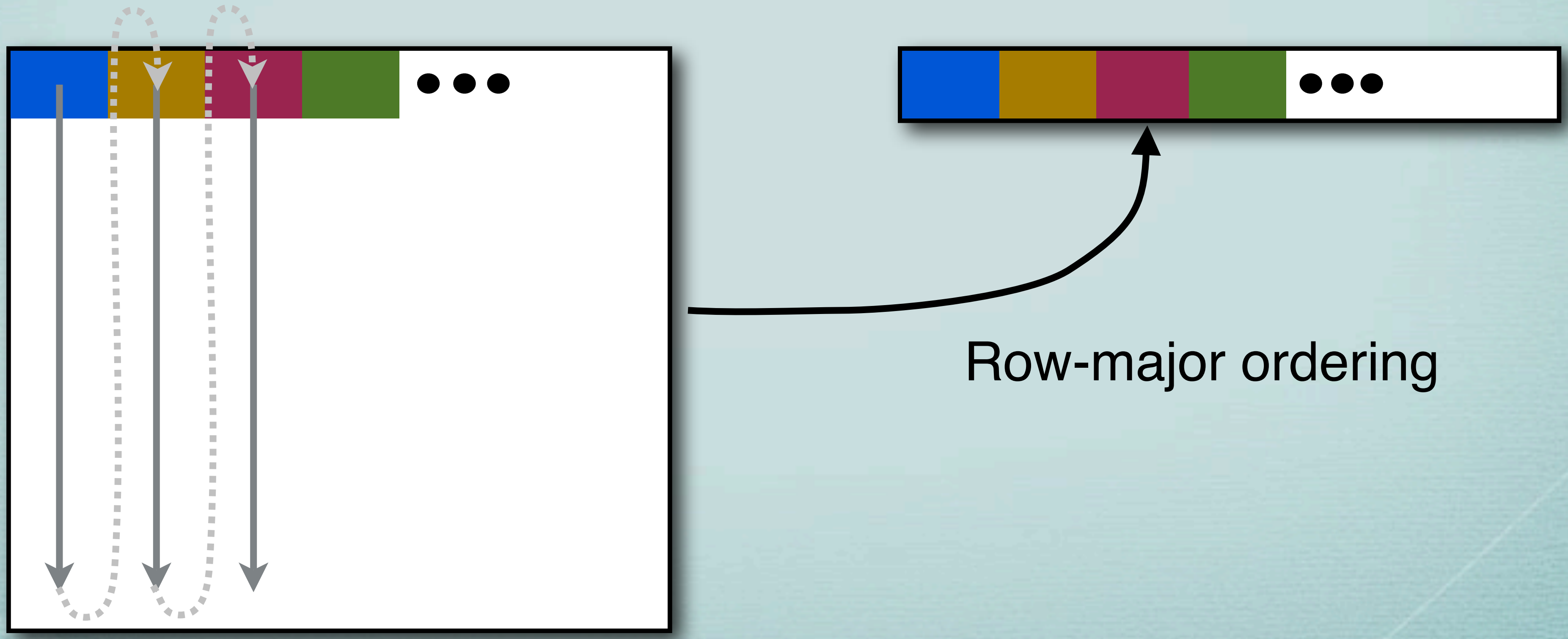
```
for (j=0; j < 100; j++)
    for (i=0; i < 5000; i++)
        x[i][j] = 2*x[i][j];
```



#9: Compiler Optimizations: Loop Interchange

(To Reduce Miss Rate)

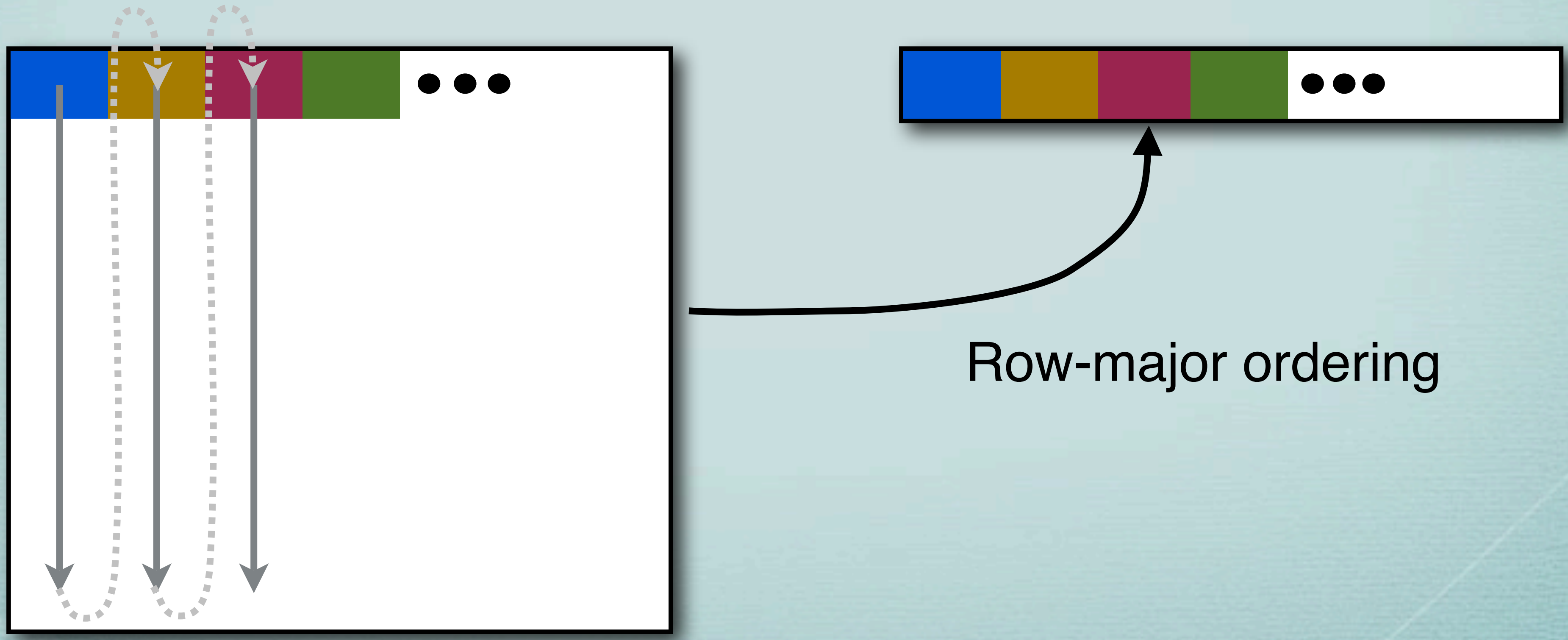
```
for (j=0; j < 100; j++)
   for (i=0; i < 5000; i++)
      x[i][j] = 2*x[i][j];
```



#9: Compiler Optimizations: Loop Interchange

(To Reduce Miss Rate)

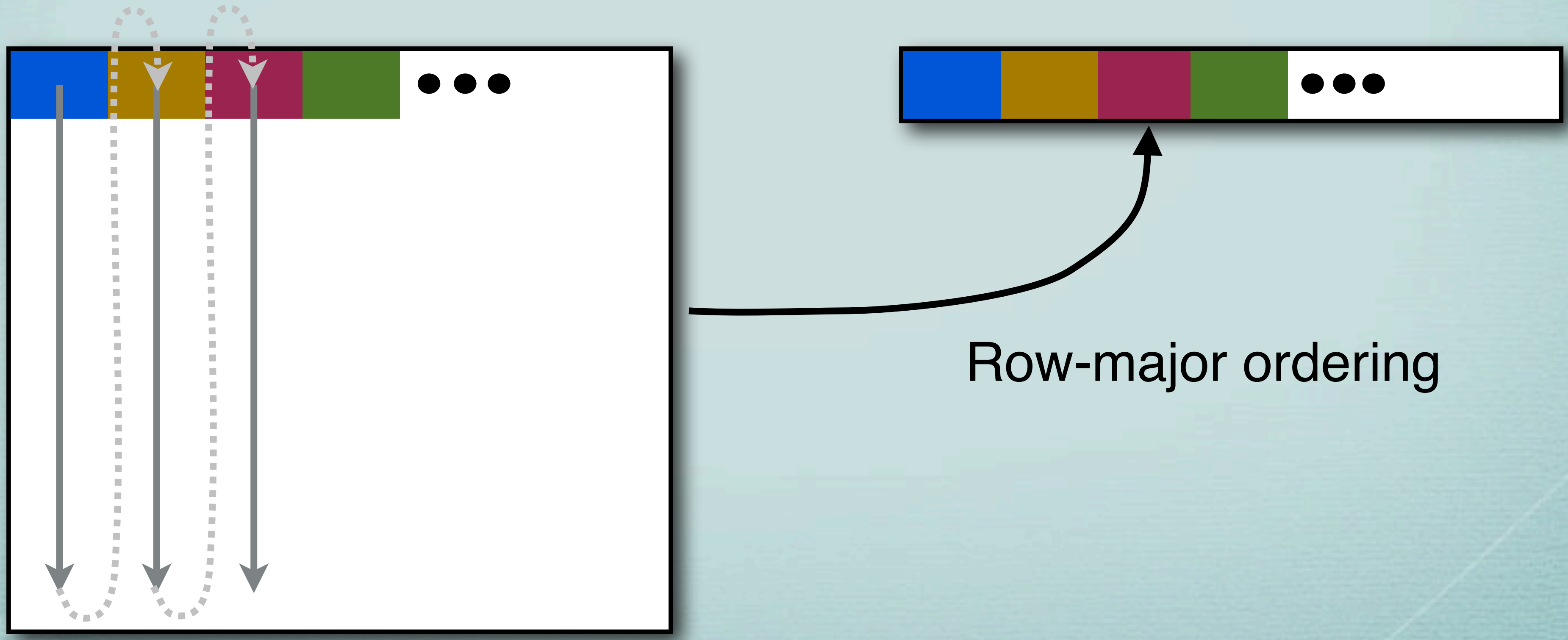
```
for (j=0; j < 100; j++)
    for (i=0; i < 5000; i++)
        x[i][j] = 2*x[i][j];
```



#9: Compiler Optimizations: Loop Interchange

(To Reduce Miss Rate)

