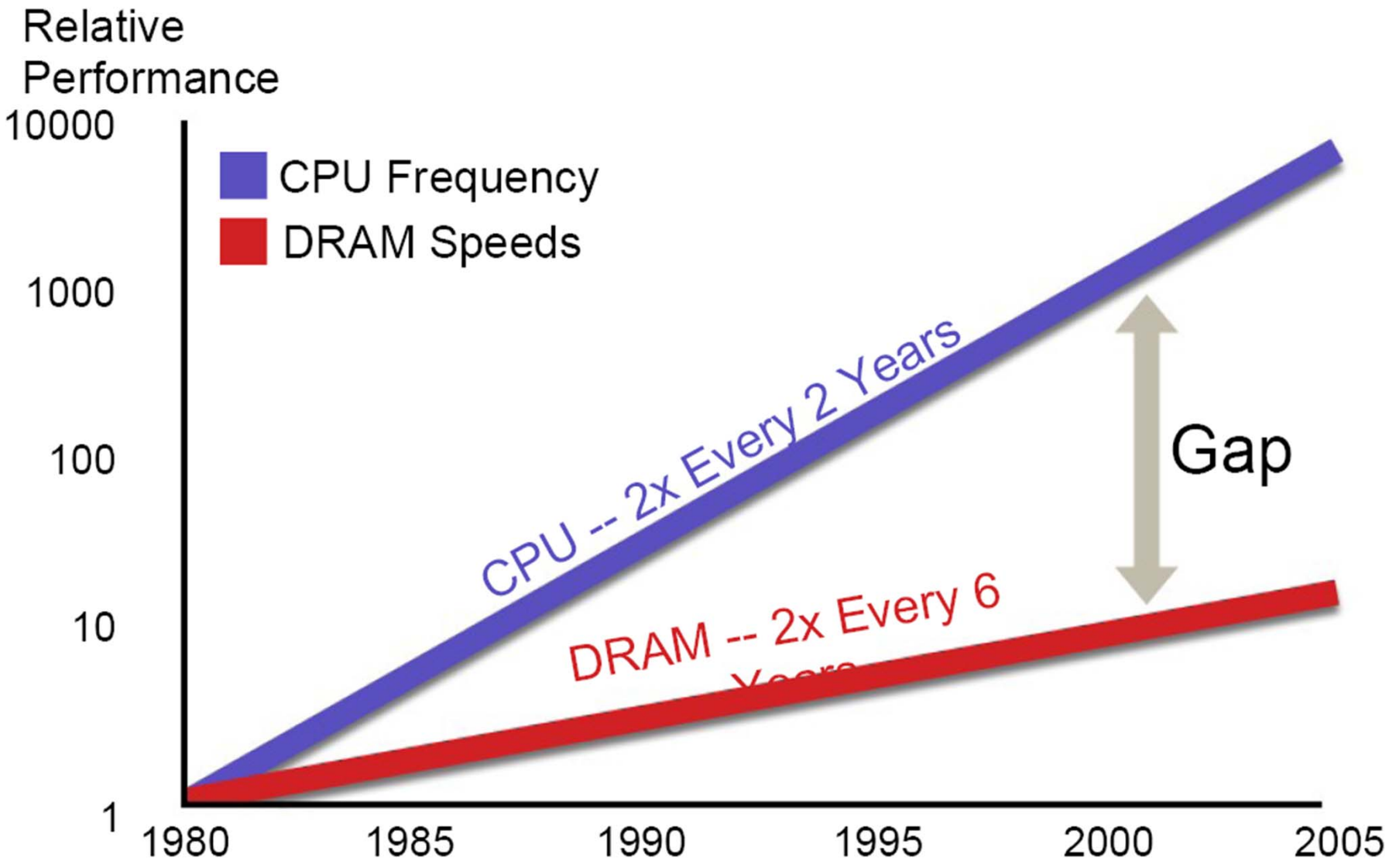
The Memory Hierarchy

Chapter 5 of B&O

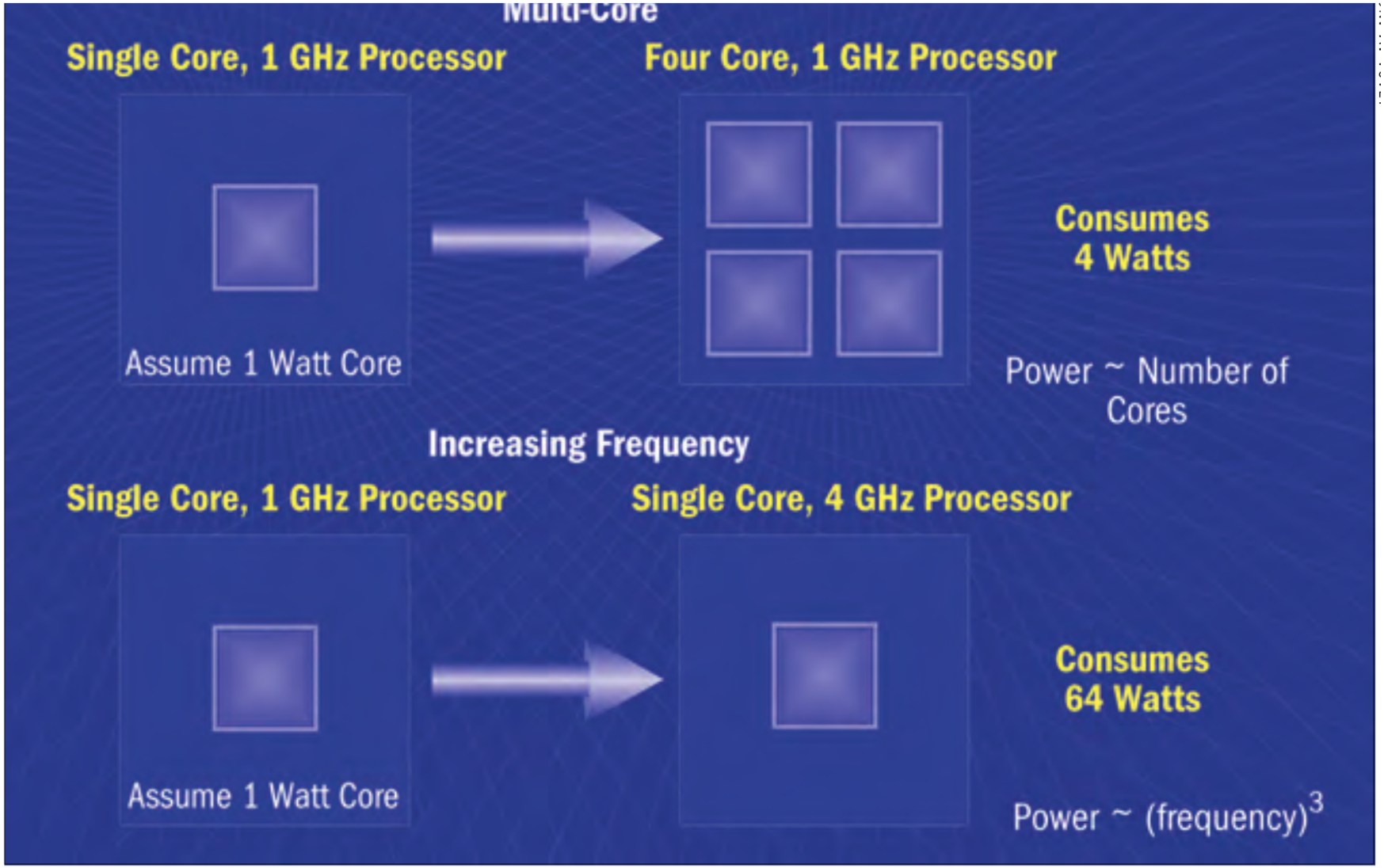
Some notes adopted from Bryant and O'Hallaron

The CPU-Memory Gap



http://blogs.sun.com/toddjobson/entry/processors_and_performance_chips_mips

The Energy Wall (SCiDAC)



The Road to Exascale (SCiDac)

