CS/ECE 250
Computer Architecture

Caches and Memory Hierarchies

Benjamin Lee

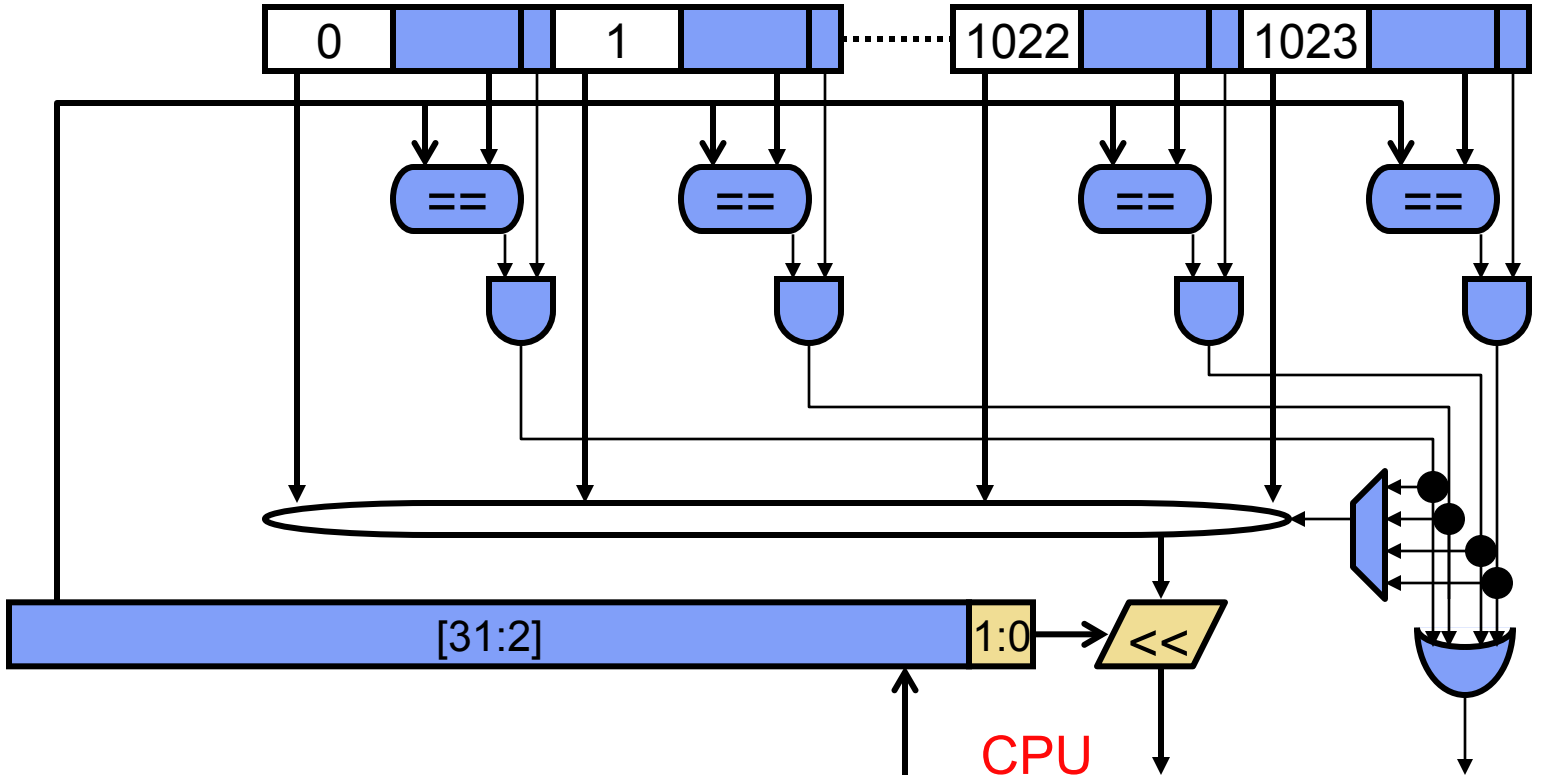
Duke University

Some slides derived from work by
Amir Roth (Penn), Alvin Lebeck (Duke), Dan Sorin (Duke)

Administrivia

- HW #4
  - Sunday, March 30 at 11:55pm
- Midterm #2
  - Tuesday, Apr 1 in class
  - Covers logic design, datapath and control (HW #3 & #4)
  - Does not include memory hierarchy. No assembly programming
- Two more homeworks
  - HW #5 Friday, Apr 11 – Memory hierarchy
  - HW #6 Wednesday, Apr 23 – Exceptions, I/O, Pipelining
- Reading: Chapter 5 in Patterson and Hennessy

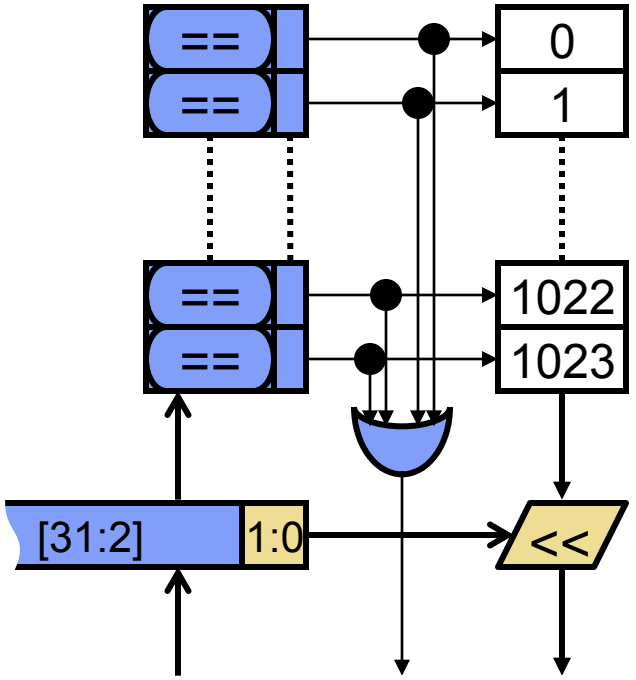
Full-Associativity (1024 Entries)



- How to implement full (or at least high) associativity?
  - Doing it this way is terribly inefficient
  - 1K matches are unavoidable, but 1K data reads + 1K-to-1 mux?

Full-Associativity with CAMs

- **CAM**: content addressable memory
  - Array of words with built-in comparators
  - Matchlines instead of bitlines
  - Output is "one-hot" (unary) encoding of match
- Fully-associative cache?
  - Tags as CAM
  - Data as RAM



Analyzing Cache Misses: 3C Model

- Divide cache misses into three categories
  - **Compulsory**: miss because cache has not previously seen address
    - Easy to identify
  - **Capacity**: miss caused because cache is too small
    - N is the number of blocks in the cache
    - Consecutive accesses to a block are separated by at least N other distinct blocks
  - **Conflict**: miss caused because cache associativity is too low
    - All other misses

ABCs of Caches

- **Associativity** (increase)
  - \+ Decreases conflict misses
  - – Increases t_{hit}
- **Block size** (increase)
  - – Increases conflict misses
  - \+ Decreases compulsory misses
  - ± Increases or decreases capacity misses
  - Negligible effect on t_{hit}
- **Capacity** (increase)
  - \+ Decreases capacity misses
  - – Increases t_{hit}