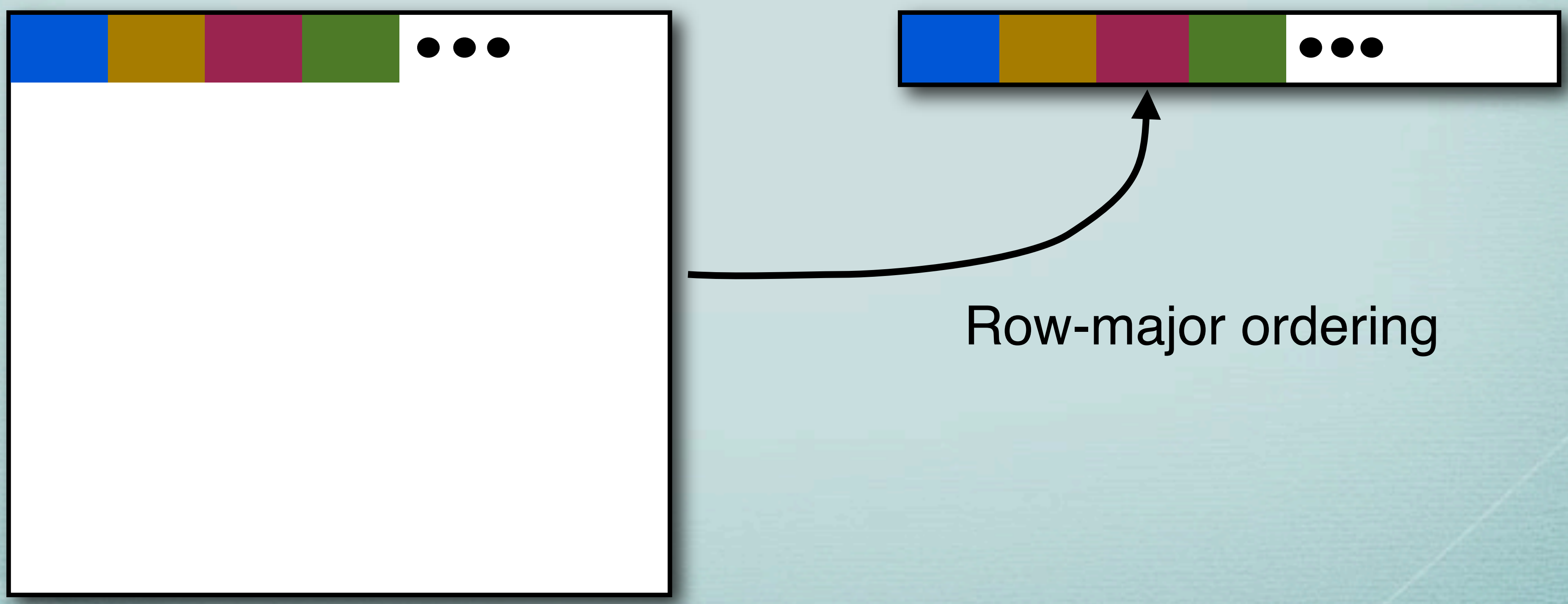
```
for (j=0; j < 100; j++)
    for (i=0; i < 5000; i++)
        x[i][j] = 2*x[i][j];
```



#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)

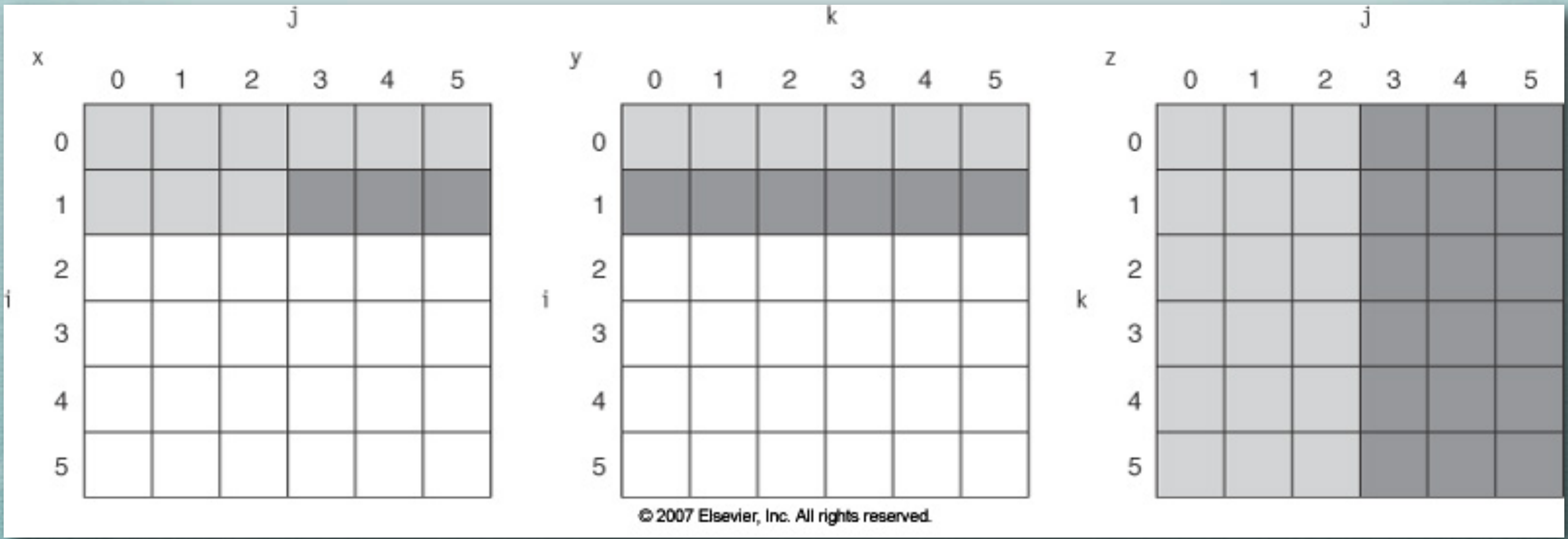
```
for (i=0; i < N; i++)
   for (j=0; j < N; j++)
   {
    r = 0.0;
    for (k=0; k < N; k++)
       r = r + y[i][k]*z[k][j];
    x[i][j] = r;
   }
```

Misses = anywhere between 0 and $(2N^3+N^2)$

#9: Compiler Optimizations: Blocking

(To Reduce Miss Rate)

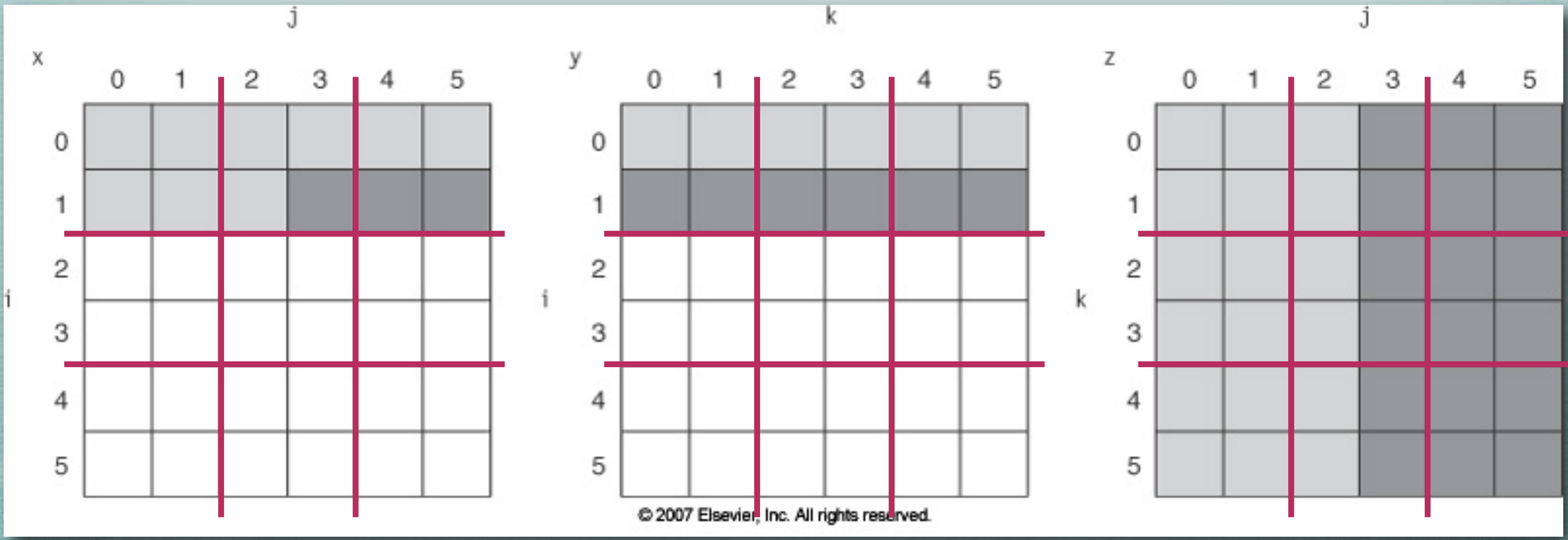
```
for (i=0; i < N; i++)
   for (j=0; j < N; j++)
   {
    r = 0.0;
    for (k=0; k < N; k++)
       r = r + y[i][k]*z[k][j];
    x[i][j] = r;
   }
```