USTRATION: A. TOVEY

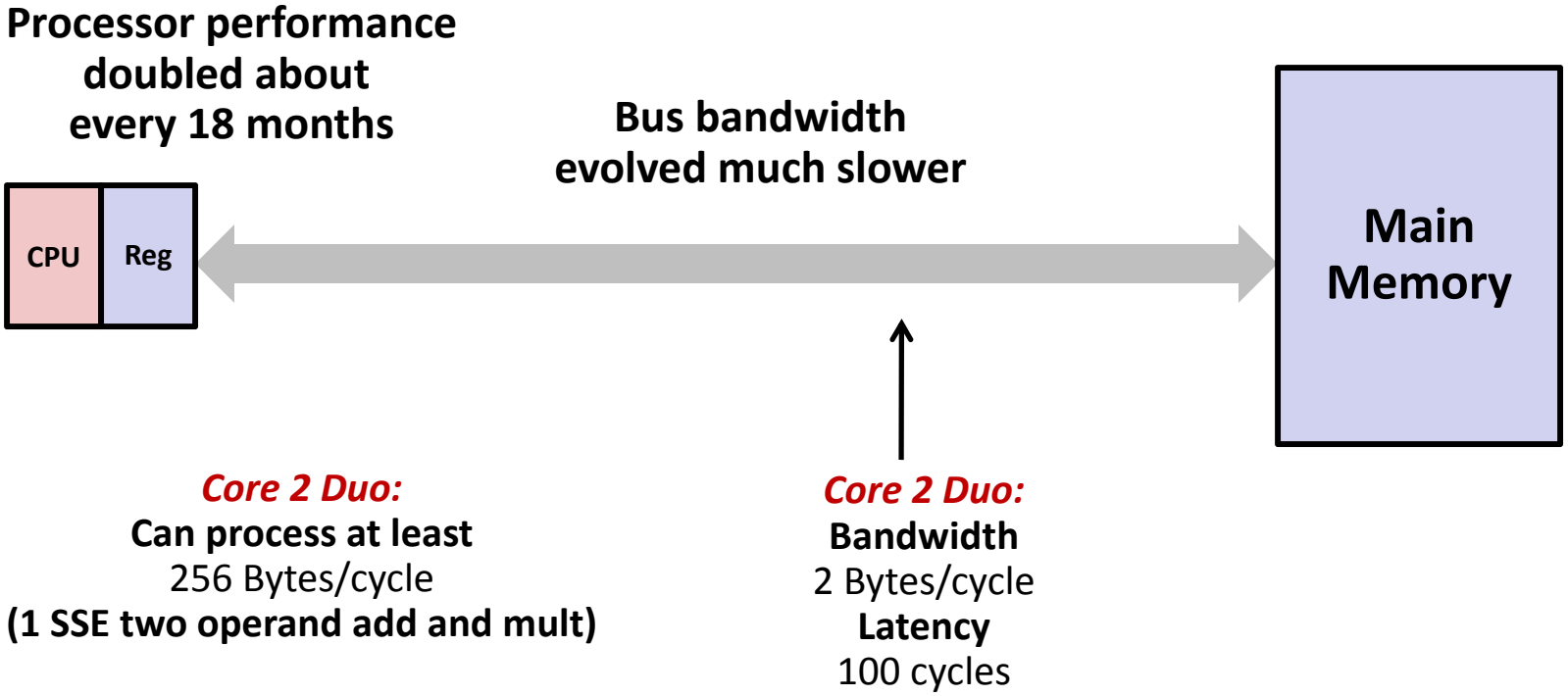
Systems	2009	2011	2015	2018
System Peak Flops/s	2 Peta	20 Peta	100-200 Peta	1 Exa
System Memory	0.3 PB	1 PB	5 PB	10 PB
Node Performance	125 GF	200 GF	400 GF	1-10 TF
Node Memory BW	25 GB/s	40 GB/s	100 GB/s	200-400 GB/s
Node Concurrency	12	32	O(100)	O(1000)
Interconnect BW	1.5 GB/s	10 GB/s	25 GB/s	50 GB/s
System Size (Nodes)	18,700	100,000	500,000	O(Million)
Total Concurrency	225,000	3 Million	50 Million	O(Billion)
Storage	15 PB	30 PB	150 PB	300 PB
I/O	0.2 TB/s	2 TB/s	10 TB/s	20 TB/s
MTTI	Days	Days	Days	O(1Day)
Power	6 MW	~10 MW	~10 MW	~20 MW

Figure 4. *Estimated specifications for Exascale Roadmap systems based on expected technologies in the respective*

Random-Access Memory (RAM)

- Key features
  - RAM is packaged as a chip.
  - Basic storage unit is a cell.
  - Multiple RAM chips form a memory.
- Static RAM (SRAM)
  - Each cell stores bit with a six-transistor circuit.
  - Retains value indefinitely, as long as it is kept powered.
  - Relatively insensitive to disturbances such as electrical noise.
  - Faster and more expensive than DRAM.
- Dynamic RAM (DRAM)
  - Each cell stores bit with a capacitor and transistor.
  - Value must be refreshed every 10-100 ms.
  - Sensitive to disturbances.
  - Slower and cheaper than SRAM.