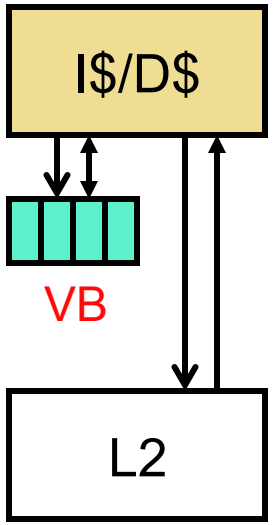
Two Possible Optimizations

- **Victim buffer**: for conflict misses
- **Prefetching**: for capacity/compulsory misses

Victim Buffer

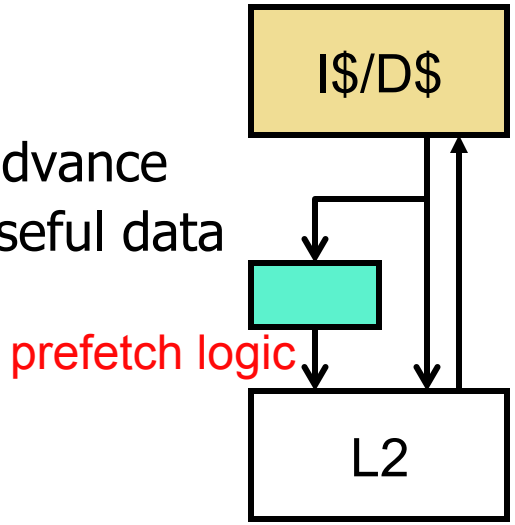
- Conflict misses: insufficient associativity
  - High-associativity is expensive, but also rarely needed
    - E.g., 3 blocks mapped to 2-way set and accessed sequentially

- **Victim buffer (VB)**: small FA cache (e.g., 4 entries)
  - Sits on I$/D$ fill path
  - VB is small → very fast
  - Blocks kicked out of I$/D$ placed in VB
  - On miss, check VB. If VB hits, return block to I$/D$
  - 4 extra ways, shared among all sets
    - + Only a few sets will need it at any given time
    - + Very effective in practice



Prefetching

- **Prefetching**: put blocks in cache proactively/speculatively
  - Anticipate upcoming miss addresses accurately
    - Prediction in software or hardware
  - Simple example: next block prefetching
    - Miss on address X → anticipate miss on X+blocksize
    - Works for instructions: sequential execution
    - Works for data: arrays
  - **Timeliness**: initiate prefetches sufficiently in advance
  - **Accuracy**: prefetch useful data, do not evict useful data



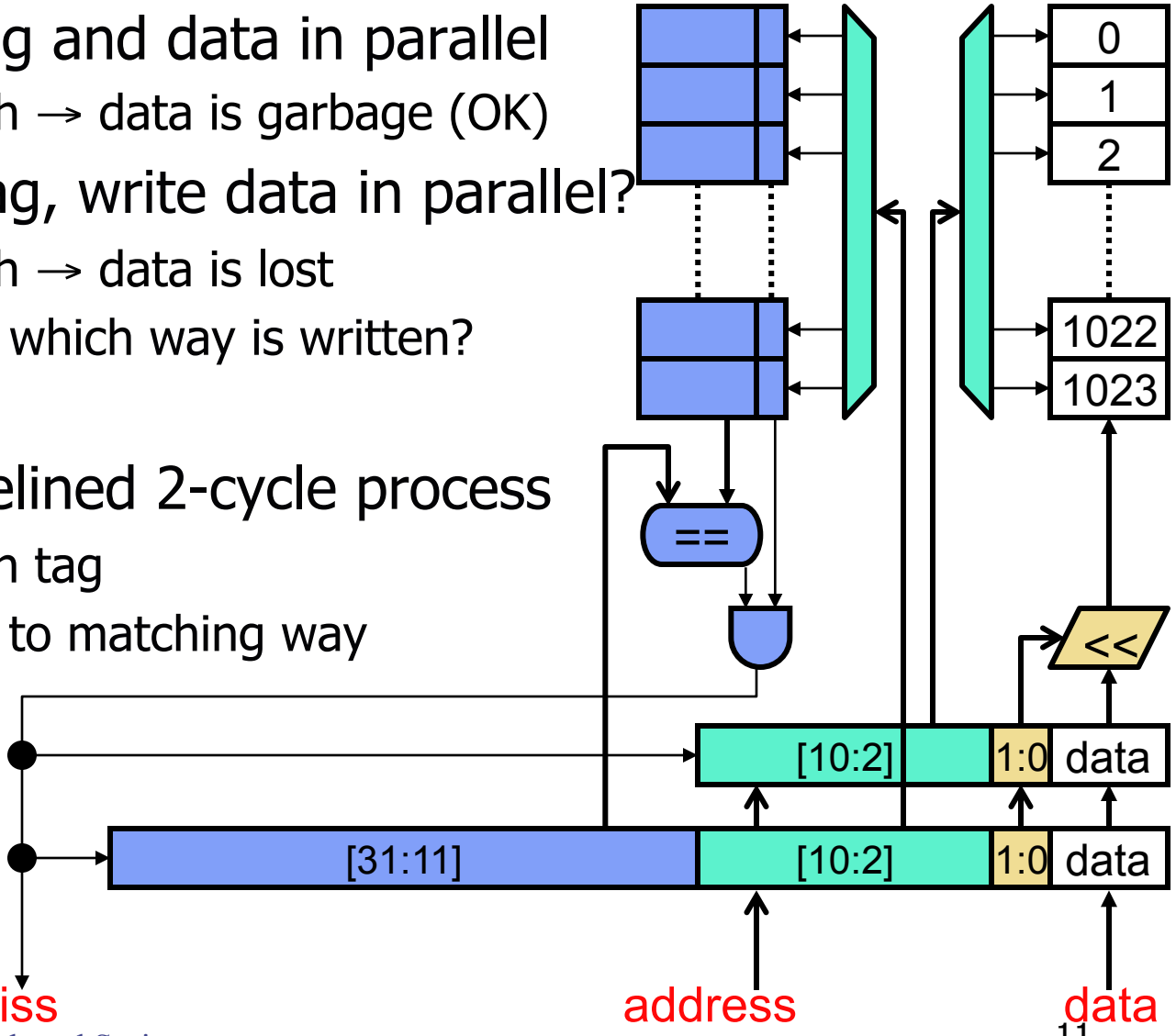
Cache Writes

- So far we have looked at reading from cache (loads)
- What about writing into cache (stores)?

- Several new issues arise during cache writes
  - Tag/data access
  - Write-through vs. write-back
  - Write-allocate vs. write-not-allocate