#9: Compiler Optimizations: Blocking

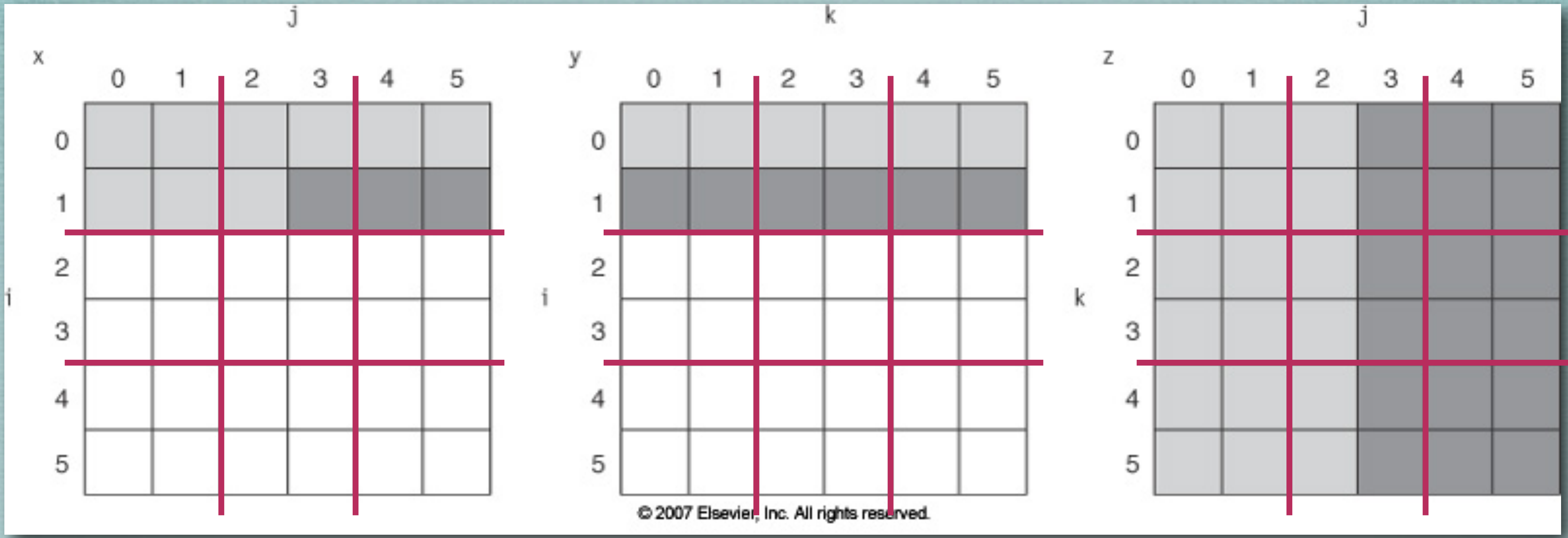
(To Reduce Miss Rate)

```
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

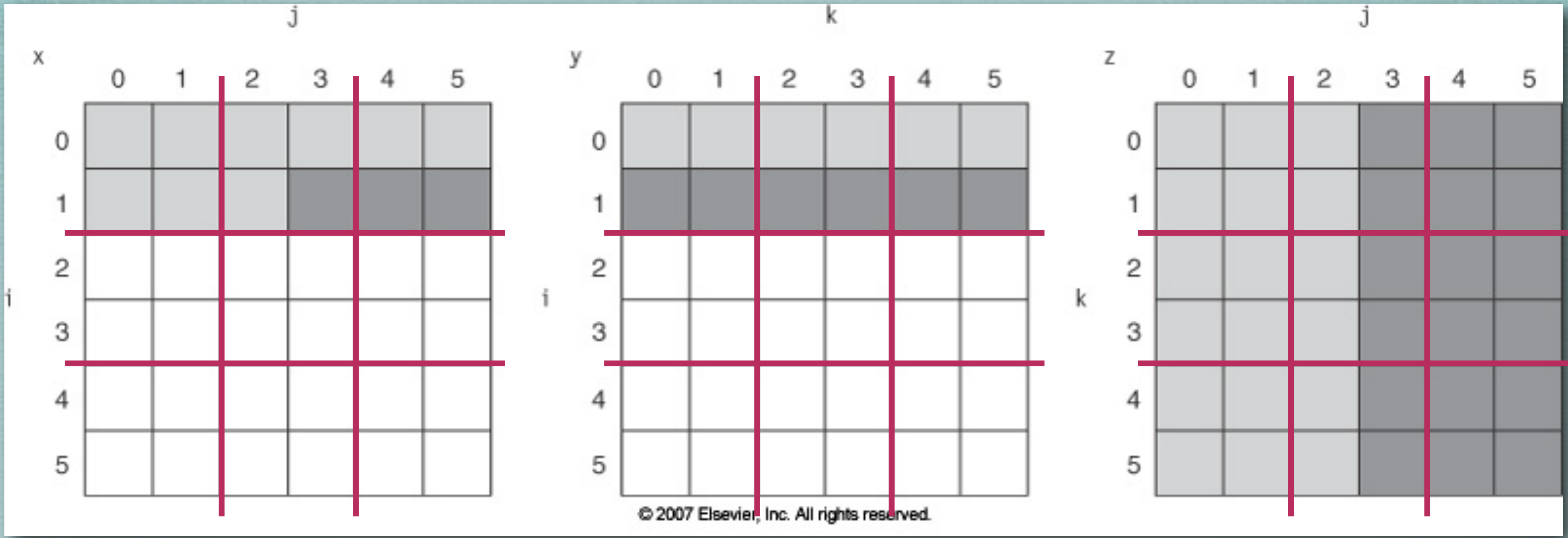
(To Reduce Miss Rate)



```
for (i=0; i < N; i++)
   for (j=0; j < N; j++)
   {
    R = ZEROS(2,2);
    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 (ii=0; ii < N/B; ii++)
   for (jj=0; jj < N/B; jj++)
   {
    R = ZEROS(2,2);
    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(2,2);
    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(2,2);
    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)

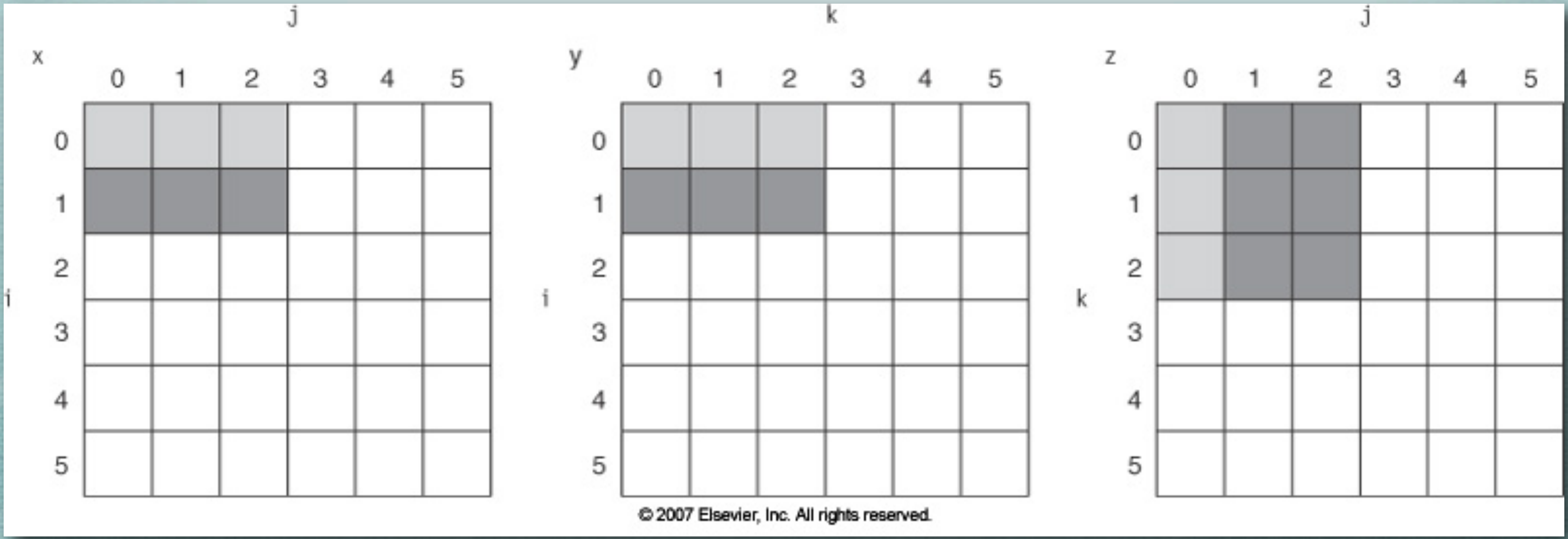
```
for (ii=0; ii < N/B; ii++)
   for (jj=0; jj < N/B; jj++)
   {
    R = ZEROS(2,2);
    for (kk=0; kk < N/B; kk++)
    {
     for (i=ii; i < ii+B; i++)
        for (j=jj; j < jj+B; j++)
           for (k=kk; k < kk+B; k++)
              R[i][j] = R[i][j] + y[i][k]*z[k][j]
    }
    X[ii][jj] = R;
   }
```

```
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;
         }
```



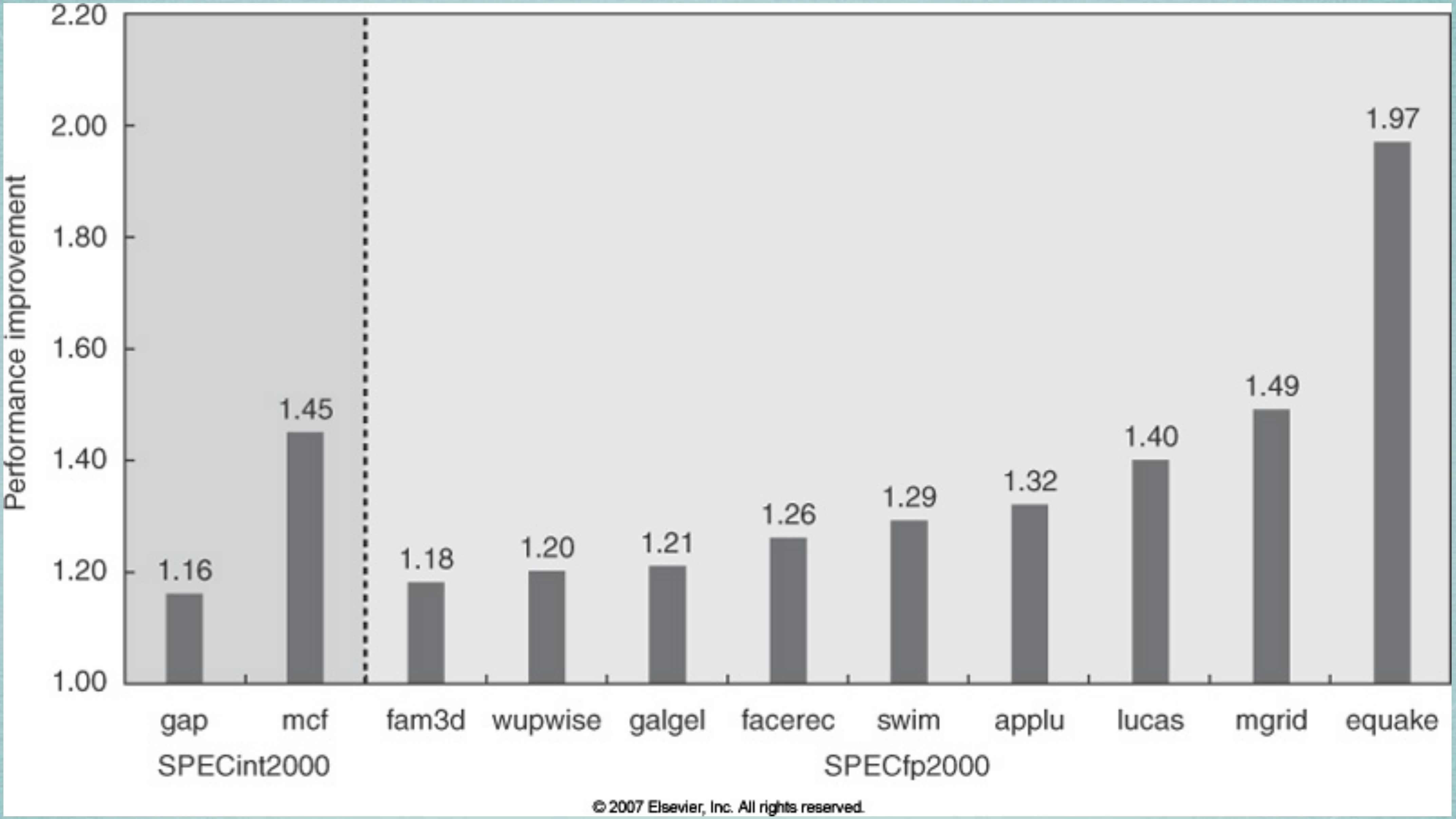
#10: Hardware Prefetching

(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)