Problem: Processor-Memory Bottleneck

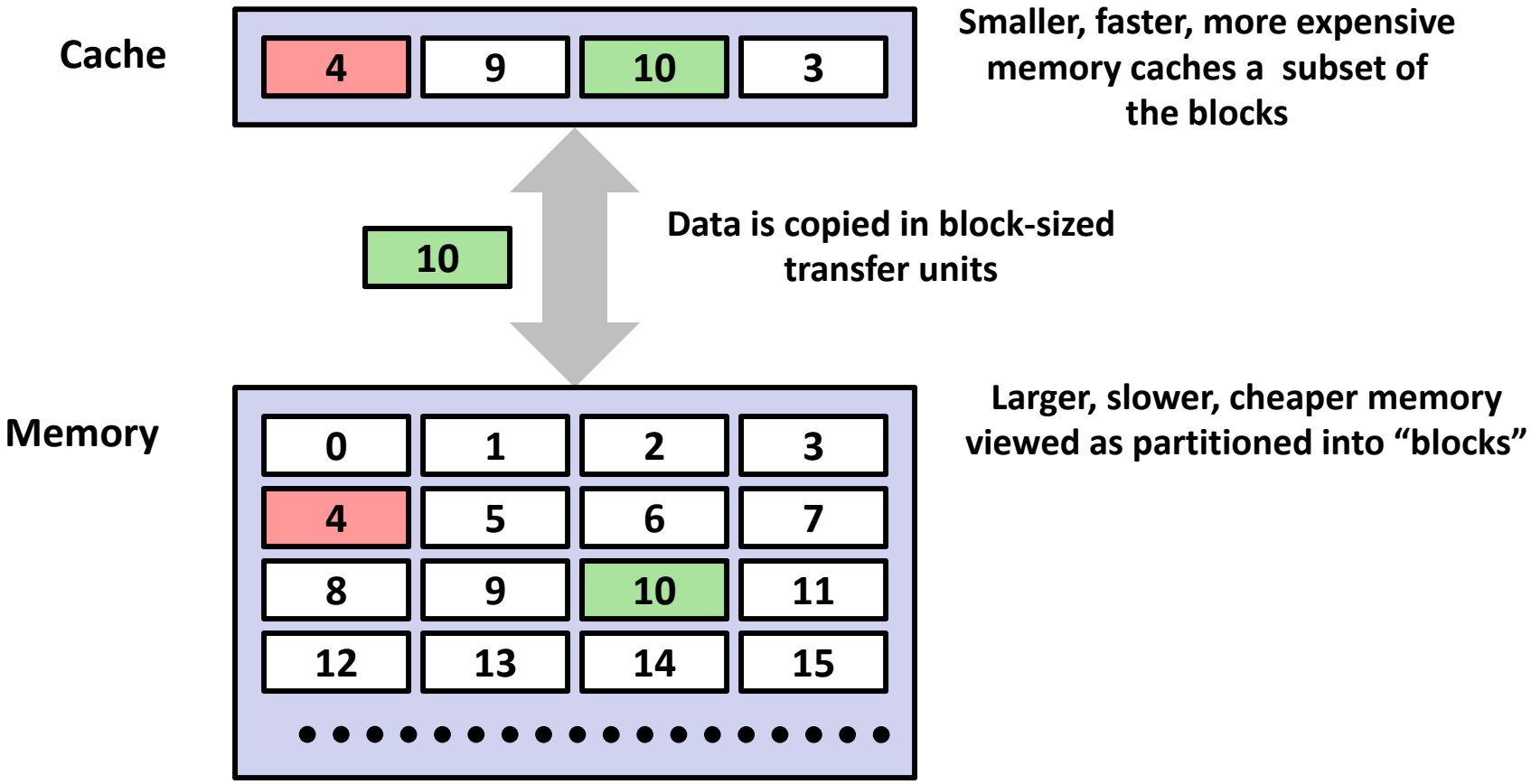


Solution: Caches

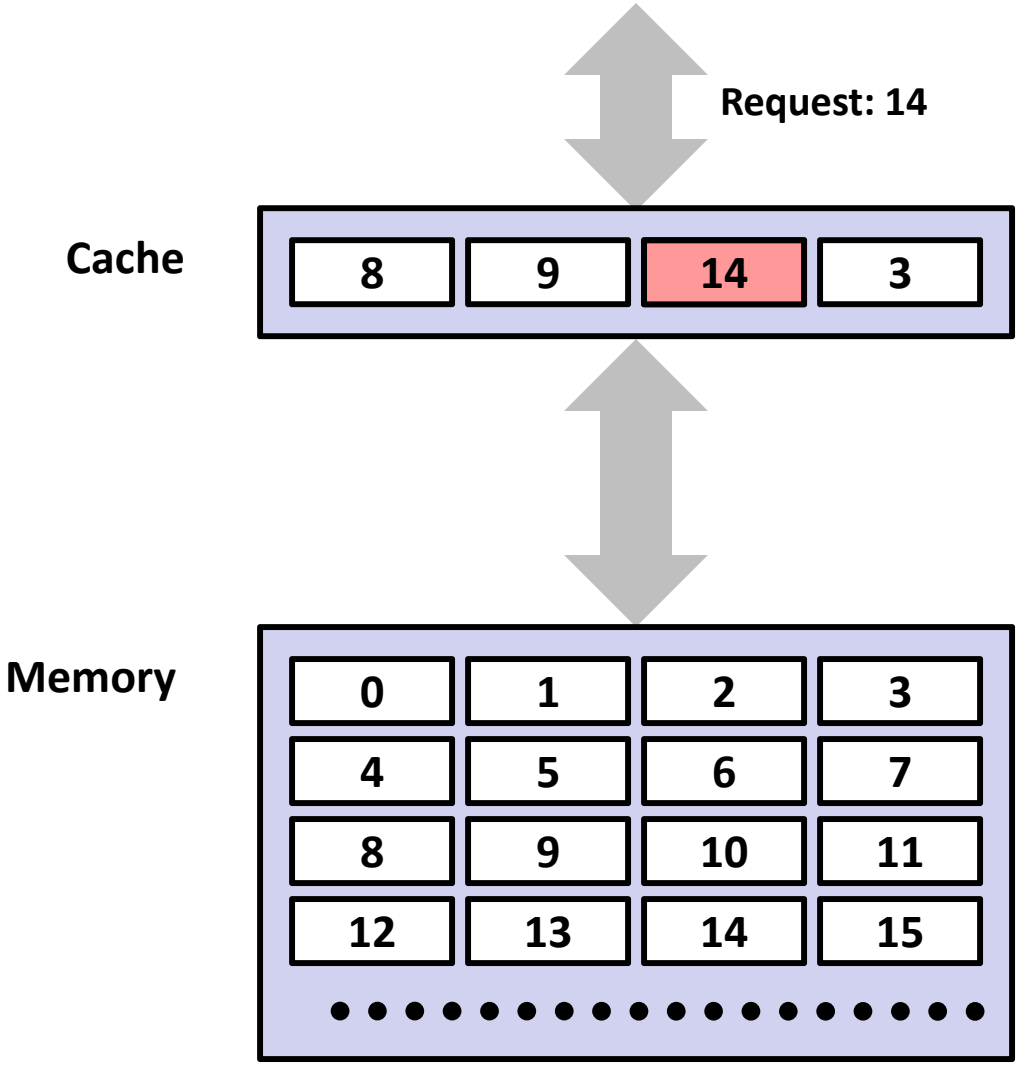
Cache

- Definition: Computer memory with short access time used for the storage of frequently or recently used instructions or data

General Cache Mechanics



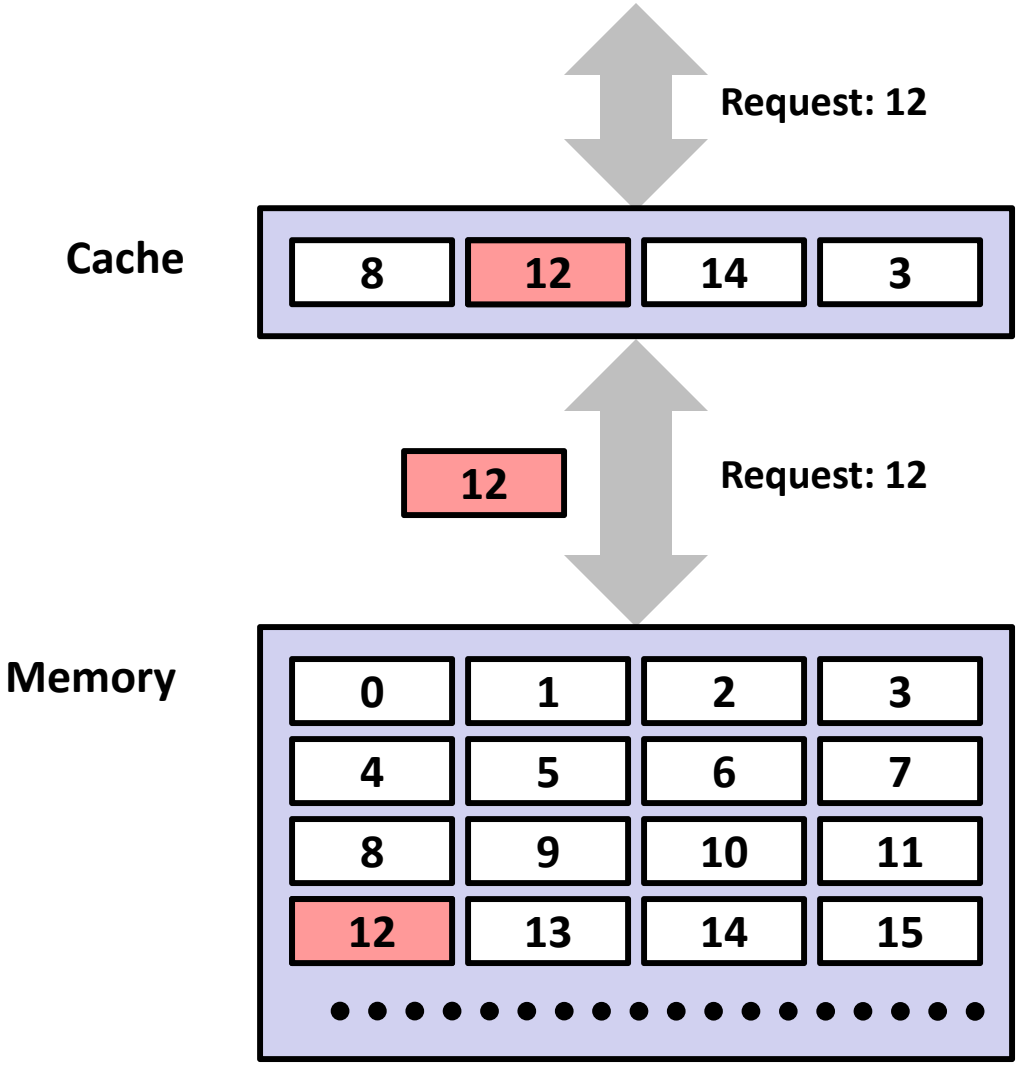
General Cache Concepts: Hit



Data in block b is needed

Block b is in cache:
Hit!

General Cache Concepts: Miss



Data in block b is needed

Block b is not in cache:
Miss!

Block b is fetched from memory

Block b is stored in cache

- Placement policy: determines where b goes
- Replacement policy: determines which block gets evicted (victim)

Cache Performance Metrics

- Miss Rate
  - Fraction of memory references not found in cache (misses/references)
- 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!)

Lets think about those numbers

- Huge difference between a hit and a miss
  - Could be 100x, if just L1 and main memory
- Would you believe 99% hits is twice as good as 97%?
  - Consider:
    cache hit time of 1 cycle
    miss penalty of 100 cycles
  - Average access time:
  - 97% hits: 1 cycle + 0.03 * 100 cycles = 4 cycles
  - 99% hits: 1 cycle + 0.01 * 100 cycles = 2 cycles
- This is why “miss rate” is used instead of “hit rate”

Why Caches Work

- 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.

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

Locality Example:

- **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**

Example: Locality?

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

- Data:
  - Temporal: sum referenced in each iteration
  - Spatial: array a[] accessed in stride-1 pattern
- Instructions:
  - Temporal: cycle through loop repeatedly
  - Spatial: reference instructions in sequence
- Being able to assess the locality of code is a crucial skill for a programmer

Locality Example #1

```
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;
}
```

Locality Example #2

```
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;
}
```