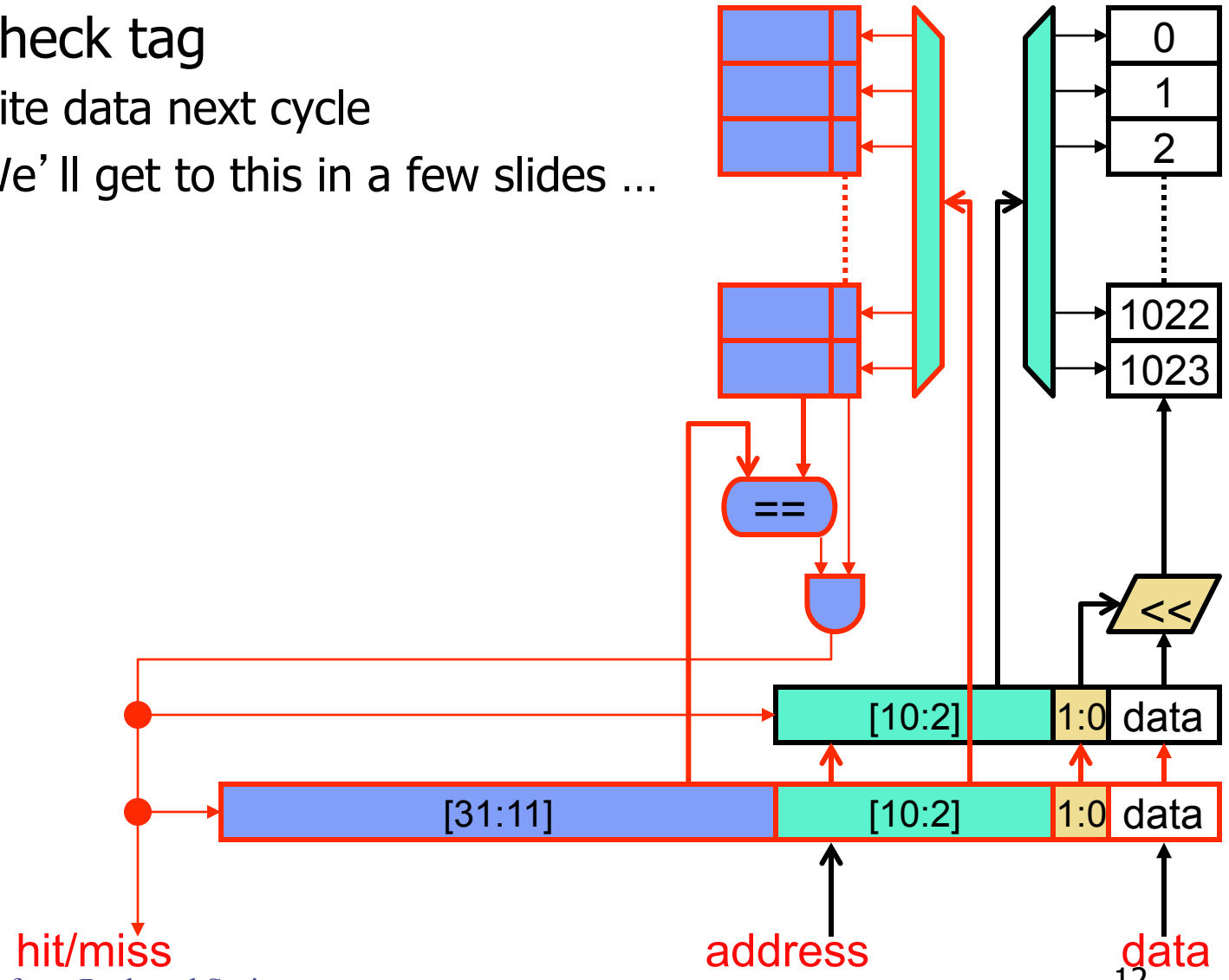
Tag/Data Access

- Reads: read tag and data in parallel
  - Tag mis-match → data is garbage (OK)
- Writes: read tag, write data in parallel?
  - Tag mis-match → data is lost
  - For SA cache, which way is written?
- Writes are pipelined 2-cycle process
  - Cycle 1: match tag
  - Cycle 2: write to matching way



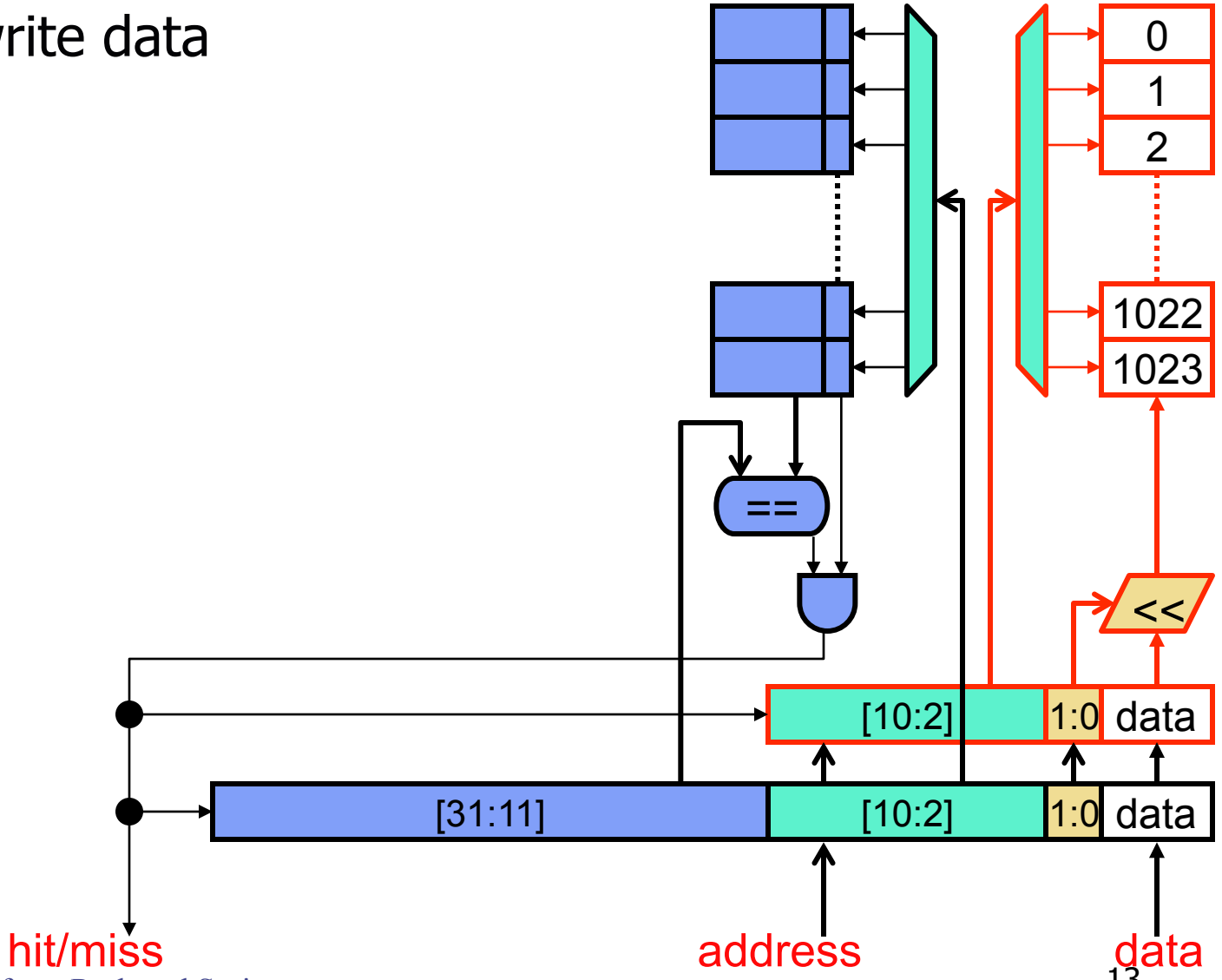
Tag/Data Access

- Cycle 1: check tag
  - Hit? Write data next cycle
  - Miss? We'll get to this in a few slides …