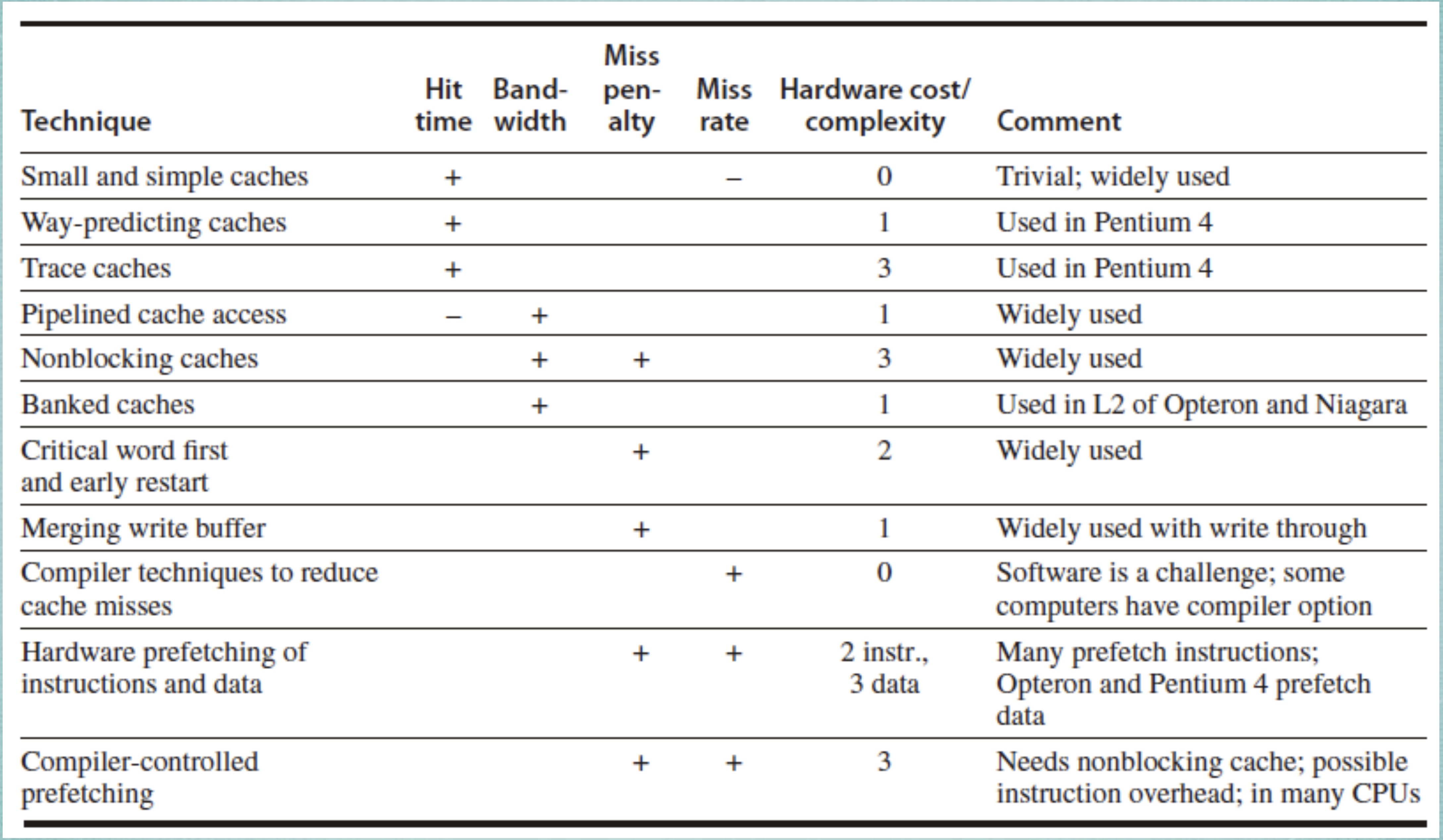
Speedup due to hardware prefetching on Pentium 4

#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

MEMORY TECHNOLOGY AND OPTIMIZATIONS

Memory Types

- SRAM
  - ⋆ static RAM
  - ⋆ uses about six transistors per bit
  - ⋆ access type close to one cycle
  - ⋆ used for caches
- DRAM
  - ⋆ dynamic RAM
  - ⋆ much more compact, needs one transistor + one capacitor per bit
  - ⋆ substantially slower than SRAM
  - ⋆ used for main memory
- Characteristics
  - ⋆ bandwidth
  - ⋆ latency (*access time* vs *cycle time*)

DRAM Sales

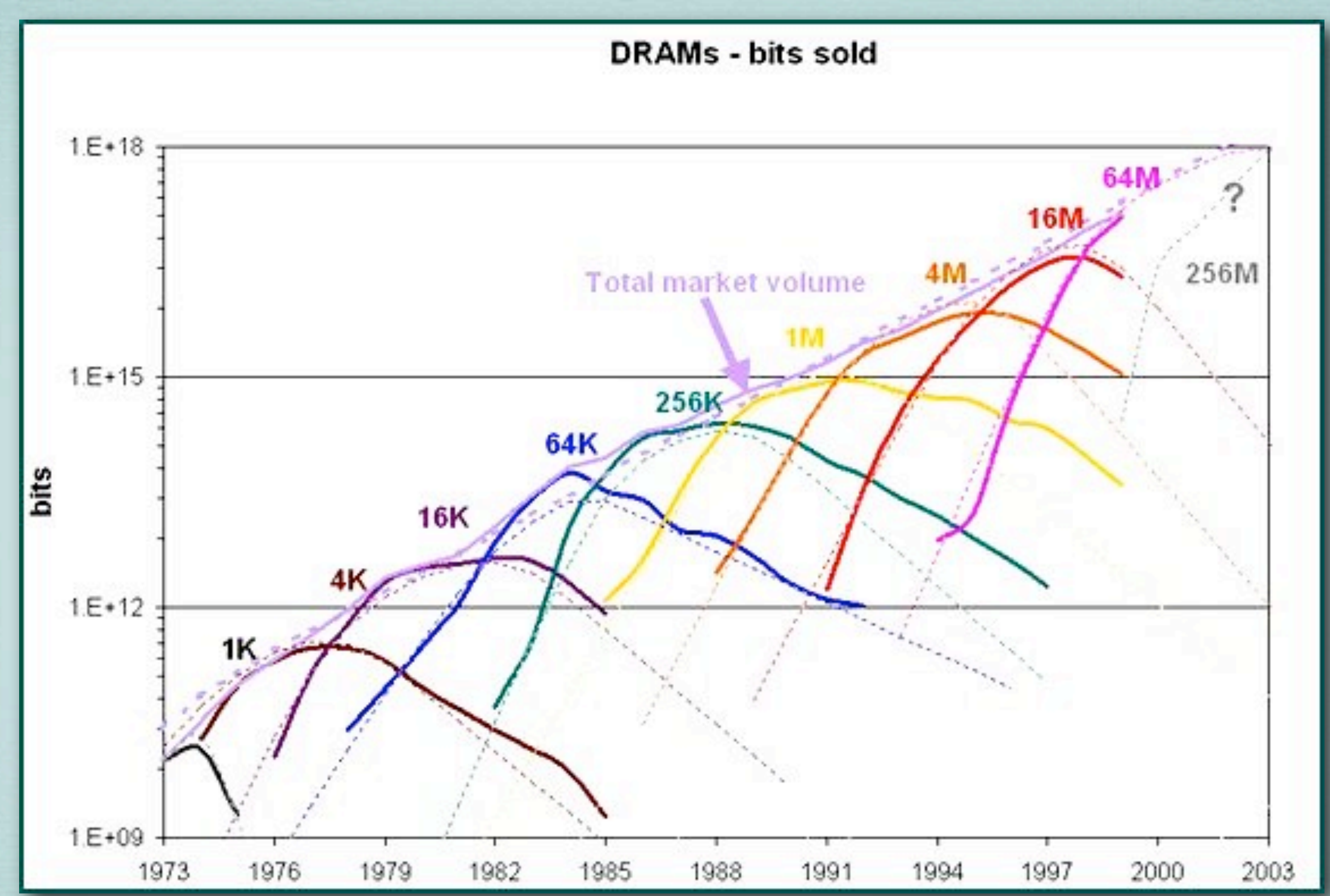
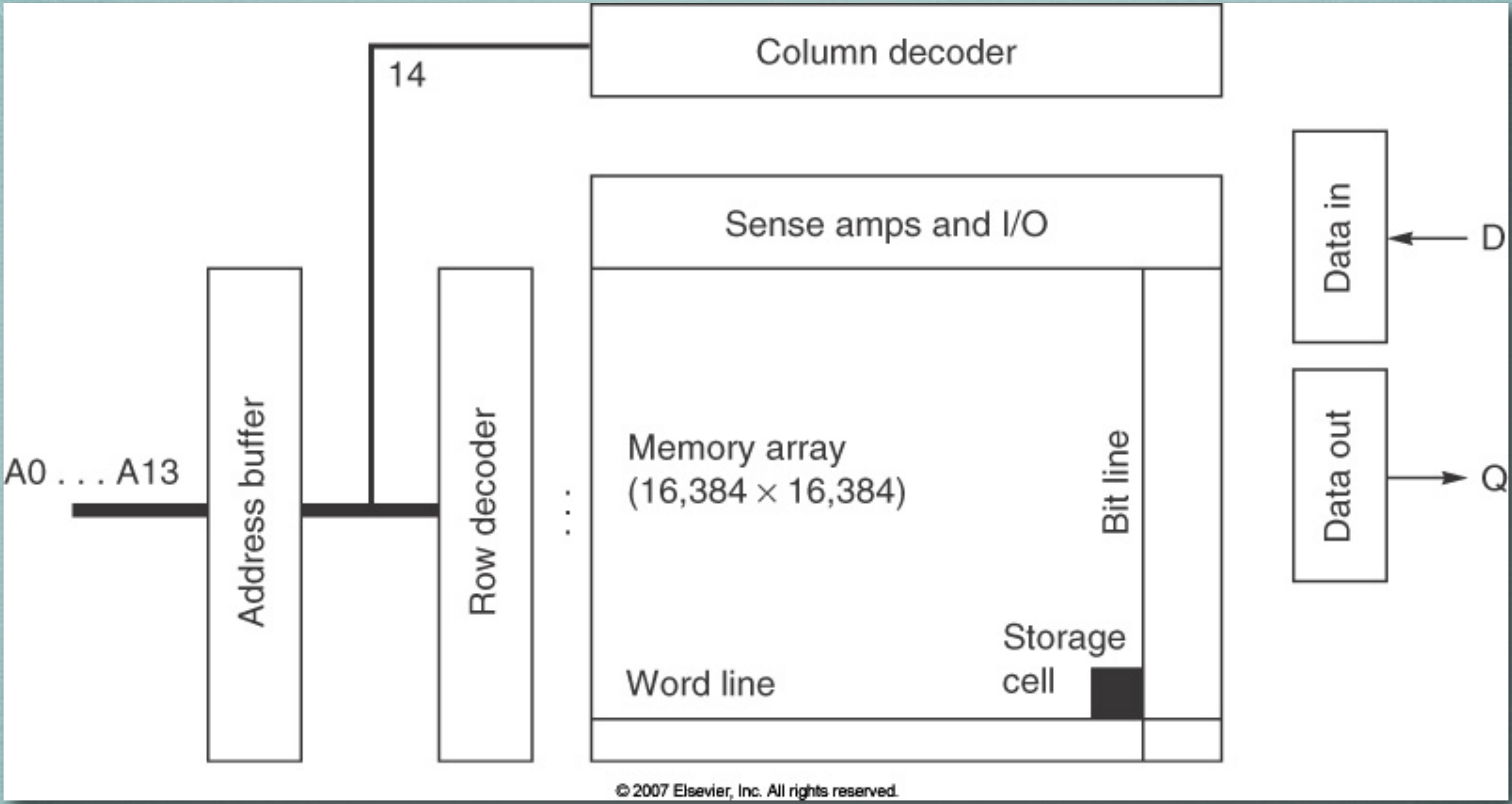


