ECE/CS 757: Advanced Computer Architecture II

Instructor:Mikko H Lipasti

Spring 2017

University of Wisconsin-Madison

Lecture notes based on slides created by John Shen, Mark Hill, David Wood, Guri Sohi, Jim Smith, Natalie Enright Jerger, Michel Dubois, Murali Annavaram, Per Stenström and probably others

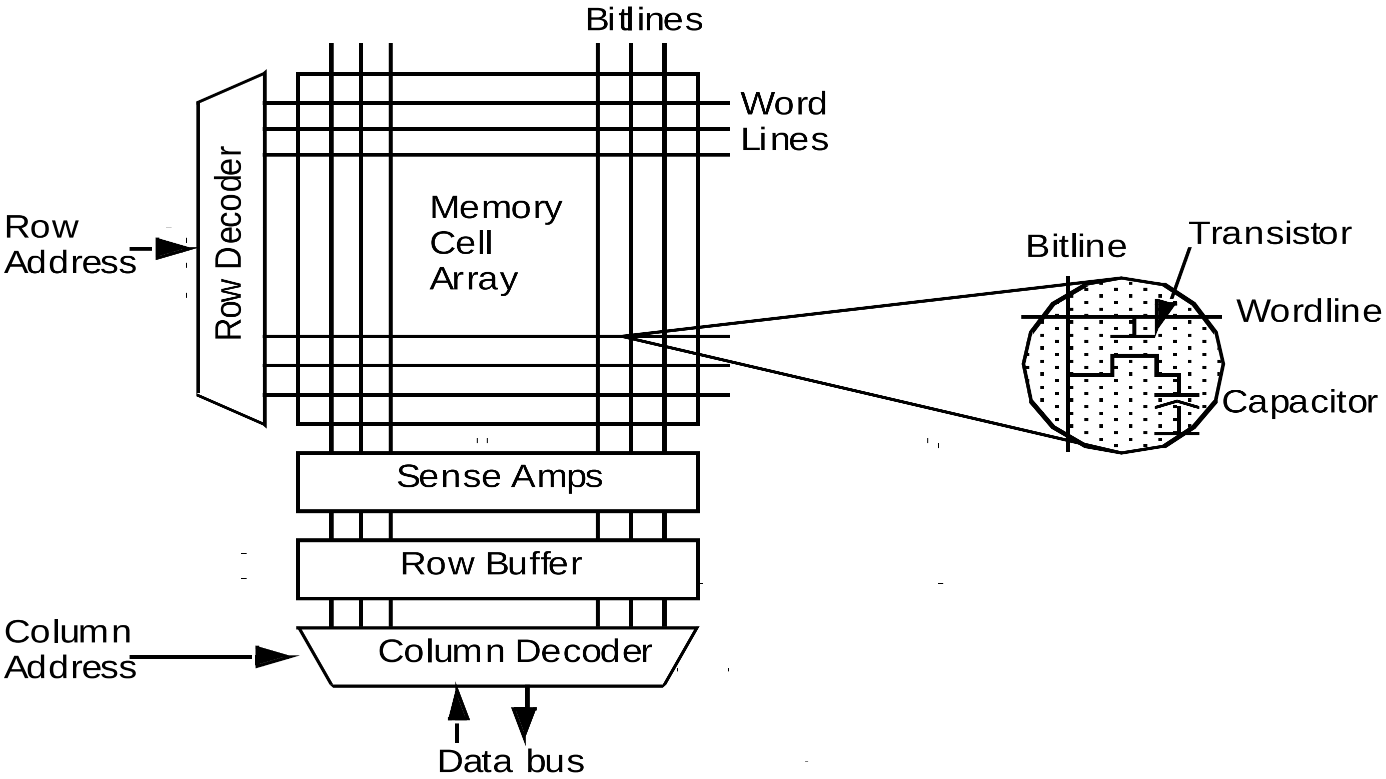
Lecture 5 Outline

- Main Memory and Cache Review
- Caches and Replacement Policies
- Cache Coherence
  - Coherence States
  - Snoopy bus-based Invalidate Protocols
  - Invalidate protocol optimizations
  - Update Protocols (Dragon/Firefly)
  - Directory protocols
  - Implementation issues