Tag/Data Access

- Cycle 2: write data



Write-Through vs. Write-Back

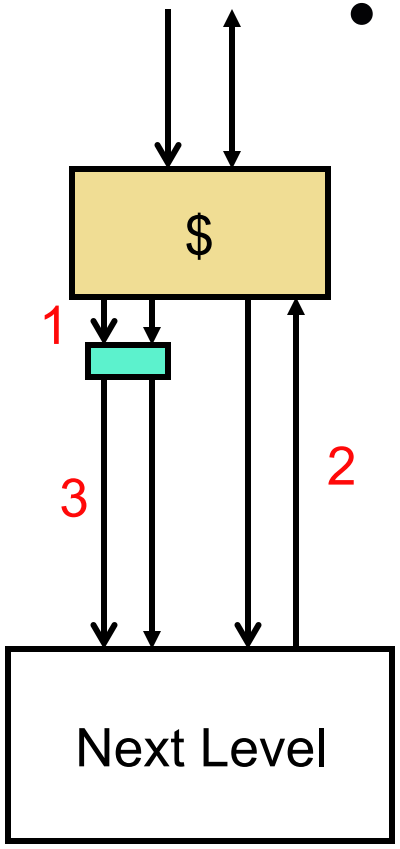
- When to propagate new value to (lower level) memory?
  - **Write-through**: immediately
    + Conceptually simpler
    + Uniform read miss latency
    – Requires additional bus bandwidth

  - **Write-back**: when block is evicted and replaced
    - Requires additional "dirty" bit per block
    + Minimal bus bandwidth
      - Only write back dirty blocks
    – Non-uniform read miss latency
      - Clean miss: one transaction (fill)
      - Dirty miss: two transactions (writeback & fill)

Write-allocate vs. Write-non-allocate

- What to do on a write miss?
  - **Write-allocate**: read block from lower level, write value into it
    + Decreases read misses
    – Requires additional bandwidth
    - Use with write-back
  - **Write-non-allocate**: just write to next level
    – Potentially more read misses
    + Uses less bandwidth
    - Use with write-through

Write Buffer



- **Write buffer**: between cache and memory
  - Write-through cache? Helps with store misses
    - + Write to buffer to avoid waiting for memory
      - Store misses become store hits
  - Write-back cache? Helps with dirty misses
    - + Allows you to do read first
      1. Write dirty block to buffer
      2. Read new block from memory to cache
      3. Write buffer contents to memory

Typical Processor Cache Hierarchy

- First level caches: optimized for t_{hit} and parallel access
  - Insns and data in separate caches (**I$**, **D$**)
  - Capacity: 8–64KB, block size: 16–64B, associativity: 1–4
  - Other: write-through or write-back
  - t_{hit}: 1–4 cycles
- Second level cache (**L2**): optimized for $\%_{miss}$
  - Insns and data in one cache for better utilization
  - Capacity: 128KB–1MB, block size: 64–256B, associativity: 4–16
  - Other: write-back
  - t_{hit}: 10–20 cycles
- Third level caches (**L3**): also optimized for $\%_{miss}$
  - Capacity: 1–8MB
  - t_{hit}: 30 cycles

Performance Calculation Example

- Parameters
  - Reference (address) stream: 20% stores, 80% loads
  - L1 D$: t_{hit} = 1ns, $\%_{miss}$ = 5%, write-through + write-buffer
  - L2: t_{hit} = 10ns, $\%_{miss}$ = 20%, write-back, 50% dirty blocks
  - Main memory: t_{hit} = 50ns, $\%_{miss}$ = 0%

- What is $t_{avgL1D\$}$ without an L2?
  - Write-through and write-buffer means all stores hit
  - $t_{missL1D\$} = t_{hitM}$
  - $t_{avgL1D\$} = t_{hitL1D\$} + \%_{loads}*\%_{missL1D\$}*t_{hitM}$ = 1ns+(0.8*0.05*50ns) = 3ns

- What is $t_{avgD\$}$ with an L2?
  - Write-back means dirty misses incur double cost (writeback, fill)
  - $t_{missL1D\$} = t_{avgL2}$
  - $t_{avgL2} = t_{hitL2}+(1+\%_{dirty})*\%_{missL2}*t_{hitM}$ = 10ns+(1.5*0.2*50ns) =25ns
  - $t_{avgL1D\$} = t_{hitL1D\$} + \%_{loads}*\%_{missL1D\$}*t_{avgL2}$ = 1ns+(0.8*0.05*25ns) =2ns

Cache Organization Summary

- Average access time of a memory component
  - $t_{avg} = t_{hit} + \%_{miss} * t_{miss}$
  - Hard to get low t_{hit} and $\%_{miss}$ in one structure → hierarchy
- Memory hierarchy
  - Cache (SRAM) → memory (DRAM) → swap (Disk)
  - Smaller, faster, more expensive → bigger, slower, cheaper
- SRAM
  - Analog technology for implementing big storage arrays
  - Cross-coupled inverters + bitlines + wordlines
  - Delay ~ √#bits * #ports

Summary, cont'd

- Cache ABCs
  - Capacity, associativity, block size
  - 3C miss model: compulsory, capacity, conflict
- Some optimizations
  - Victim buffer for conflict misses
  - Prefetching for capacity, compulsory misses
- Write issues
  - Pipelined tag/data access
  - Write-back vs. write-through/write-allocate vs. write-no-allocate
  - Write buffer

Next Your Programs and Caches

Cache Performance

Tave = number of cycles we stall waiting for memory operation

Execution time = (Core execution clock cycles + Memory stall clock cycles) x Clock cycle time

Memory stall clock cycles = Memory accesses x Miss rate x Miss penalty

Example

- Assume every instruction takes 1 cycle
- Miss penalty = 20 cycles
- Miss rate = 10%
- 1000 total instructions, 300 memory accesses
- Memory stall cycles? CPU clocks?

Cache Performance

- Memory Stall Cycles = 300 * 0.10 * 20 = 600
- Core Cycles = 1000 + 600 = 1600

- 60% slower because of cache misses!

- Change miss penalty to 100 cycles
- Core Cycles = 1000 + 3000 = 4000 cycles

Improving Cache Performance

1. Reduce the miss rate,

2. Reduce the miss penalty, or

3. Reduce the time to hit in the cache.

Reducing Misses (The 3 Cs)

- **Compulsory**—The first access to a block is not in the cache, so the block must be brought into the cache. These are also called cold start misses or first reference misses.

- **Capacity**—If the cache cannot contain all the blocks needed during execution of a program, capacity misses will occur due to blocks being discarded and later retrieved.

- **Conflict**—If the block-placement strategy is set-associative or direct mapped, conflict misses (in addition to compulsory, capacity misses) will occur because a block can be discarded and later retrieved if too many blocks map to its set. These are also called collision misses or interference misses.

Cache Performance

- Your program and caches
- Can you affect performance?
- Think about 3Cs

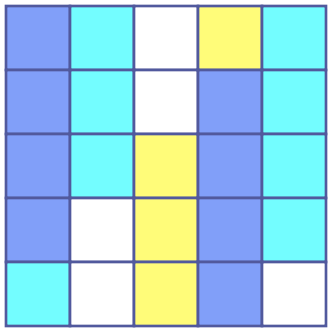
Mapping Arrays to Memory

Row-major

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

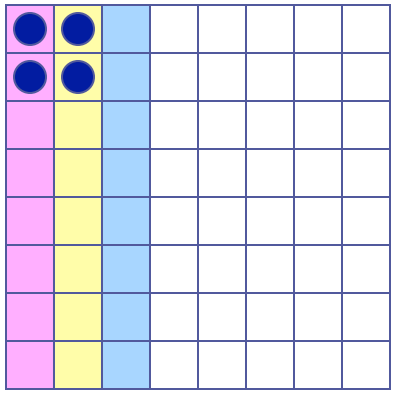
Column major

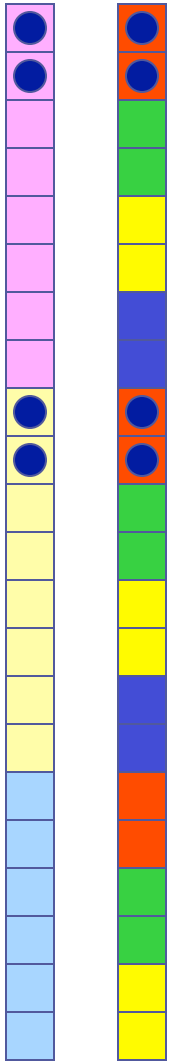
0	5	10	15	20
1	6	11	16	21
2	7	12	17	22
3	8	13	18	23
4	9	14	19	24