Figure from the web-site of Christian-Albrechts-University of Kiel, Germany

DRAM Technology



DRAM Technology

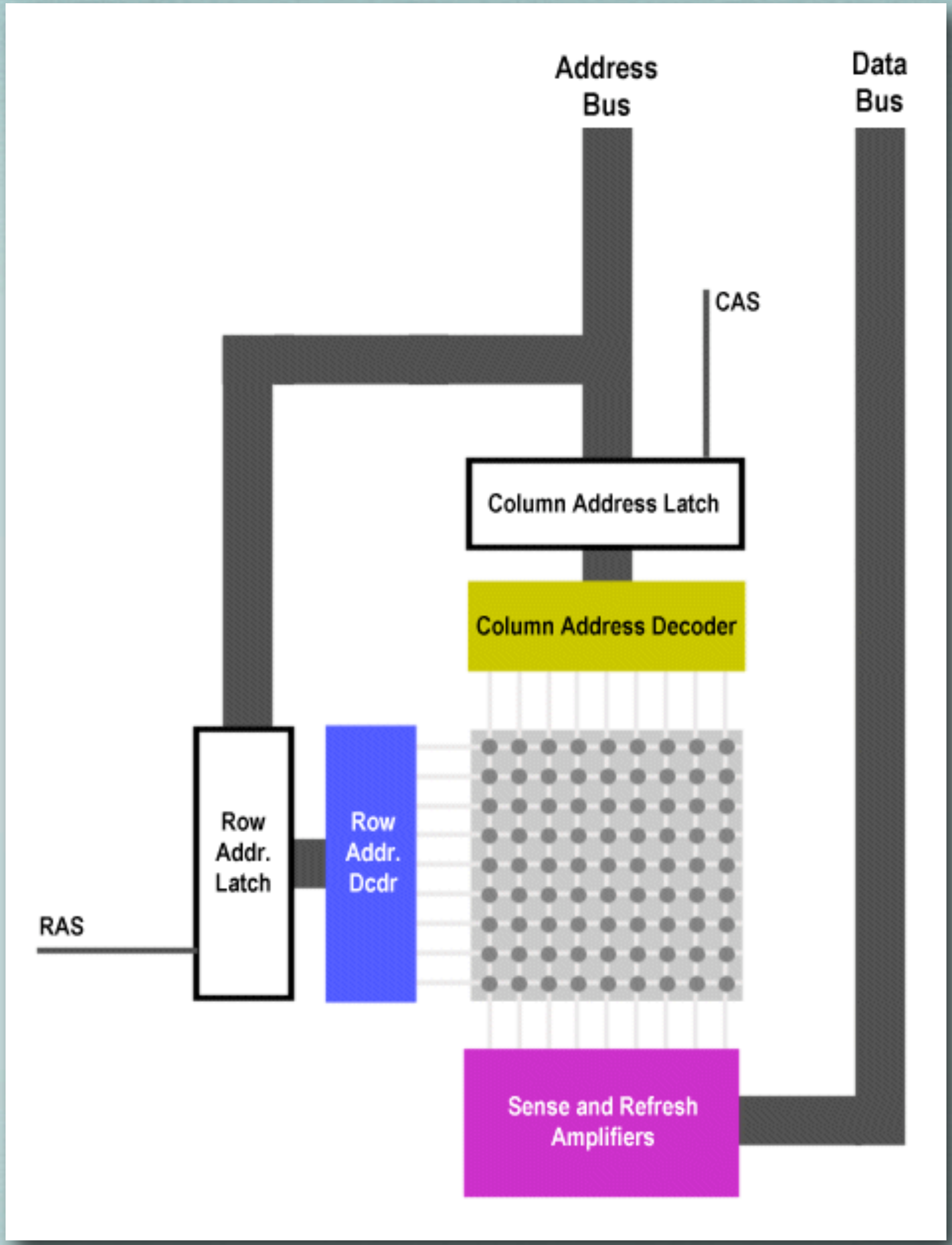


Figure from the web-site of Ars Technica

DRAM Technology

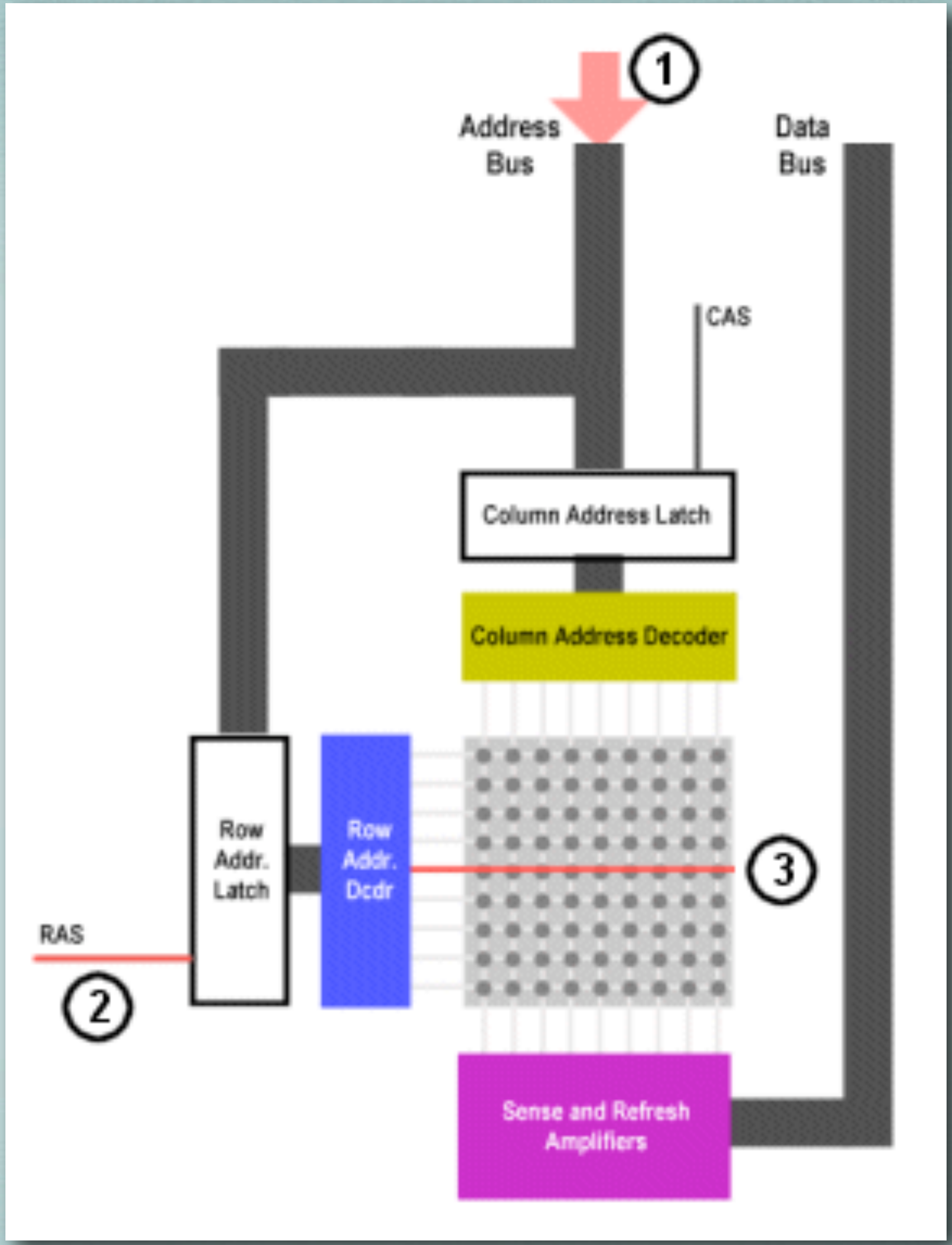