Locality Example #3

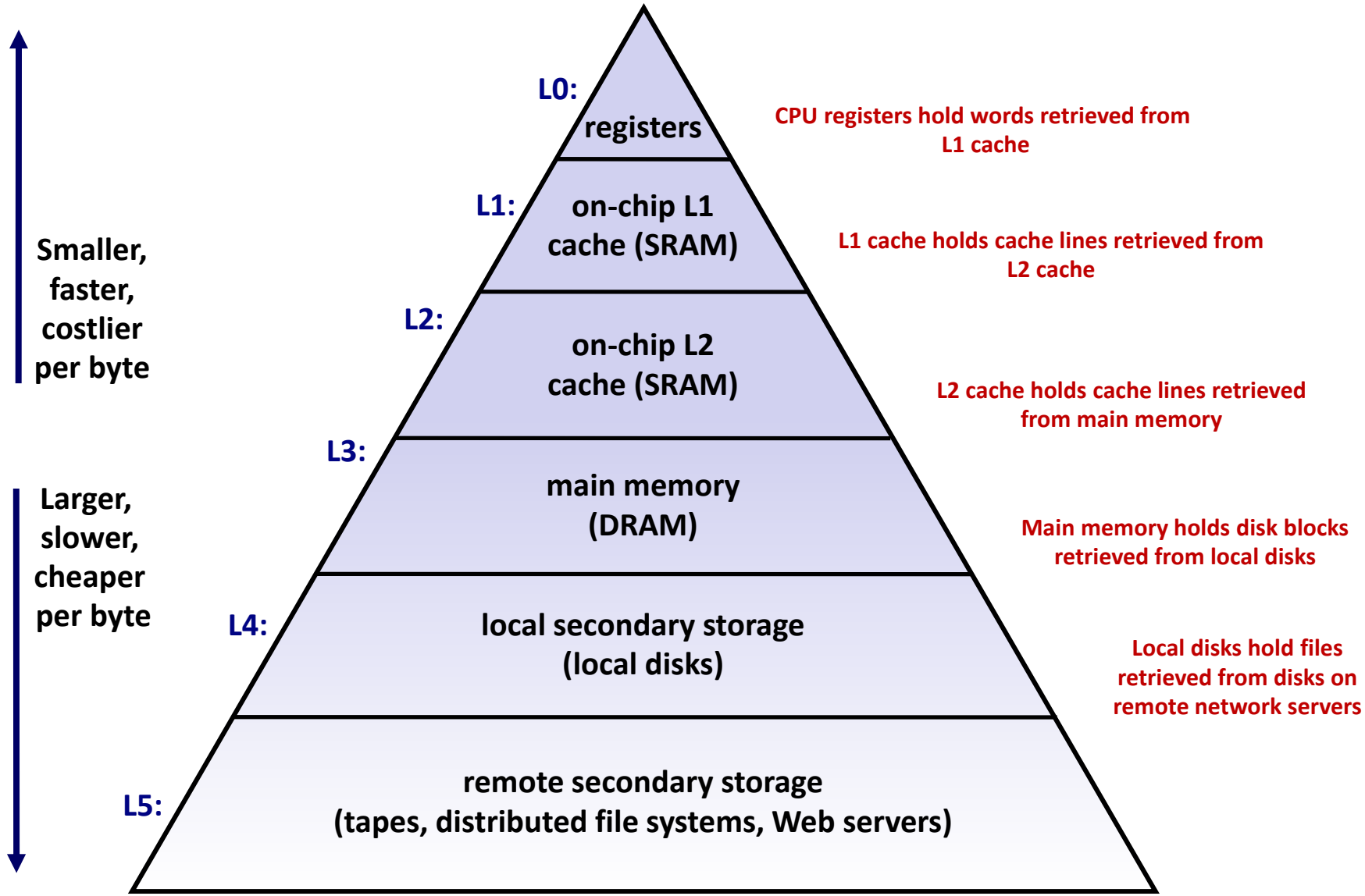
```
int sum_array_3d(int a[M][N][N])
{
    int i, j, k, sum = 0;

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

- How can it be fixed?

Memory Hierarchies

- Some fundamental and enduring properties of hardware and software:
  - Fast storage technologies cost more per byte and have less capacity.
  - The gaps between memory technology speeds are widening
    - True of registers ↔ DRAM, DRAM ↔ disk, etc.
  - Well-written programs tend to exhibit good locality.
- These fundamental properties complement each other beautifully.
- They suggest an approach for organizing memory and storage systems known as a memory hierarchy.

An Example Memory Hierarchy



Memory Hierarchies

- 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.
  - Net effect: A large pool of memory that costs as much as the cheap storage near the bottom, but that serves data to programs at the rate of the fast storage near the top.

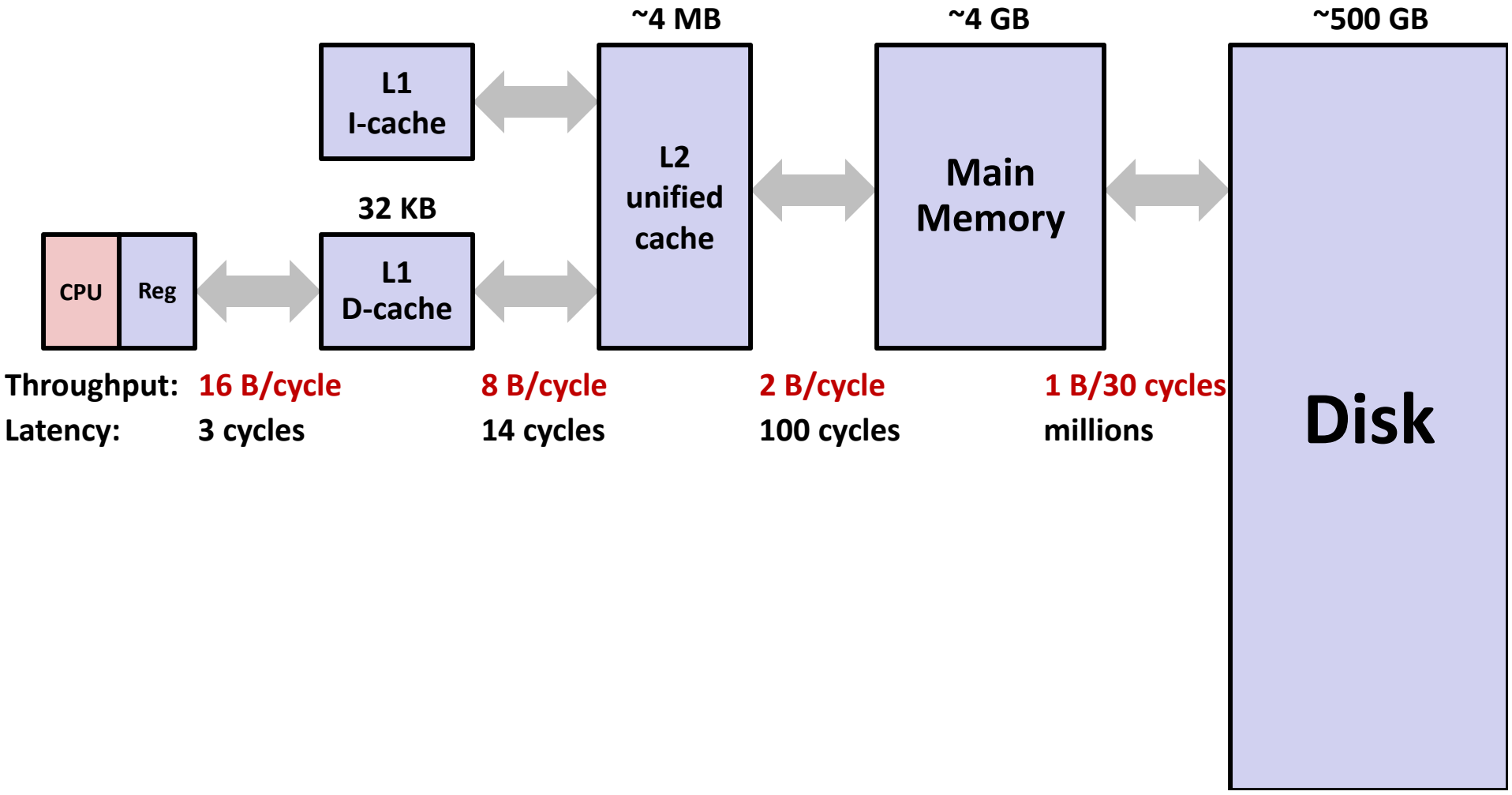
Examples of Caching in the Hierarchy

Cache Type	What is Cached?	Where is it Cached?	Latency (cycles)	Managed By
Registers	4-byte words	CPU core	0	Compiler
TLB	Address translations	On-Chip TLB	0	Hardware
L1 cache	64-bytes block	On-Chip L1	1	Hardware
L2 cache	64-bytes block	Off-Chip L2	10	Hardware
Virtual Memory	4-KB page	Main memory	100	Hardware+OS
Buffer cache	Parts of files	Main memory	100	OS
Network buffer cache	Parts of files	Local disk	10,000,000	AFS/NFS client
Browser cache	Web pages	Local disk	10,000,000	Web browser
Web cache	Web pages	Remote server disks	1,000,000,000	Web proxy server

Memory Hierarchy: Core 2 Duo

L1/L2 cache: 64 B blocks

Not drawn to scale



Writing Cache Friendly Code

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

```
int sumarrayrows(int a[M][N])
{
    int i, j, sum = 0;