Main Memory

- DRAM chips
- Memory organization
  - Interleaving
  - Banking
- Memory controller design

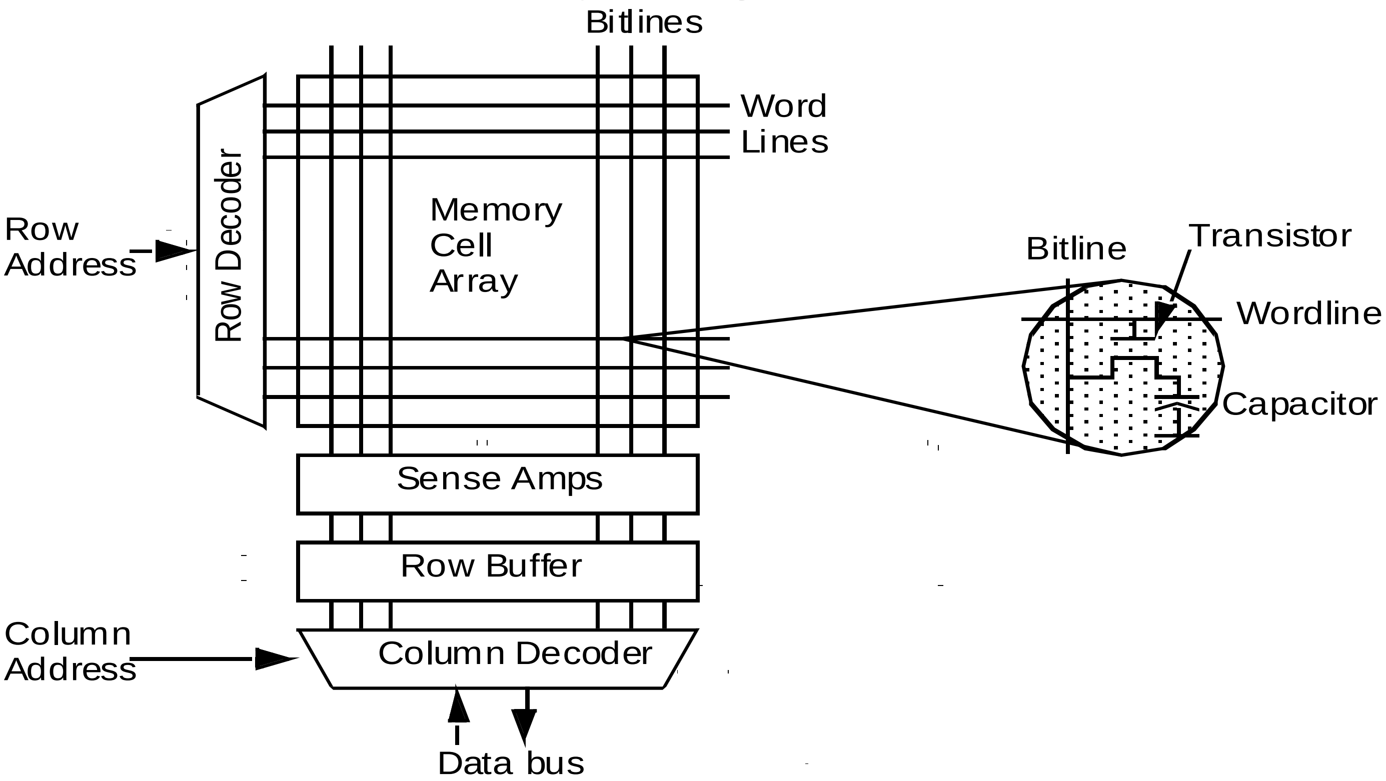
DRAM Chip Organization



- Optimized for density, not speed
- Data stored as charge in capacitor
- Discharge on reads => destructive reads
- Charge leaks over time
  - refresh every 64ms

- Cycle time roughly twice access time
- Need to precharge bitlines before access