Figure from the web-site of Ars Technica

DRAM Technology

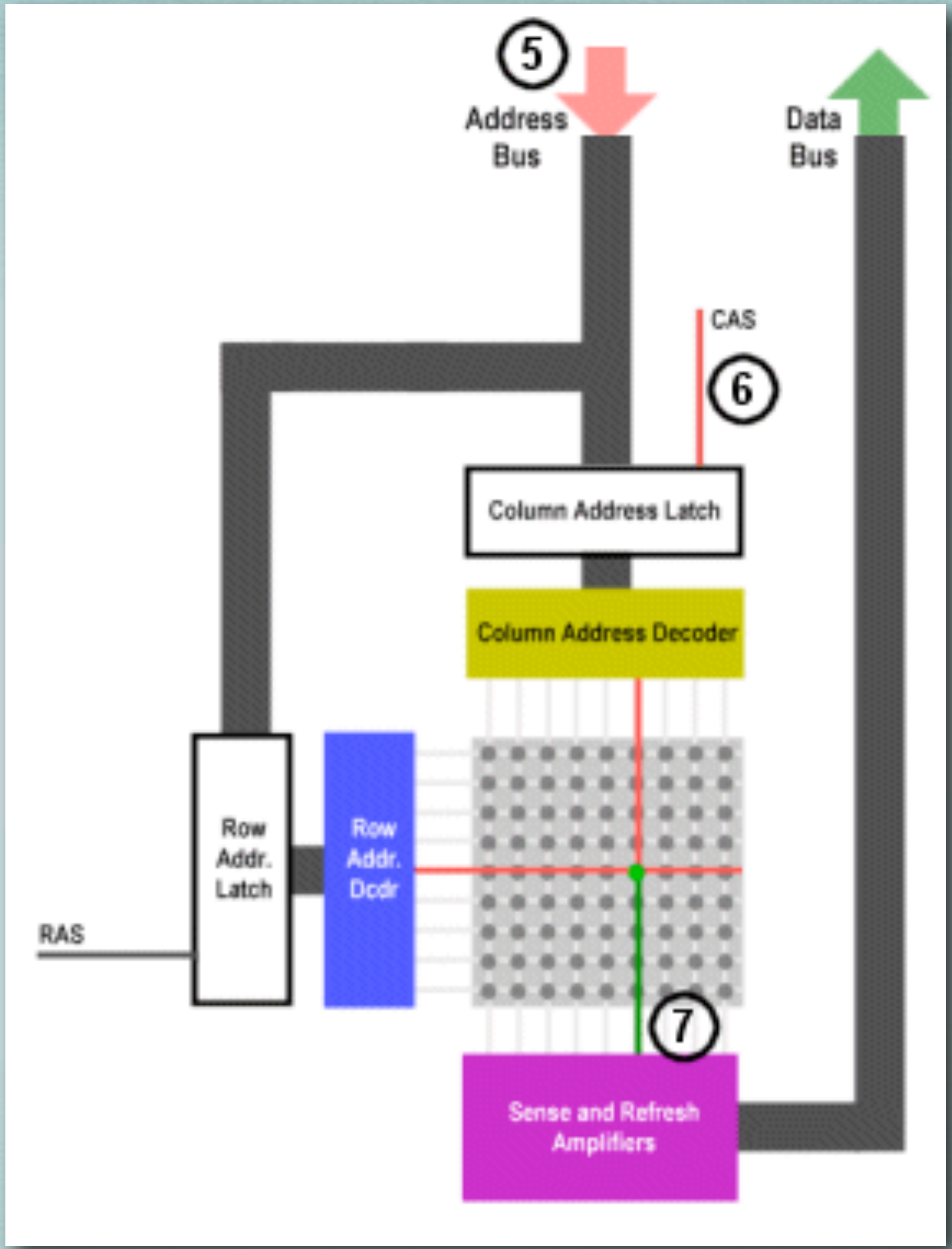


Figure from the web-site of Ars Technica

DRAM Technology

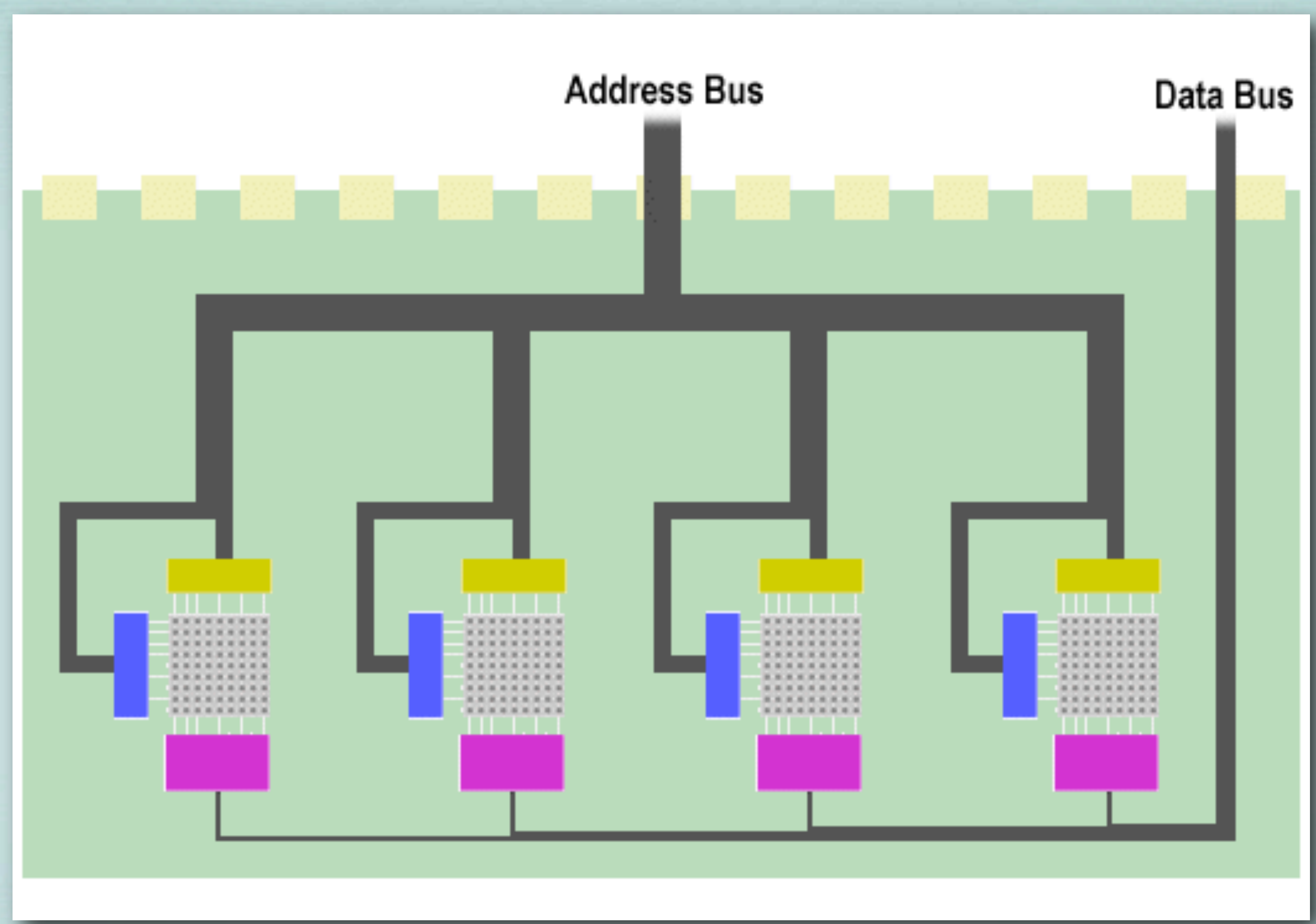
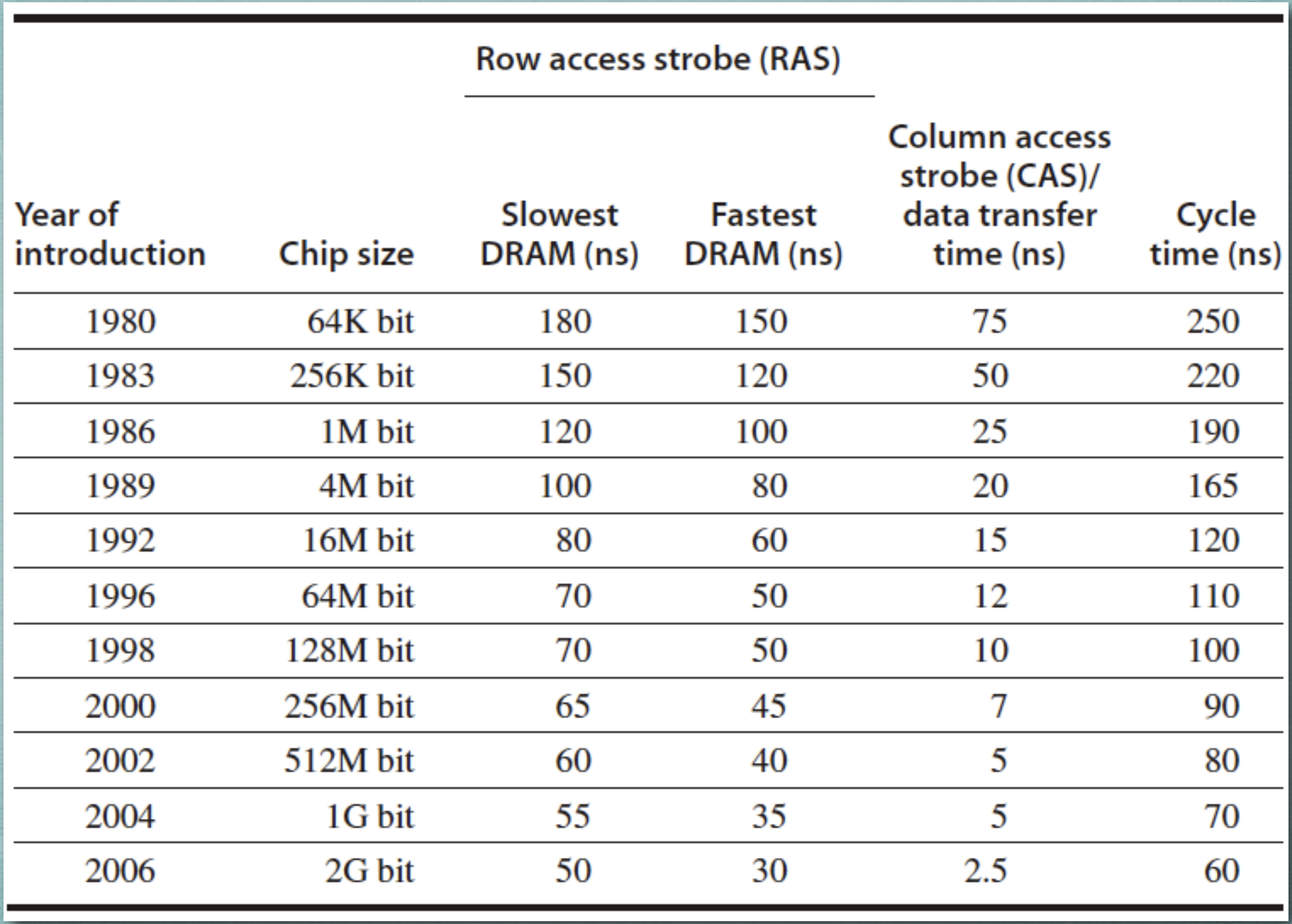