Part of the Row maps into cache

Array Mapping and Cache Behavior



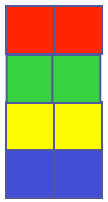


Memory

Cache Mapping

- Elements spread out in memory because of column-major mapping
- Fixed mapping into cache
- Self-interference in cache

Cache



Data Cache Performance

- Instruction Sequencing
  - **Loop Interchange**: change nesting of loops to access data in order stored in memory
  - **Loop Fusion**: Combine 2 independent loops that have same looping and some variables overlap
  - **Blocking**: Improve temporal locality by accessing "blocks" of data repeatedly vs. going down entire columns or rows

- Data Layout
  - **Merging Arrays**: Improve spatial locality by single array of compound elements vs. 2 separate arrays
  - **Nonlinear Array Layout**: Mapping 2 dimensional arrays to the linear address space
  - **Pointer-based Data Structures**: Node-allocation

Loop Interchange Example

```
/* Before */
for (k = 0; k < 100; k = k+1)
   for (j = 0; j < 100; j = j+1)
        for (i = 0; i < 5000; i = i+1)
              x[i][j] = 2 * x[i][j];
/* After */
for (k = 0; k < 100; k = k+1)
   for (i = 0; i < 5000; i = i+1)
        for (j = 0; j < 100; j = j+1)
              x[i][j] = 2 * x[i][j];
```

Matrix x stored in row-major format (i.e., row layout is sequential in memory).

Interchange produces sequential accesses instead of 100-word strides through memory

Loop Fusion Example

```
/* Before */
for (i = 0; i < N; i = i+1)
   for (j = 0; j < N; j = j+1)
       a[i][j] = 1/b[i][j] * c[i][j];
for (i = 0; i < N; i = i+1)
   for (j = 0; j < N; j = j+1)
       d[i][j] = a[i][j] + c[i][j];

/* After */
for (i = 0; i < N; i = i+1)
   for (j = 0; j < N; j = j+1)
   {   a[i][j] = 1/b[i][j] * c[i][j];
       d[i][j] = a[i][j] + c[i][j];}
```

Baseline incurs two misses when accessing matrices a and c.

Fusion incurs only one miss when accessing matrices a and c.

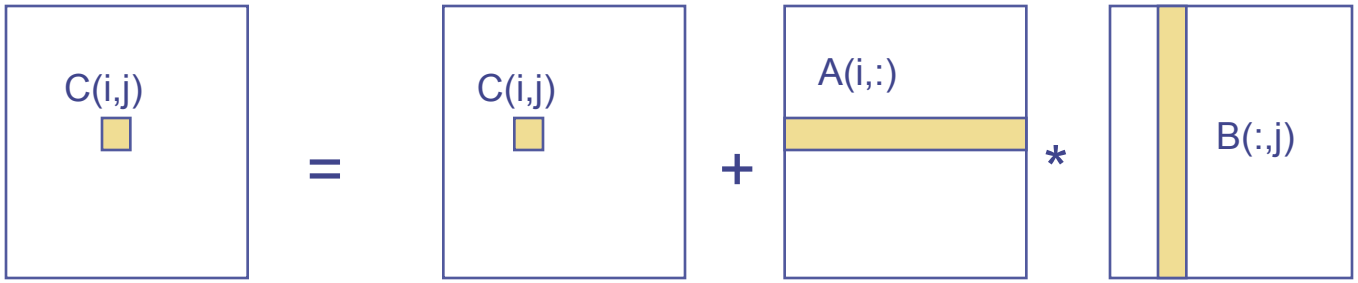
Naïve Matrix Multiply

```
/* Before */
 for(i = 0; i < n; i++)
    for (j = 0; j < n; j++)
      for (k = 0; k < n; k++)
        C[i][j] = C[i][j] + A[i][k]*B[k][j];
```

- Misses depend on N and cache size

Naïve Matrix Multiply

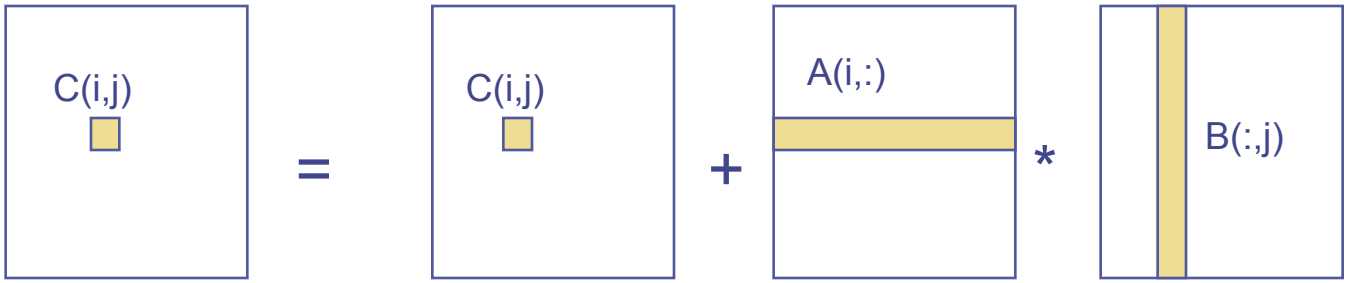
```
{implements C = C + A*B}
for i = 1 to n
  {read row i of A into fast memory}
   for j = 1 to n
       {read C(i,j) into fast memory}
       {read column j of B into fast memory}
       for k = 1 to n
           C(i,j) = C(i,j) + A(i,k) * B(k,j)
           {write C(i,j) back to slow memory}
```



Naïve Matrix Multiply

Number of slow memory references on unblocked matrix multiply

$m = n^3$	read each column of B n times
$+ n^2$	read each row of A once
$+ 2n^2$	read and write each element of C once
$= n^3 + 3n^2$	



Blocking (Tiling) Example

```
/* Before */
 for(i = 0; i < N; i++)
    for (j = 0; j < N; j++)
      for (k = 0; k < N; k++)
        c[i][j] = c[i][j] + a[i][k]*b[k][j];
```

- Two inner loops
  - Read all NxN elements of c[ ][ ]
  - Read N elements of rows in a[ ][ ], b[ ][ ] repeatedly
  - Write all NxN elements of c[ ][ ]
- Capacity misses depend on N and cache size
  - 3 NxN => no capacity misses; otherwise ...
- Idea is to compute on BxB submatrix that fits