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

Miss rate = 1/4 = 25%

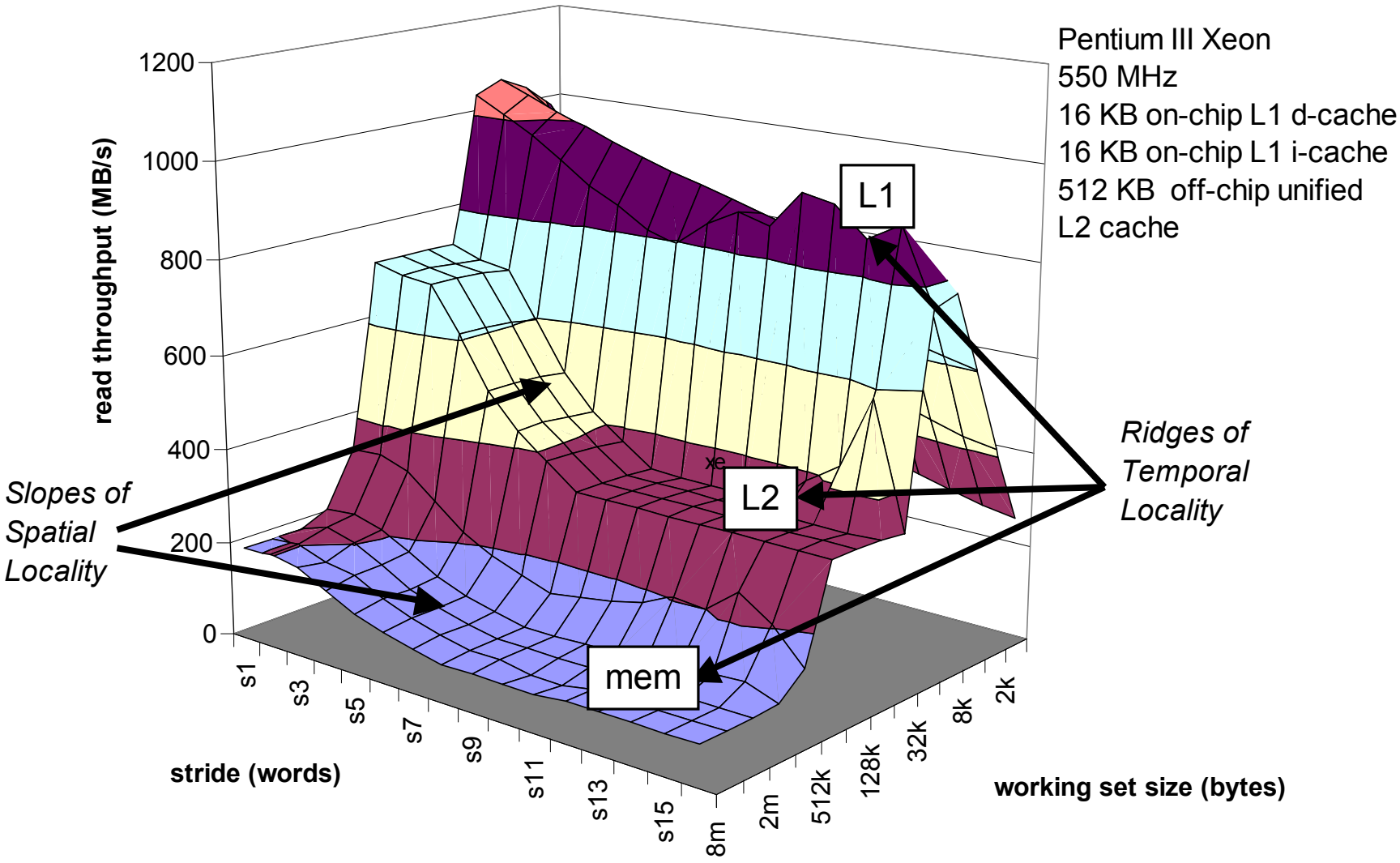
```
int sumarraycols(int a[M][N])
{
    int i, j, sum = 0;

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

Miss rate = 100%

The Memory Mountain

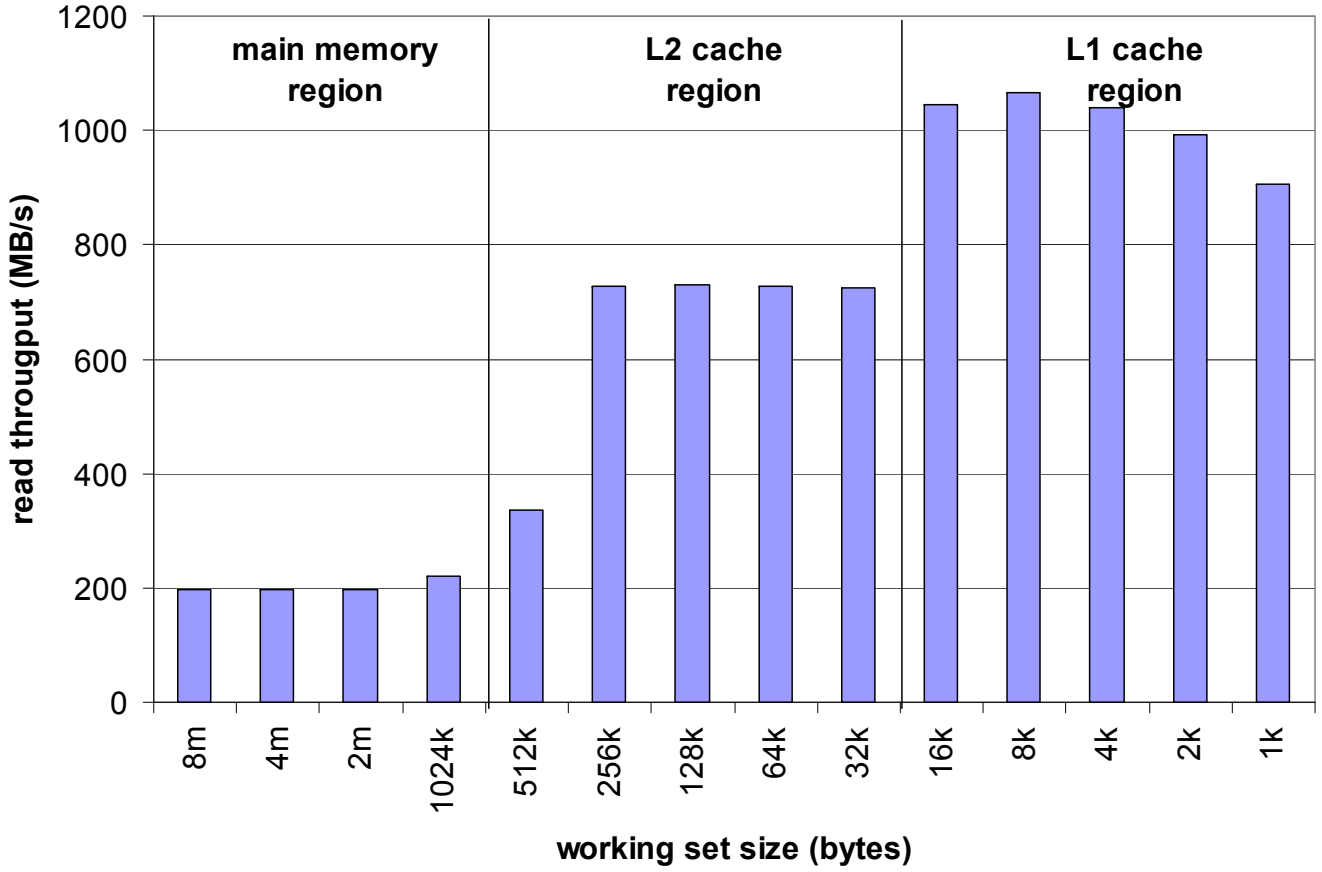
- Read throughput (read bandwidth)
  - Number of bytes read from memory per second (MB/s)
- Memory mountain
  - Measured read throughput as a function of spatial and temporal locality.
  - Compact way to characterize memory system performance.

The Memory Mountain



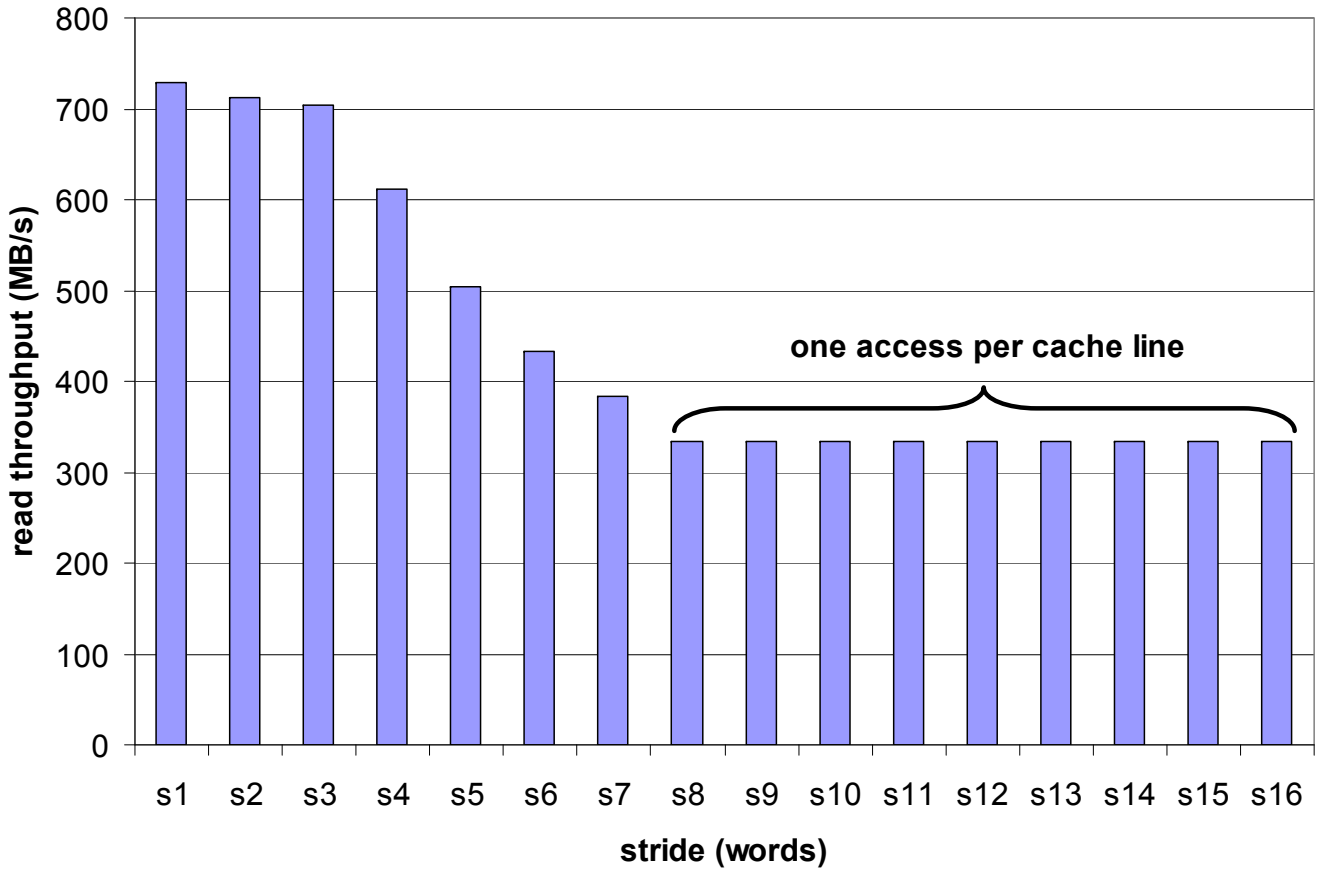
Ridges of Temporal Locality

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

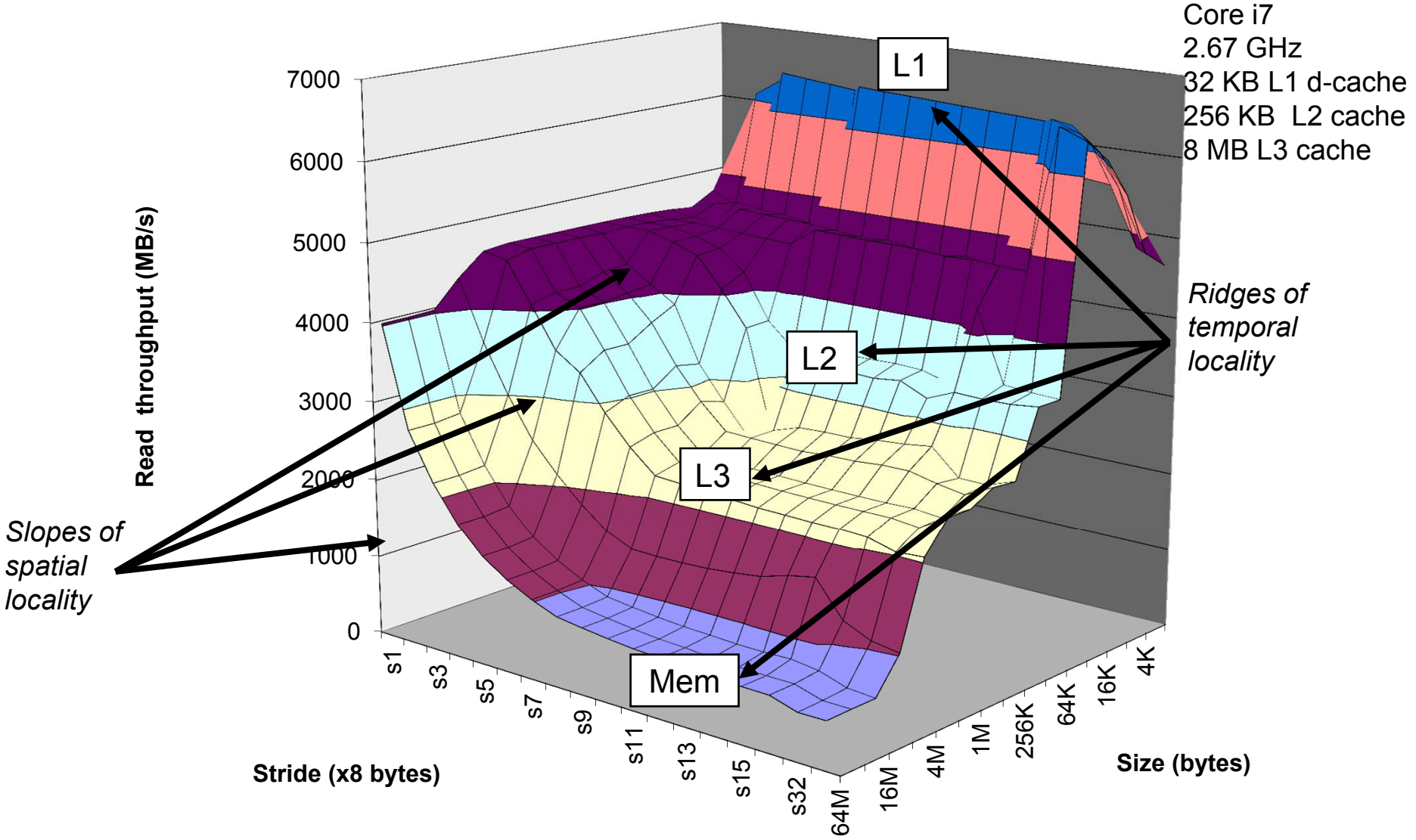


A Slope of Spatial Locality

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



The Memory Mountain (Core i7)



Matrix Multiplication Example

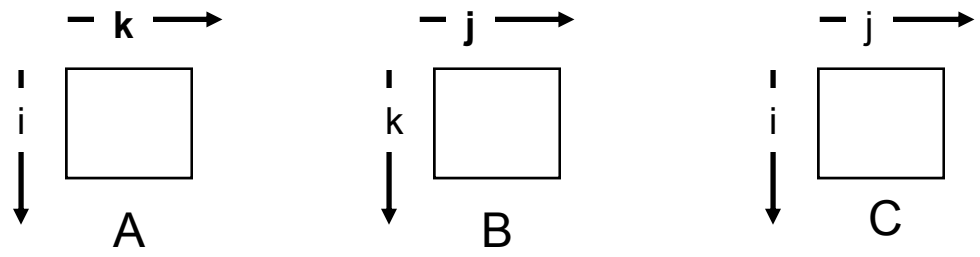
- Major Cache Effects to Consider
  - Total cache size
    - Exploit temporal locality and keep the working set small (e.g., by using blocking)
  - Block size
    - Exploit spatial locality