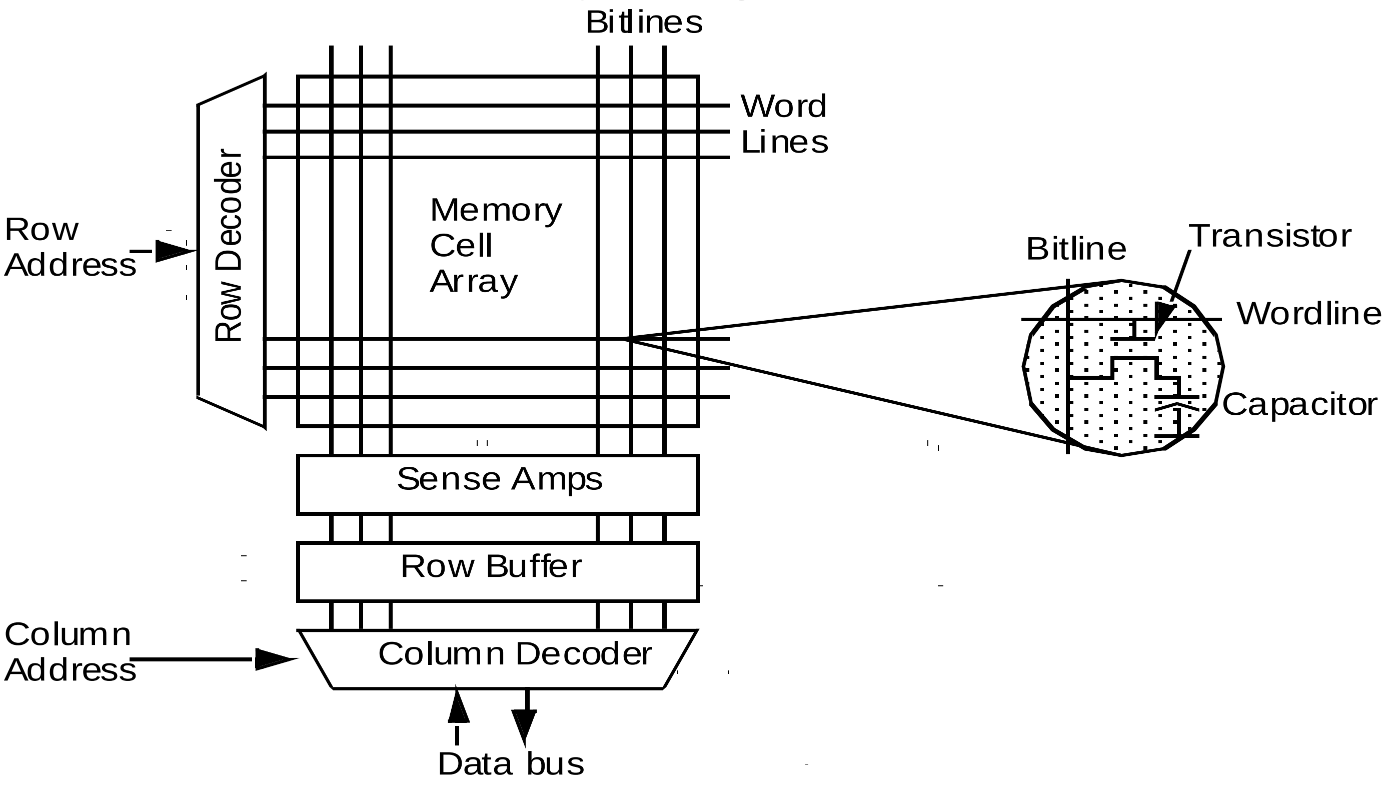
DRAM Chip Organization



- Current generation DRAM
  - 8Gbit @25nm
  - 266 MHz synchronous interface
  - Data clock 4x (1066MHz), double-data rate so 2133 MT/s

- Address pins are time-multiplexed
  - Row address strobe (RAS)
  - Column address strobe (CAS)

DRAM Chip Organization



- New RAS results in:
  - Bitline precharge
  - Row decode, sense
  - Row buffer write (up to 8K)
- New CAS
  - Read from row buffer
  - Much faster (3x)
- Streaming row accesses desirable