Blocked (Tiled) Matrix Multiply

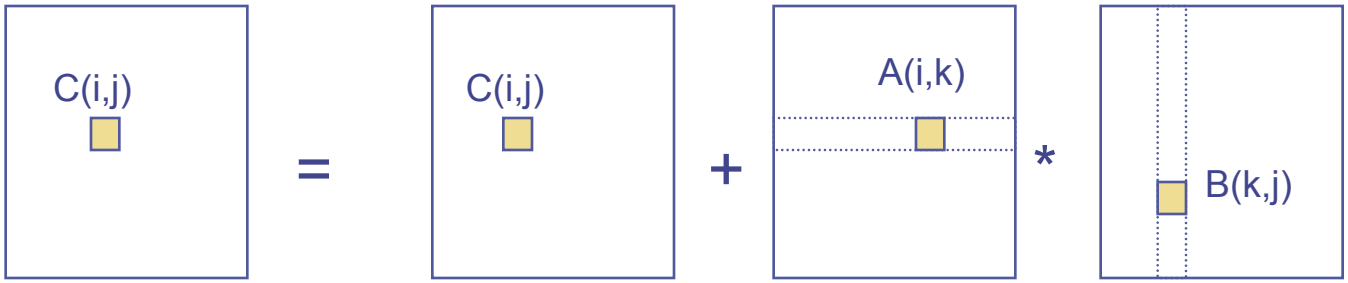
```
/* After */
 for(ii = 0; ii < n; ii += B)
    for (jj = 0; jj < n; jj += B)
      for (kk = 0; kk < n; kk +=B)

        for(i = ii; i < MIN(ii+B-1,n); i++)
          for (j = jj; j < MIN(jj+B-1,n); j++)
            for (k = kk; k < MIN(kk+B-1,n); k++)

              c[i][j] = c[i][j] + a[i][k]*b[k][j];
```

- B is called the blocking factor or tile size

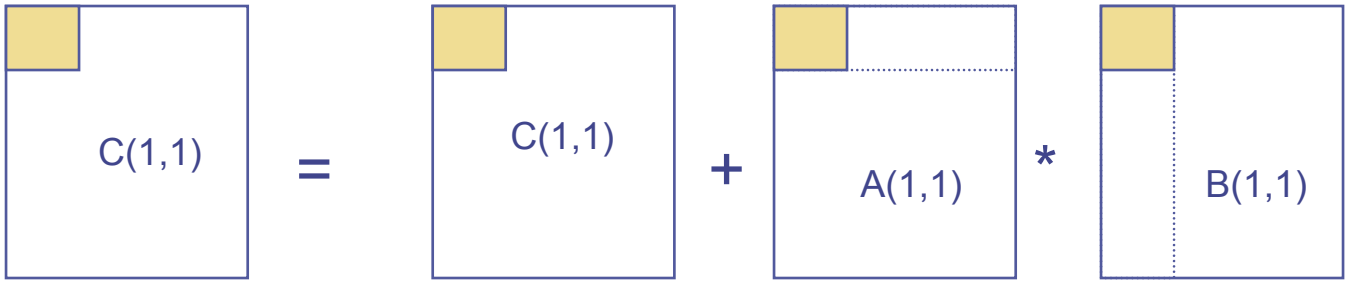
Blocked (Tiled) Matrix Multiply

Consider A,B,C to be N by N matrices of b by b sub-blocks
Where b =n / N is called the block size

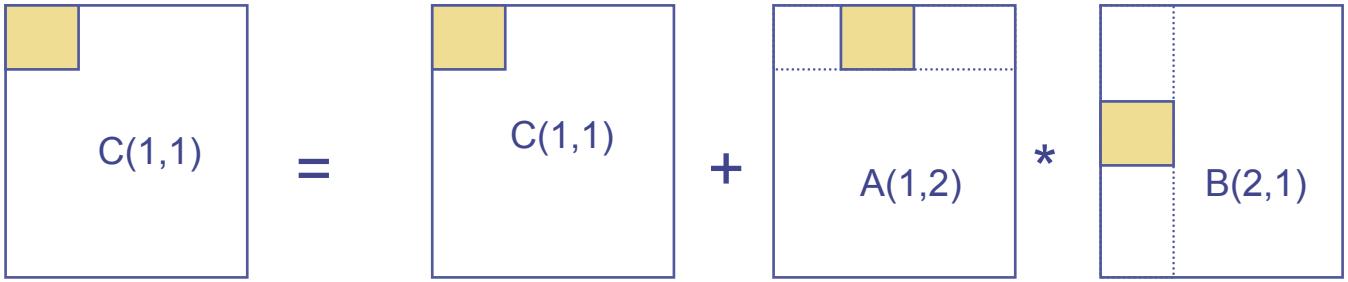
```
for i = 1 to N
  for j = 1 to N
    {read block C(i,j) into fast memory}
    for k = 1 to N
        {read block A(i,k) into fast memory}
        {read block B(k,j) into fast memory}
        C(i,j) = C(i,j) + A(i,k) * B(k,j) {do a matrix multiply on blocks}
    {write block C(i,j) back to slow memory}
```