```
/* ijk */                          Variable sum
for (i=0; i<n; i++)  {             held in register
  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;
  }
}
```

- 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

Miss Rate Analysis for Matrix Multiply

- Assume:
  - Line size = 32B (big enough for 4 64-bit words)
  - Matrix dimension (N) is very large
    - Approximate 1/N as 0.0
  - Cache is not even big enough to hold multiple rows
- Analysis Method:
  - Look at access pattern of inner loop



Layout of C Arrays in Memory (review)

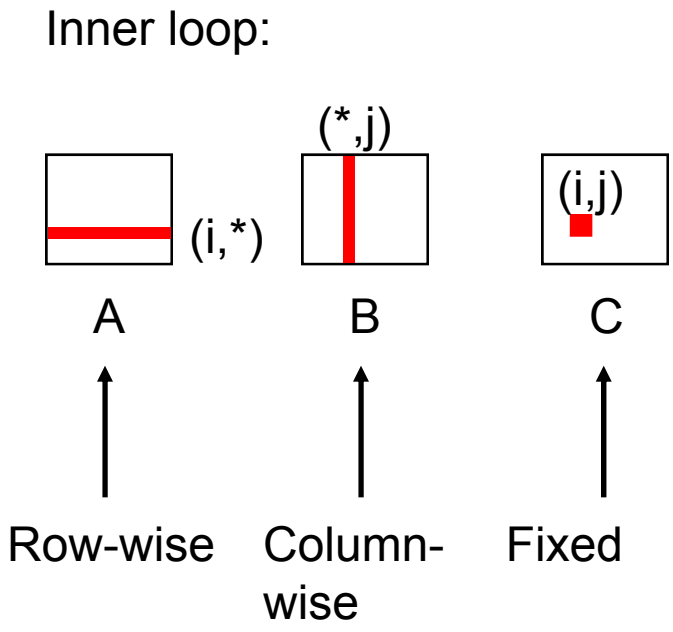
- C arrays allocated in row-major order
  - each row in contiguous memory locations
- Stepping through columns in one row:
  - `for (i = 0; i < N; i++)`

    `sum += a[0][i];`
  - accesses successive elements
  - if block size (B) > 4 bytes, exploit spatial locality
    - compulsory miss rate = 4 bytes / B
- Stepping through rows in one column:
  - `for (i = 0; i < N; i++)`

    `sum += a[i][0];`
  - accesses distant elements
  - no spatial locality!
    - compulsory miss rate = 1 (i.e. 100%)

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;
  }
}
```