Simple Main Memory

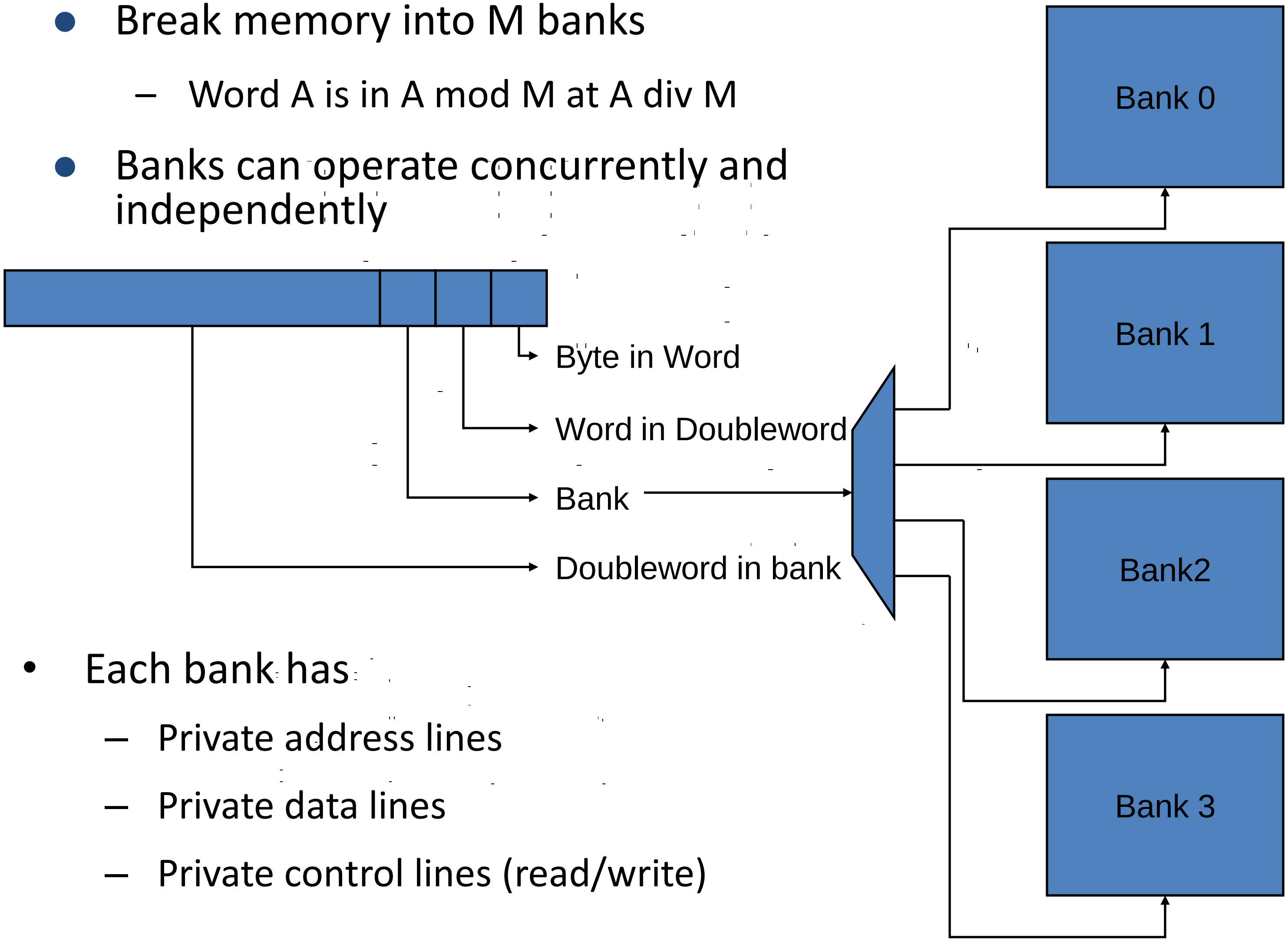
- Consider these parameters:
  - 10 cycles to send address
  - 60 cycles to access each word
  - 10 cycle to send word back
- Miss penalty for a 4-word block
  - (10 + 60 + 10) x 4 = 320
- How can we speed this up?