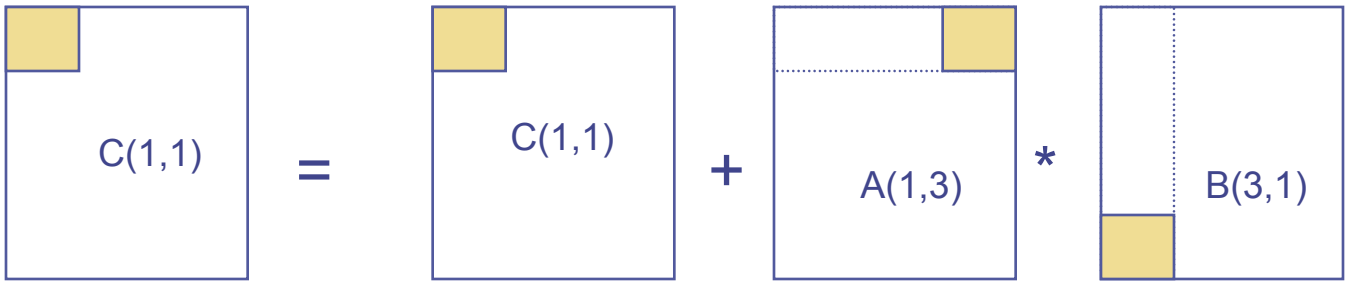
Blocked (Tiled) Matrix Multiply



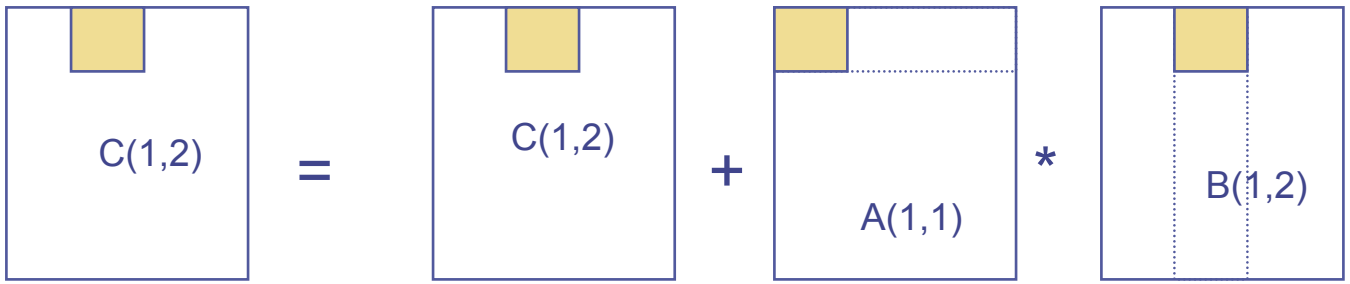
Blocked (Tiled) Matrix Multiply



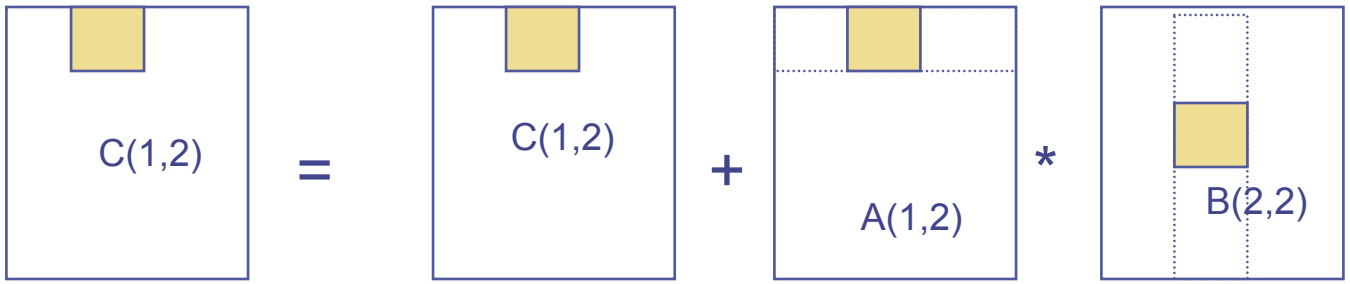
Blocked (Tiled) Matrix Multiply



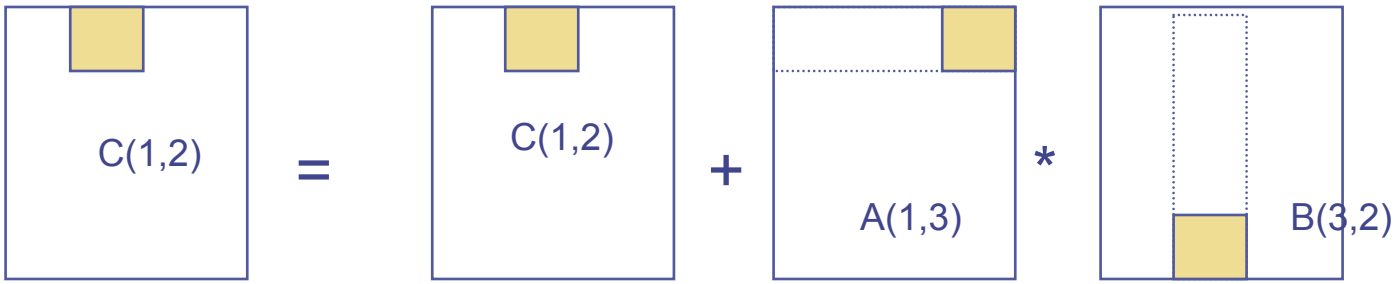
Blocked (Tiled) Matrix Multiply



Blocked (Tiled) Matrix Multiply



Blocked (Tiled) Matrix Multiply



Blocked (Tiled) Matrix Multiply

m is amount memory traffic between slow and fast memory

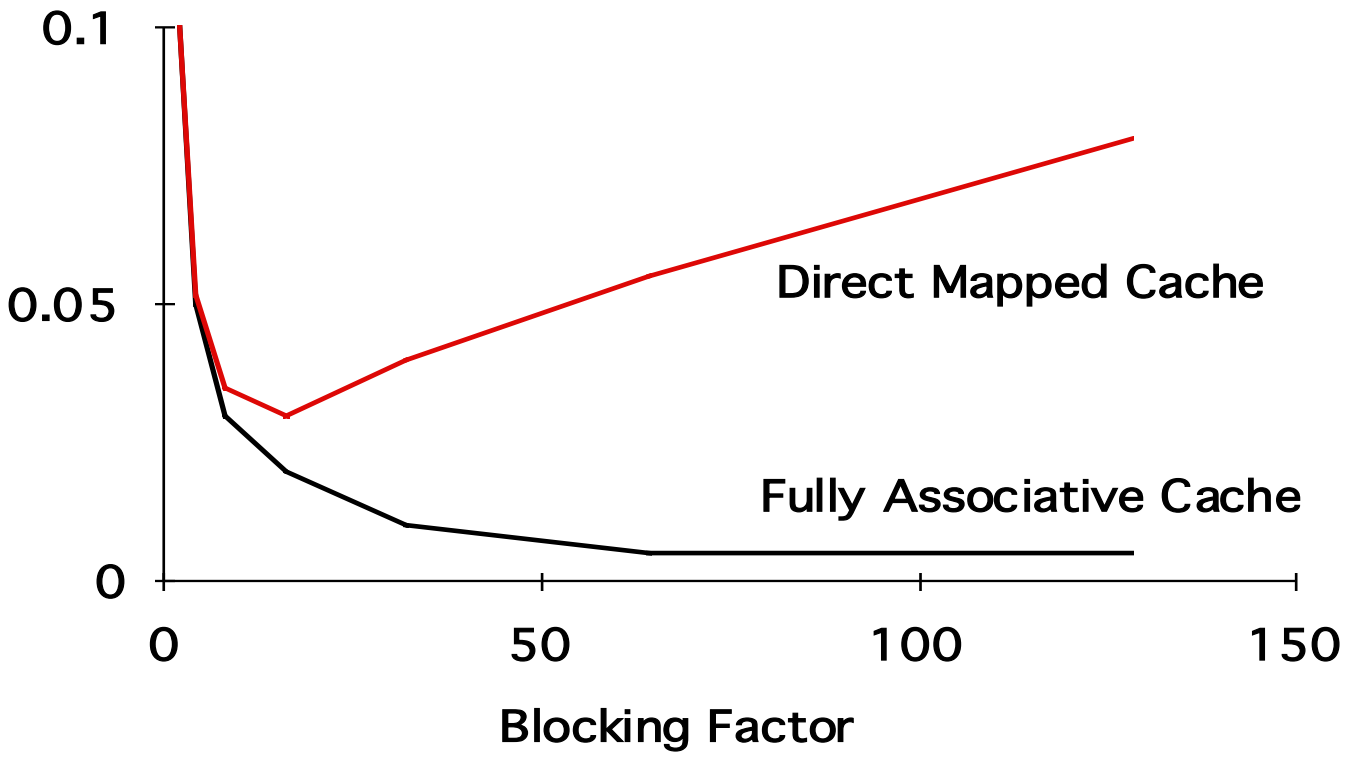
matrix has nxn elements, and NxN blocks each of size bxb

b = n / N

$m = N*n^2$	read each block of B N^3 times (N^3 * n/N * n/N)
$+ N*n^2$	read each block of A N^3 times
$+ 2n^2$	read and write each block of C once
$= (2N + 2) * n^2$	
$= 2(n/b + 1) * n^2$	
$= 2n^3 / b + 2n^2$	compare to naïve matrix multiply $n^3 + 3n^2$

So we can improve performance by increasing the blocksize b

Reducing Conflict Misses by Blocking



- Conflict misses in caches not FA vs. Blocking size
  - Lam et al [1991] a blocking factor of 24 had a fifth the misses vs. 48 despite both fit in cache

Data Layout Optimizations

- Changes in program control affect the order in which memory is accessed

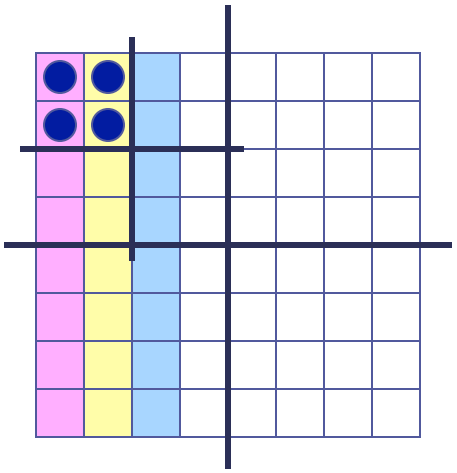
- Changes in data layout affect how data structures map to memory locations