Figure from the web-site of Ars Technica

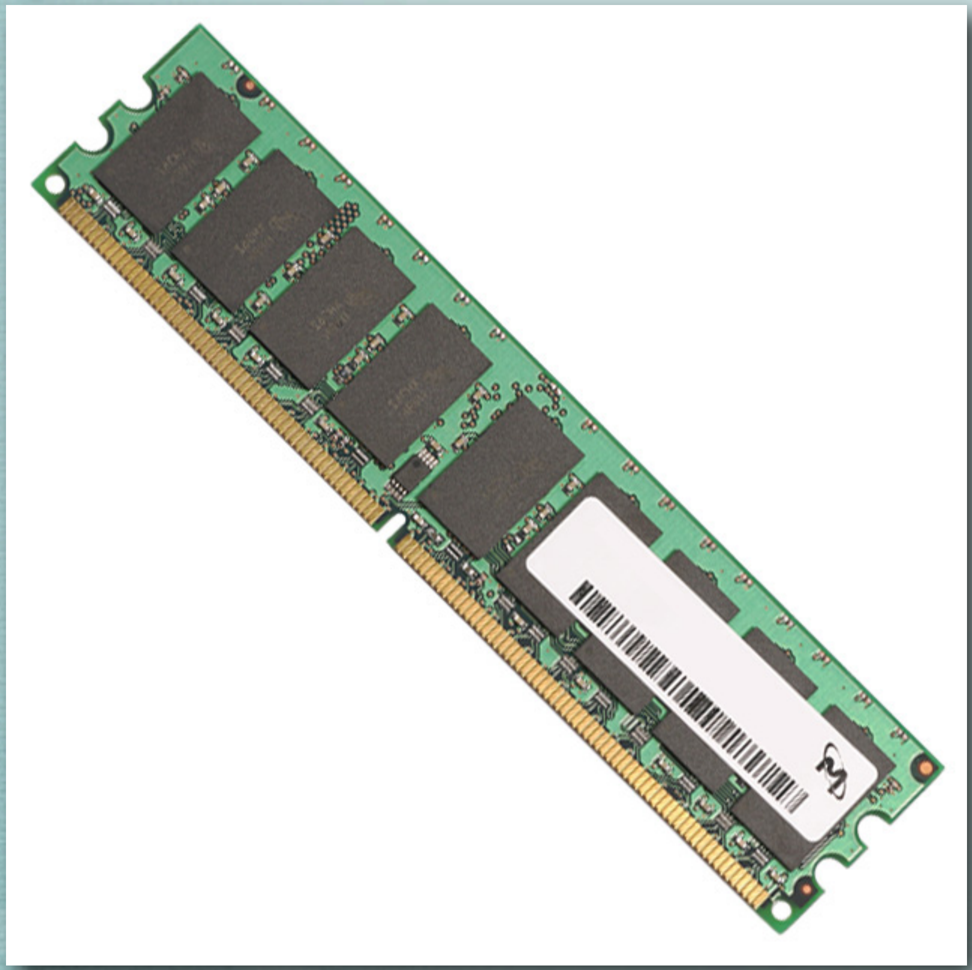
DRAM Speeds

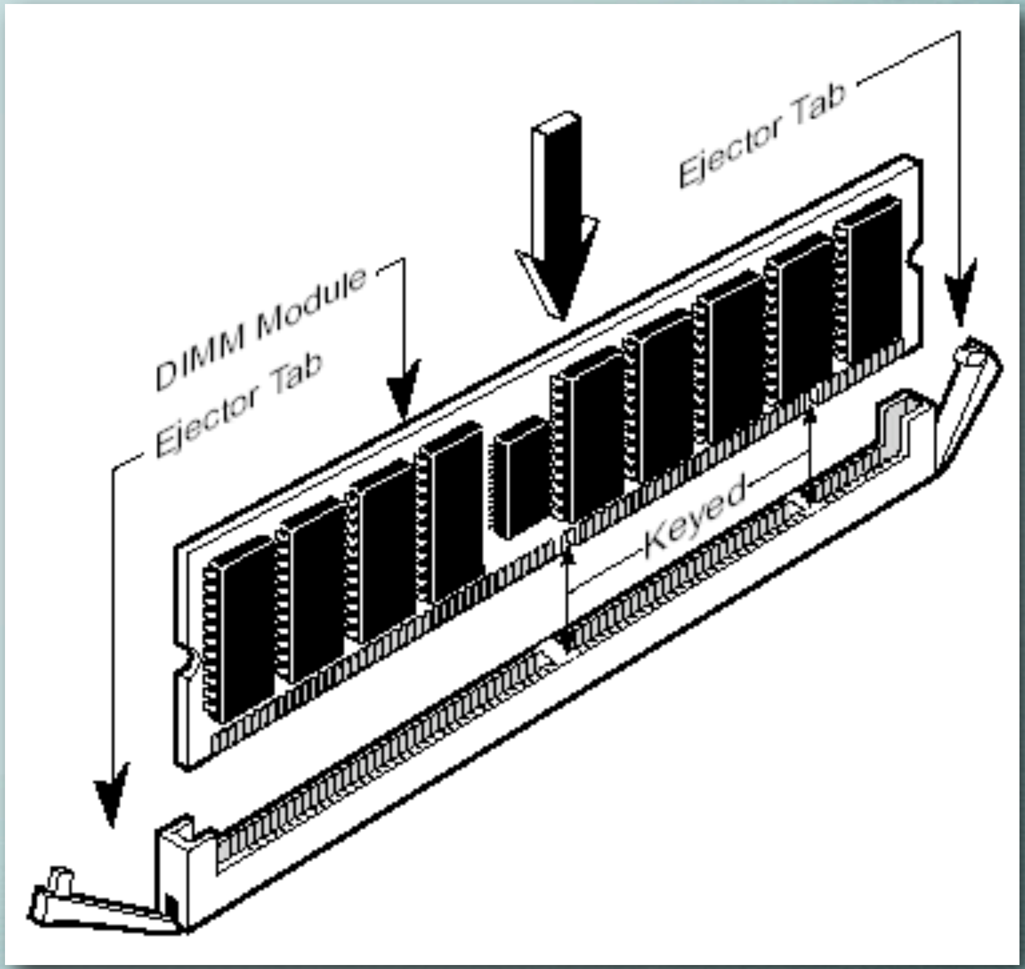


		Row access strobe (RAS)			
Year of introduction	Chip size	Slowest DRAM (ns)	Fastest DRAM (ns)	Column access strobe (CAS)/ data transfer time (ns)	Cycle time (ns)
1980	64K bit	180	150	75	250
1983	256K bit	150	120	50	220
1986	1M bit	120	100	25	190
1989	4M bit	100	80	20	165
1992	16M bit	80	60	15	120
1996	64M bit	70	50	12	110
1998	128M bit	70	50	10	100
2000	256M bit	65	45	7	90
2002	512M bit	60	40	5	80
2004	1G bit	55	35	5	70
2006	2G bit	50	30	2.5	60

DIMMs = Packaged DRAMs

- DIMM = *Dual Inline Memory Modules*
  - ★ SODIMM = *Small Outline DIMM*

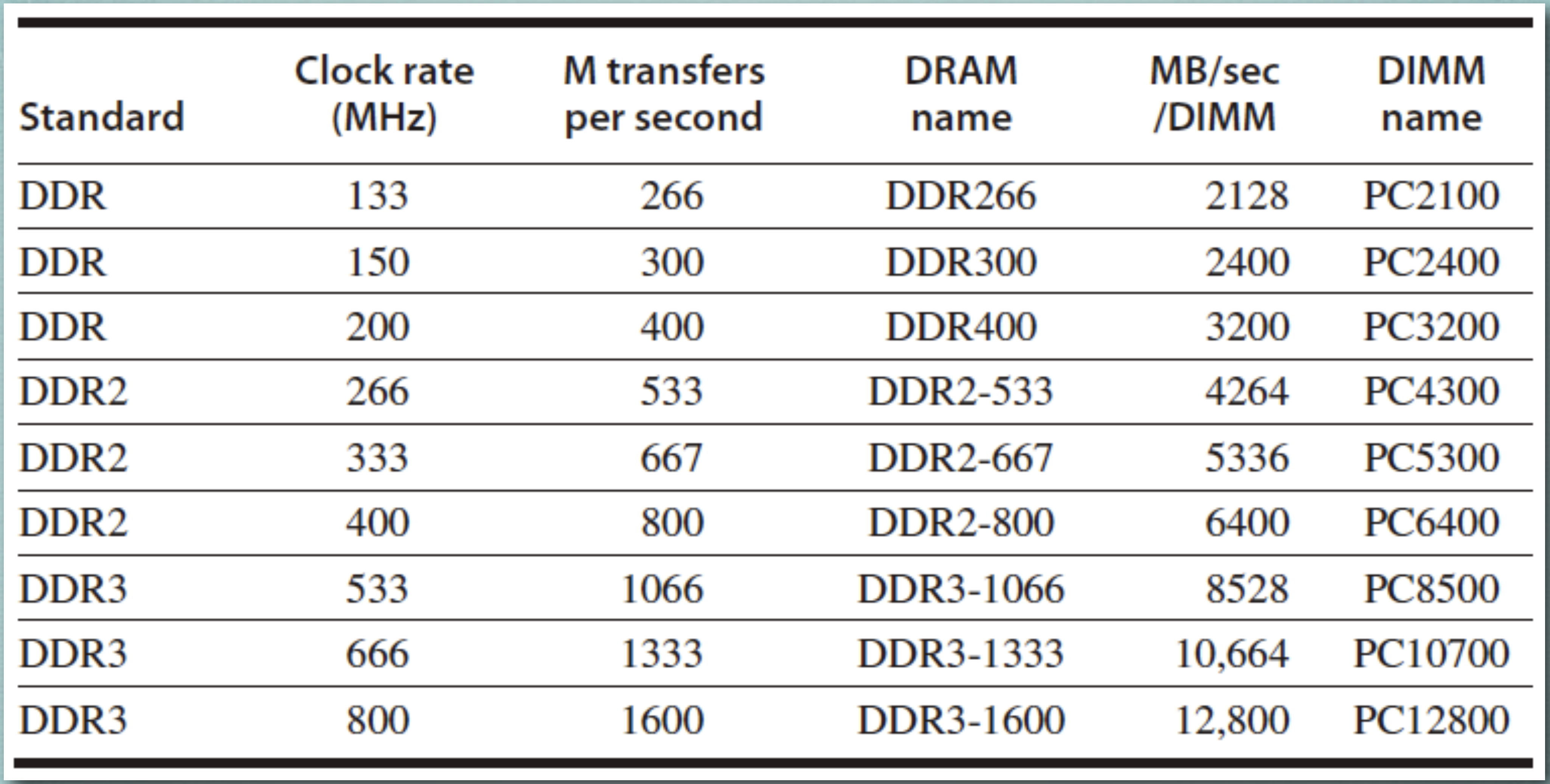




DRAM Bandwidth Optimizations

- DRAMs consist of multiple modules (1-4M bits)
- Fast-page mode
  - ★ repeated access to row buffer without incurring row-access time
  - ★ obsoleted by synchronous access (modern DRAM still support it, though)
- Synchronous DRAM (SDRAM)
  - ★ after initial latency, can read several bytes with one cycle latency (bus cycle)
- Double Data Rate SDRAM (DDR SDRAM)
  - ★ typically, use multiple banks internally