Wider(Parallel) Main Memory

- Make memory wider
  - Read out all words in parallel
- Memory parameters
  - 10 cycle to send address
  - 60 to access a double word
  - 10 cycle to send it back
- Miss penalty for 4-word block: 2x(10+60+10) = 160
- Costs
  - Wider bus
  - Larger minimum expansion unit (e.g. paired DIMMs)

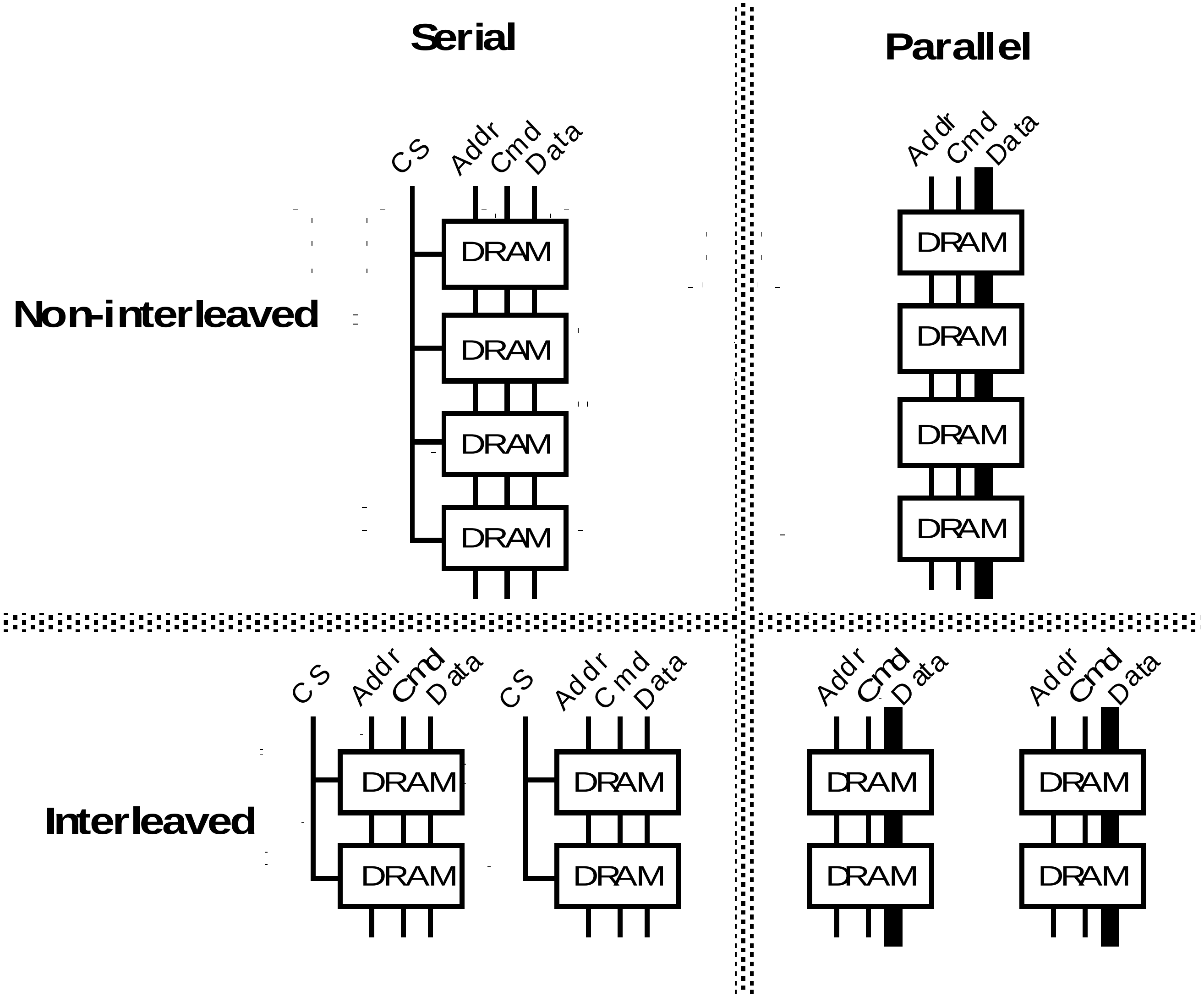
Interleaved Main Memory

- Break memory into M banks
  - Word A is in A mod M at A div M
- Banks can operate concurrently and independently



- Each bank has
  - Private address lines
  - Private data lines
  - Private control lines (read/write)