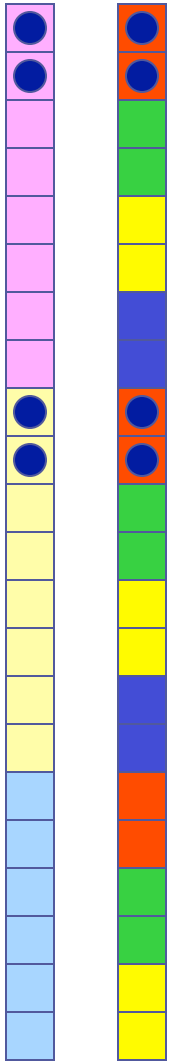
Merging Arrays Example

```
/* Before */
int val[SIZE];
int key[SIZE];

/* After */
struct merge {
   int val;
   int key;
};
struct merge merged_array[SIZE];
```

Reducing conflicts between val & key

Layout and Cache Behavior



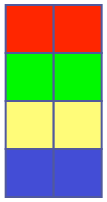


Memory

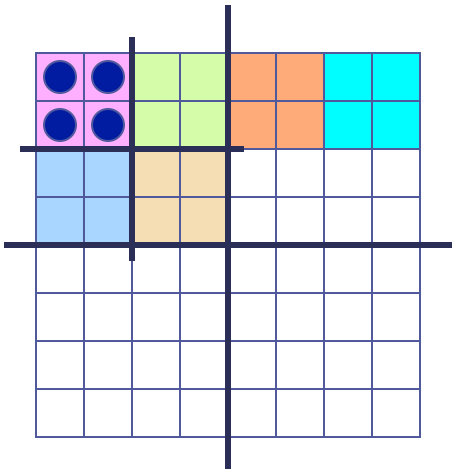
Cache Mapping

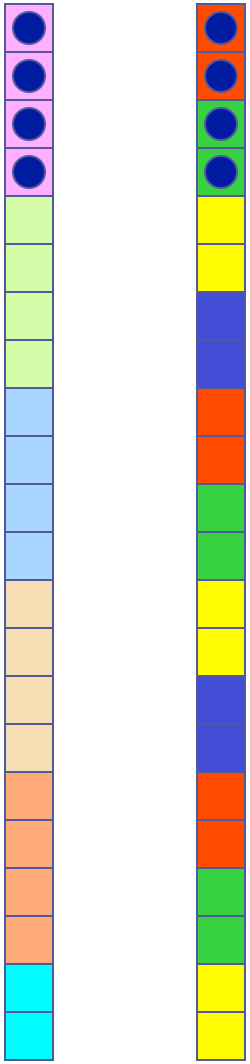
- Tile elements spread out in memory because of column-major mapping
- Fixed mapping into cache
- Self-interference in cache
- Each block holds two elements

Cache



Making Tiles Contiguous





Memory

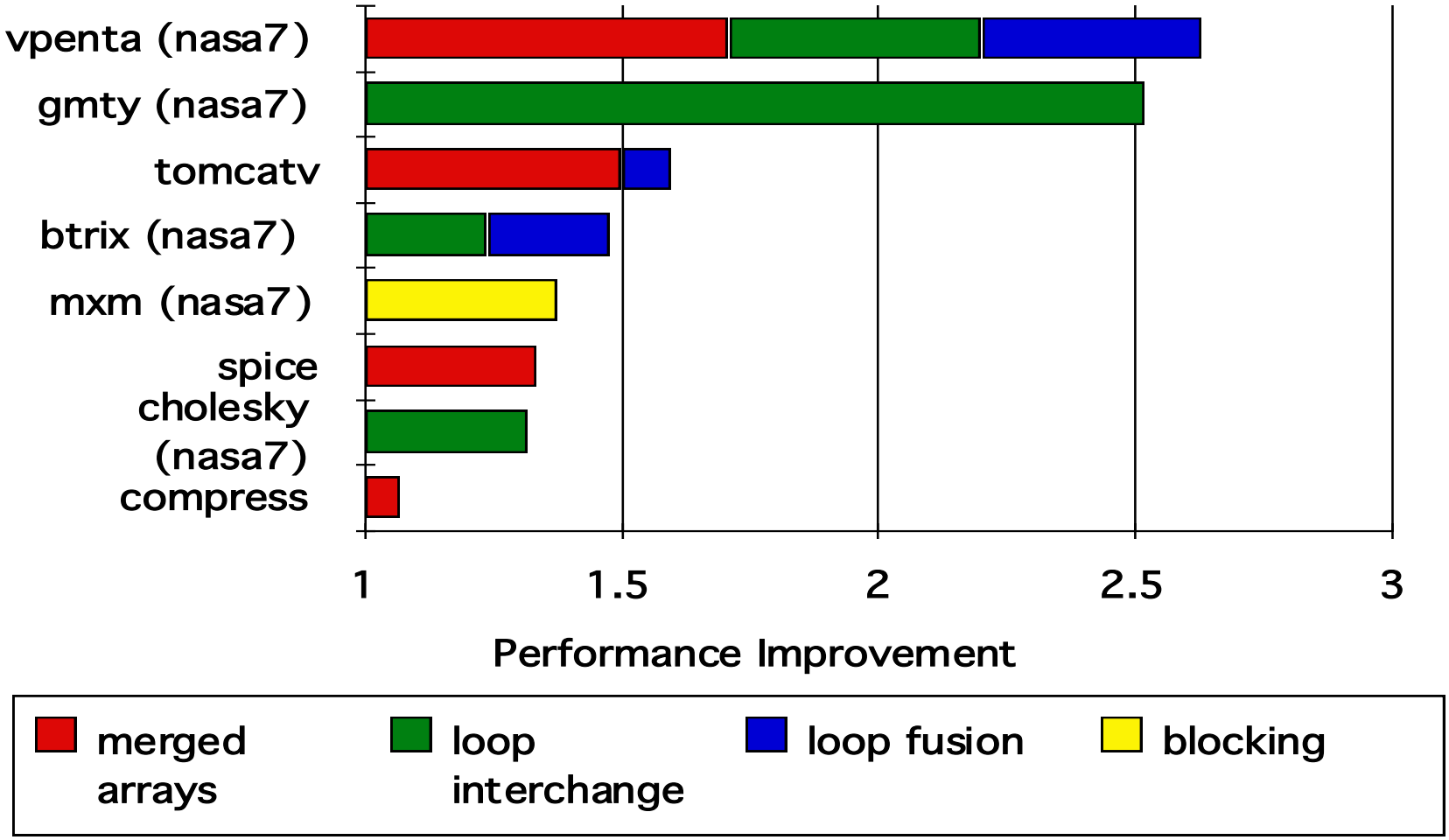
Cache Mapping

- Elements of a quadrant are contiguous
- Recursive layout
- Elements of a tile are contiguous
- No self-interference in cache

Pointer-based Data Structures

- Linked List, Binary Tree
- Group linked elements close together in memory
- Need relatively static traversal pattern
- Or could do it during garbage collection/compaction

Summary of Program Optimizations to Reduce Cache Misses



Reducing I-Cache Misses by Compiler Optimizations

- Instructions
  - Reorder procedures in memory to reduce misses
  - Profiling to look at conflicts
  - McFarling [1989] reduced caches misses by 75% on 8KB direct mapped cache with 4 byte blocks

Summary

- Cost-effective memory hierarchy
- Works by exploiting temporal and spatial locality
- Associativity, Blocksize, Capacity (ABCs of caches)
- Know how a cache works
  - Break address into tag,index, block offset
- Know how to draw a block diagram of a cache
- Know CPU cycles/time, Memory Stall Cycles
- Know programs and cache performance