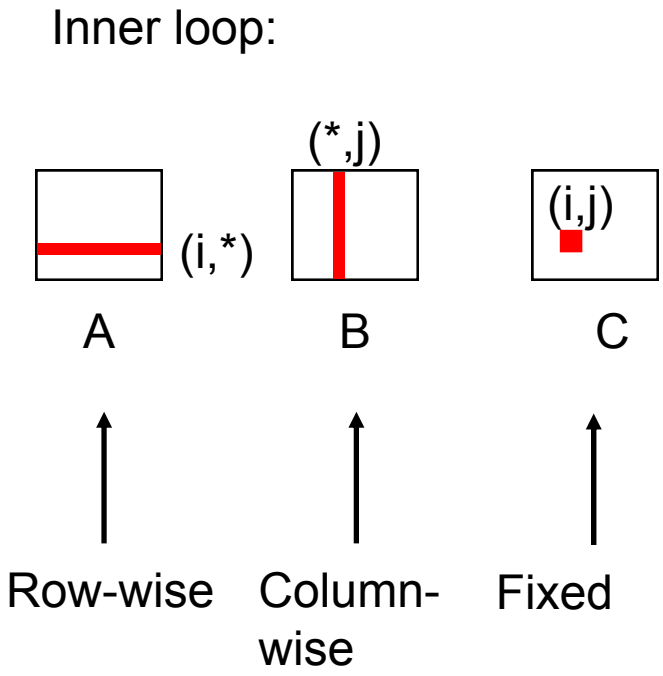


- Misses per Inner Loop Iteration:

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
  }
}
```

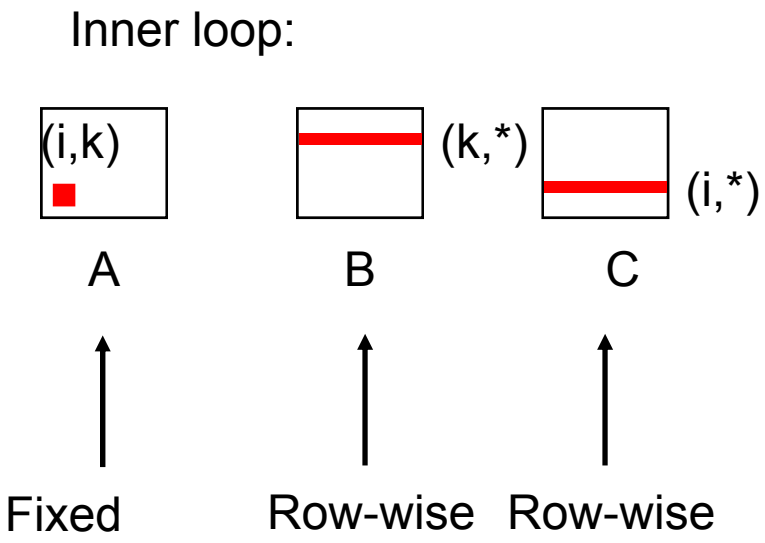


- Misses per Inner Loop Iteration:

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];
  }
}
```

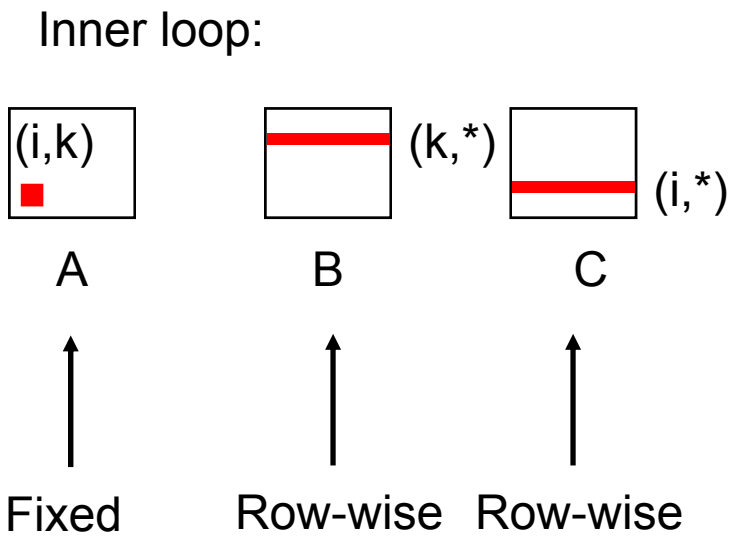


- Misses per Inner Loop Iteration:

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];
  }
}
```

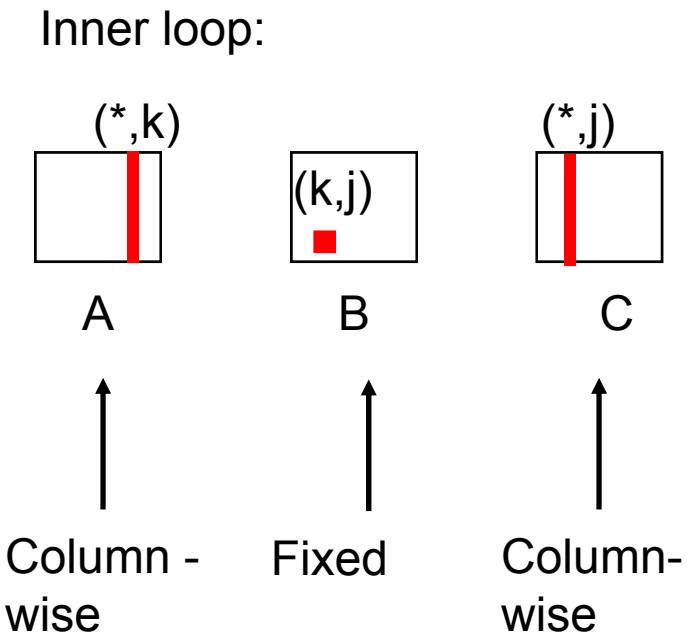


- Misses per Inner Loop Iteration:

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;
  }
}
```

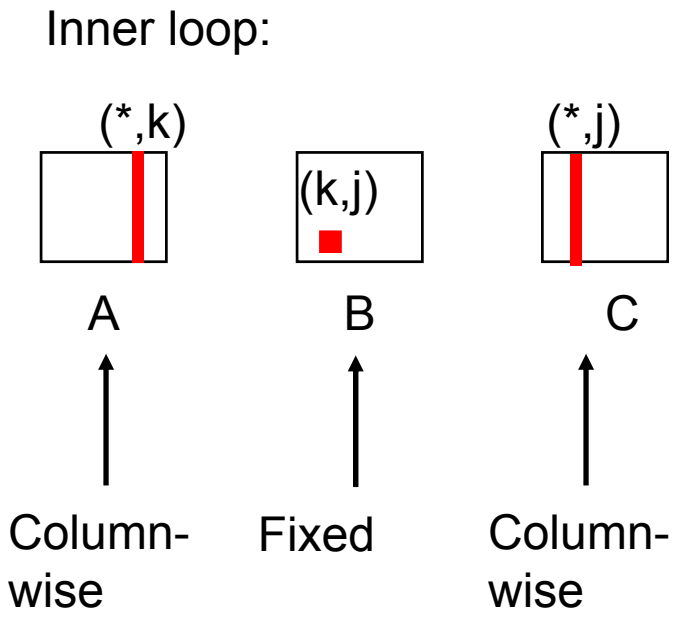


- Misses per Inner Loop Iteration:

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;
  }
}
```



- Misses per Inner Loop Iteration:

A	B	C
1.0	0.0	1.0

Summary of Matrix Multiplication

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

```
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;
   }
}
```

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

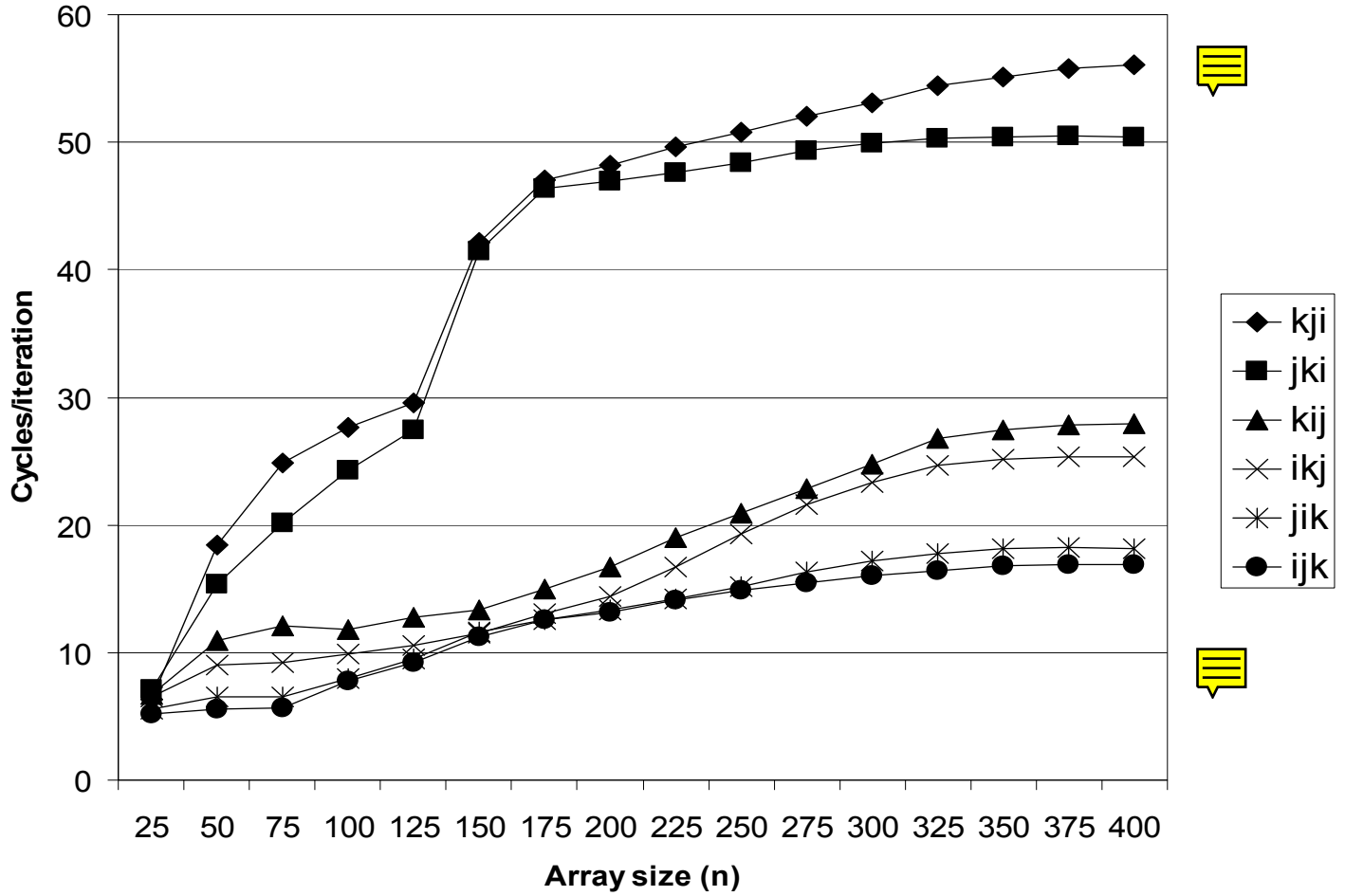
```
for (k=0; k<n; k++) {
   for (i=0; i<n; i++) {
      r = a[i][k];
      for (j=0; j<n; j++)
         c[i][j] += r * b[k][j];
   }
}
```

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