Interleaved and Parallel Organization

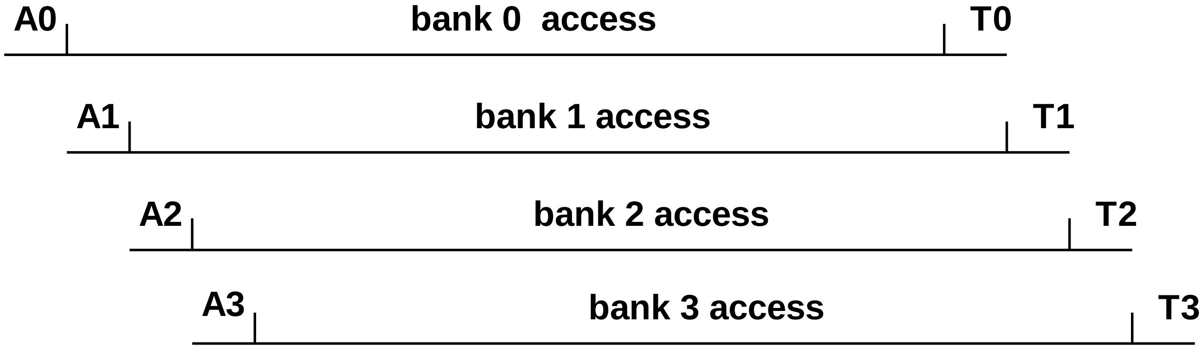


Interleaved Memory Examples

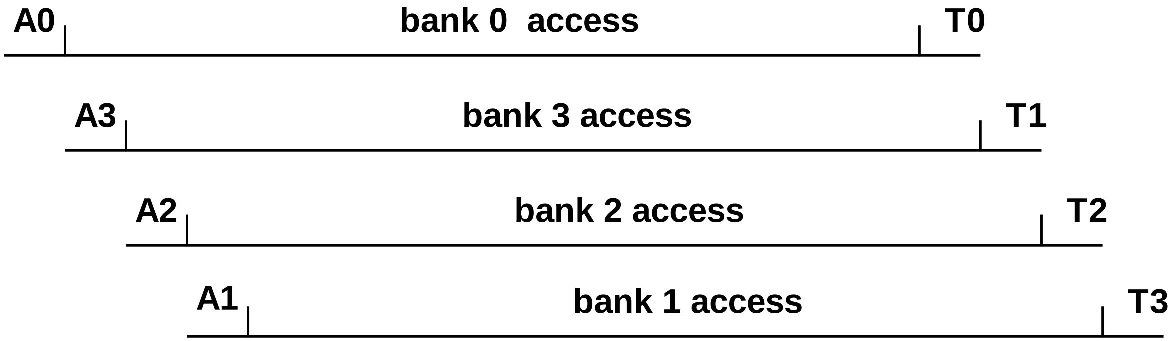
Ai = address to bank i

Ti = data transfer

- Unit Stride:



- Stride 3:



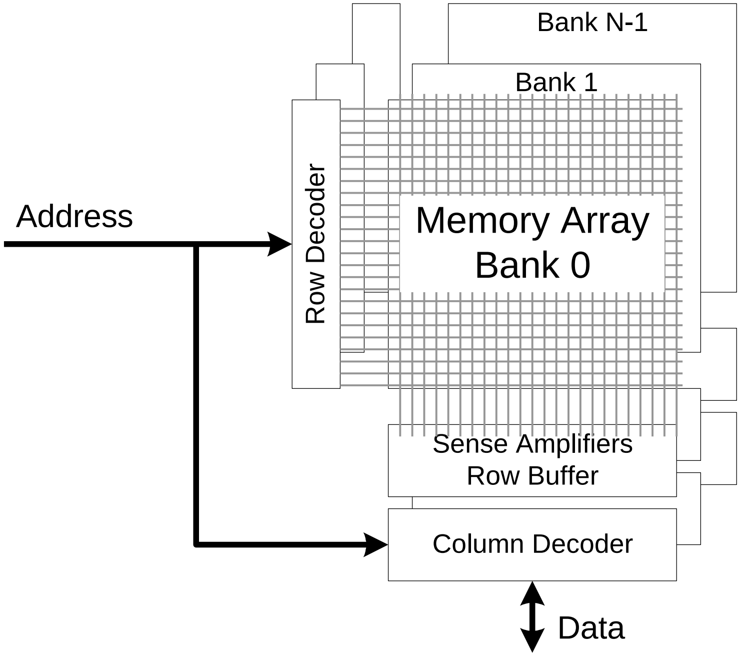
DDR SDRAM Control

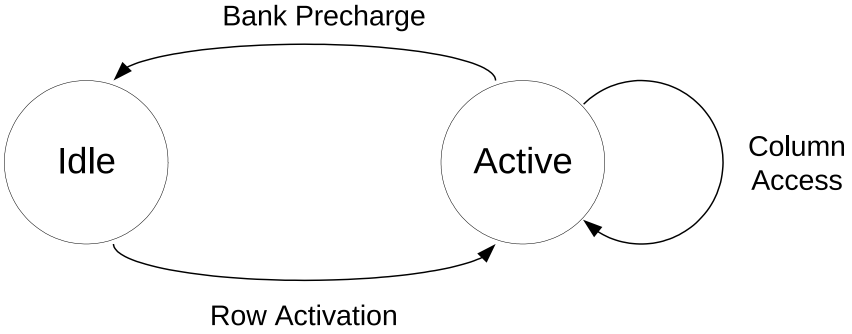
- **Raise level of abstraction: commands**
  - Activate row