DDR DRAMs and DIMMs



Standard	Clock rate (MHz)	M transfers per second	DRAM name	MB/sec /DIMM	DIMM name
DDR	133	266	DDR266	2128	PC2100
DDR	150	300	DDR300	2400	PC2400
DDR	200	400	DDR400	3200	PC3200
DDR2	266	533	DDR2-533	4264	PC4300
DDR2	333	667	DDR2-667	5336	PC5300
DDR2	400	800	DDR2-800	6400	PC6400
DDR3	533	1066	DDR3-1066	8528	PC8500
DDR3	666	1333	DDR3-1333	10,664	PC10700
DDR3	800	1600	DDR3-1600	12,800	PC12800

VIRTUAL MEMORY
AND
VIRTUAL MACHINES

Virtual Machines: An Old Idea

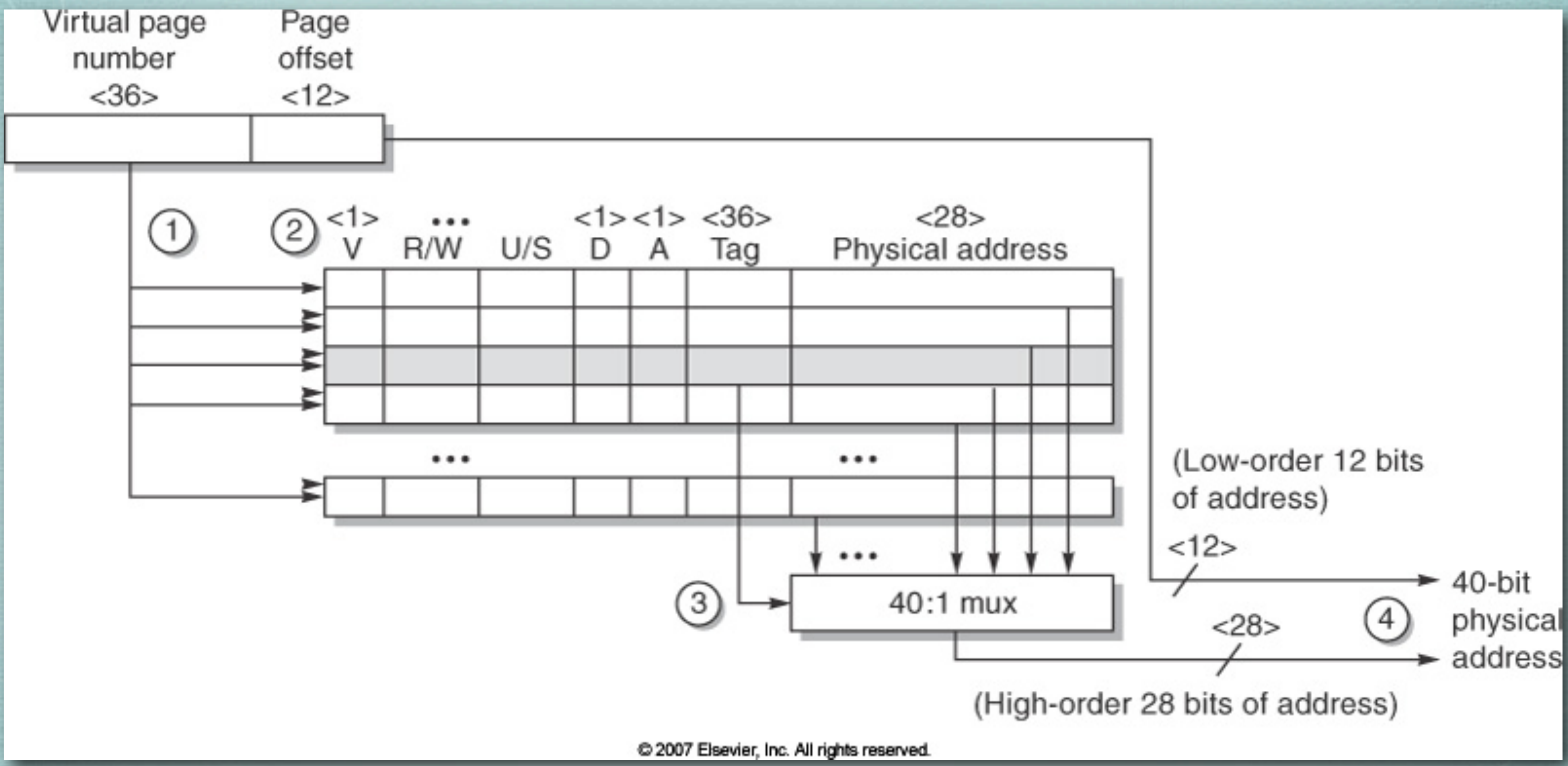
A virtual machine is taken to be an efficient, isolated duplicate *of the real machine. We explain these notions through the idea of a virtual machine monitor (VMM) ... a VMM has three essential characteristics. First, the VMM provides an environment for programs which is essentially identical with the 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 Memory

- Two process modes
  - ⋆ *user* mode and *kernel* mode
- Read-only state
  - ⋆ user processes may not modify state bits such as user/kernel bit, exception enable/disable bit, memory protection information, etc.
- Mechanisms to go from user to kernel level and vice-versa
  - ⋆ system calls
- Mechanisms to limit memory access

Virtual Memory



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

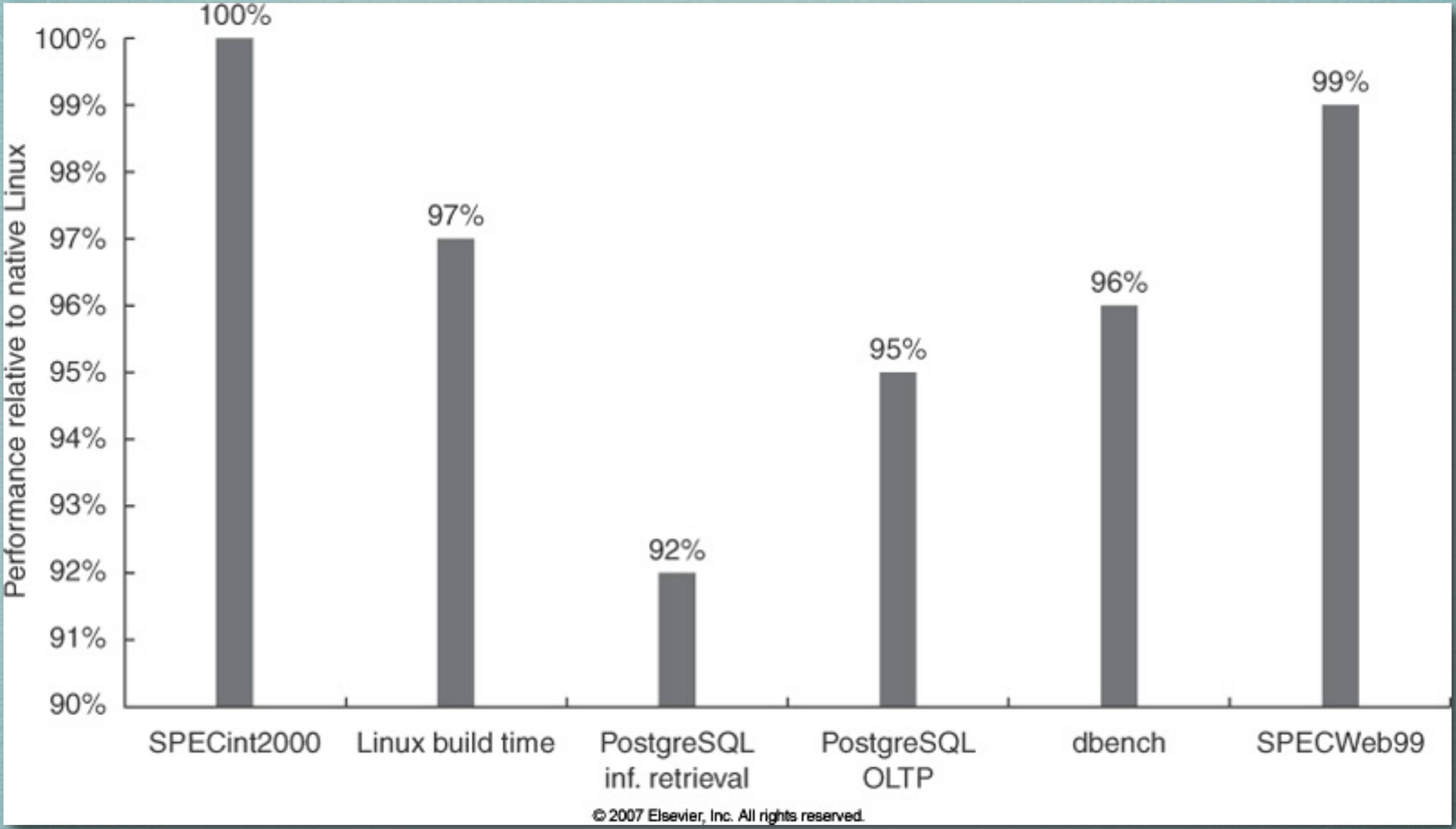
Popularity of Virtual Machines: II

- 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

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



NEXT:
PARALLEL COMPUTING
OR
SCIENTIFIC COMPUTING
OR
COMPILER TECHNIQUES