```
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;
   }
}
```

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
  - "block" (in this context) does not mean "cache block".
  - Instead, it mean a sub-block within the matrix.
  - Example: N = 8; sub-block size = 4

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

Key idea: Sub-blocks (i.e., $\mathbf{A_{xy}}$) 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)

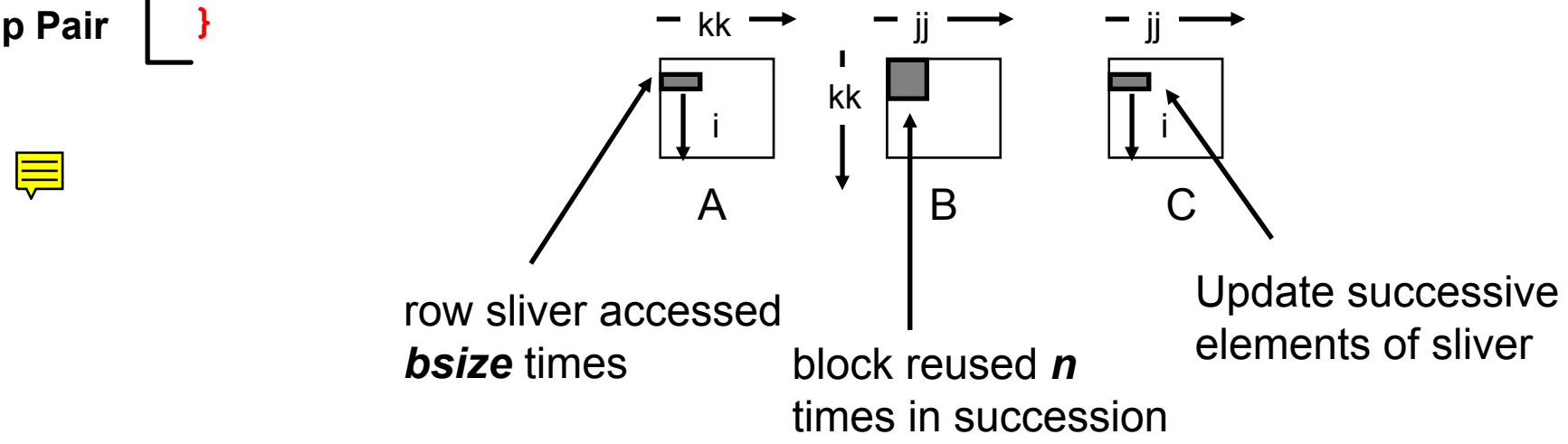
```
for (jj=0; jj<n; jj+=bsize) {
  for (i=0; i<n; i++)
    for (j=jj; j < min(jj+bsize,n); j++)
      c[i][j] = 0.0;
  for (kk=0; kk<n; kk+=bsize) {
    for (i=0; i<n; i++) {
      for (j=jj; j < min(jj+bsize,n); j++) {
        sum = 0.0
        for (k=kk; k < min(kk+bsize,n); k++) {
          sum +=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

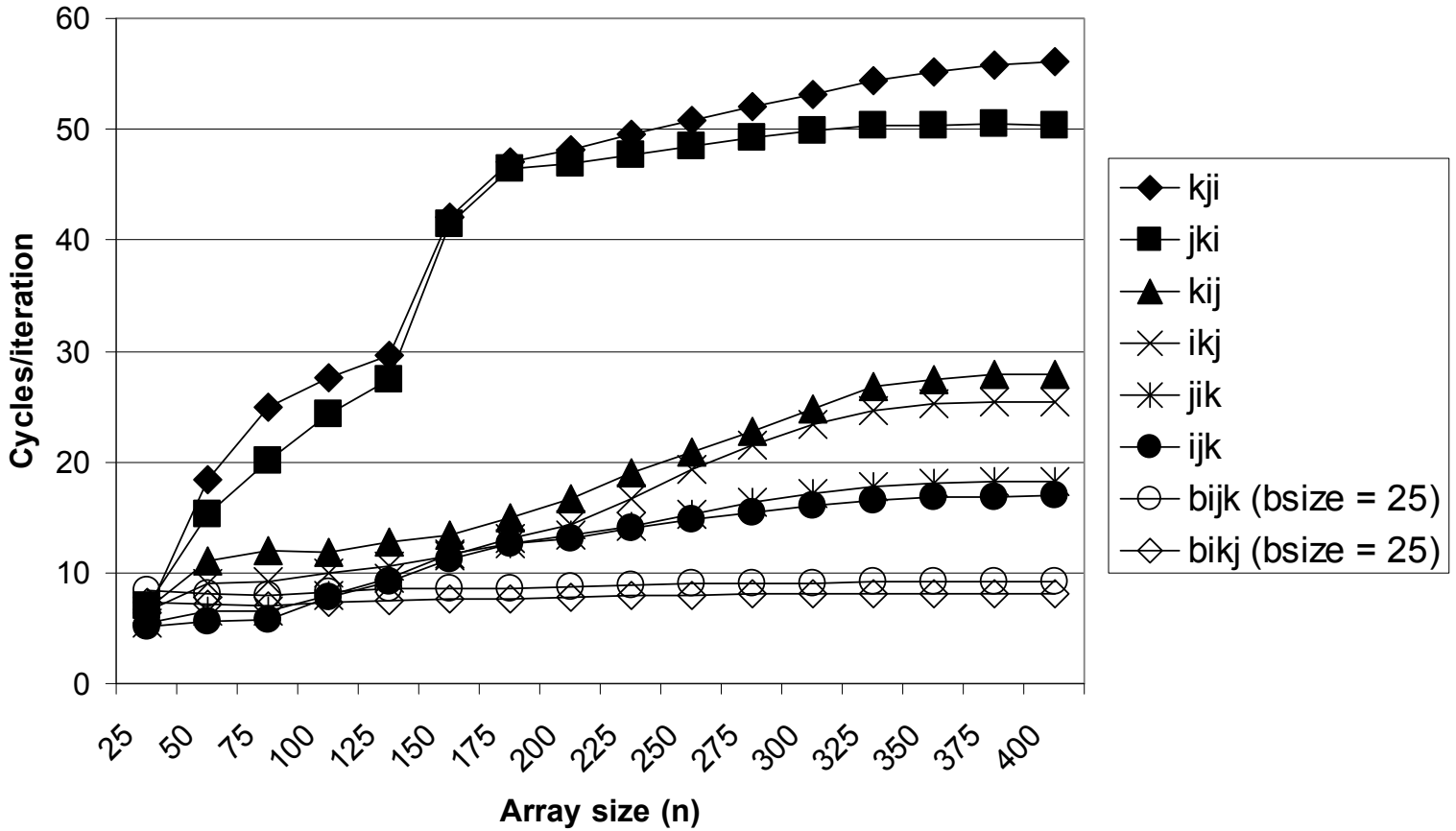
```
for (i=0; i<n; i++) {
    for (j=jj; j < min(jj+bsize,n); j++) {
      sum = 0.0
      for (k=kk; k < min(kk+bsize,n); k++) {
        sum += a[i][k] * b[k][j];
      }
      c[i][j] += sum;
    }
```

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)