    Read row into row buffer
  - Column access

    Read data from addressed row
  - Bank Precharge

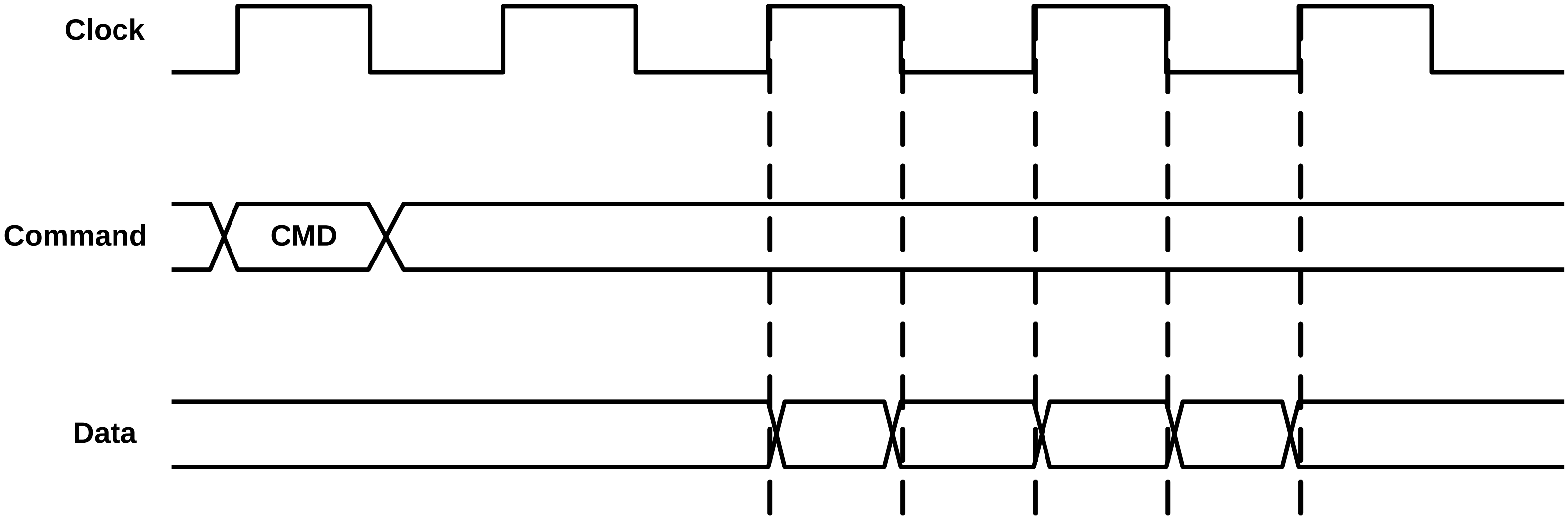
    Get ready for new row access





DDR SDRAM Timing

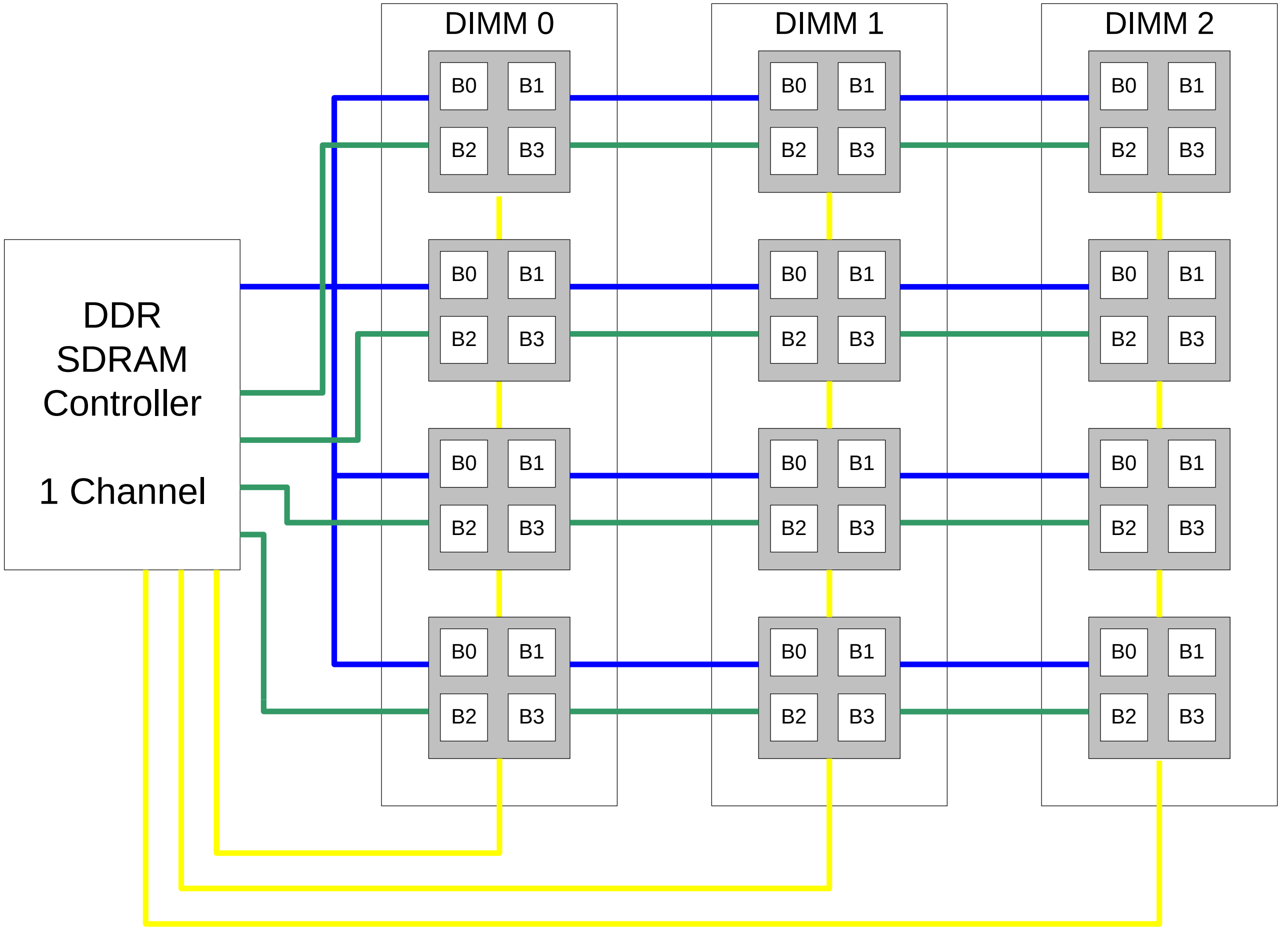
- **Read access**



Constructing a Memory System

- Combine chips in parallel to increase access width
  - E.g. 4 16-bit wide DRAMs for a 64-bit parallel access
  - DIMM – Dual Inline Memory Module
- Combine DIMMs to form multiple *ranks*
- Attach a number to DIMMs to a memory channel
  - Memory Controller manages a channel (or two lock-step channels)
- Interleave patterns:
  - Rank, Row, Bank, Column, [byte]
  - Row, Rank, Bank, Column, [byte]
    - Better dispersion of addresses
    - Works better with power-of-two ranks

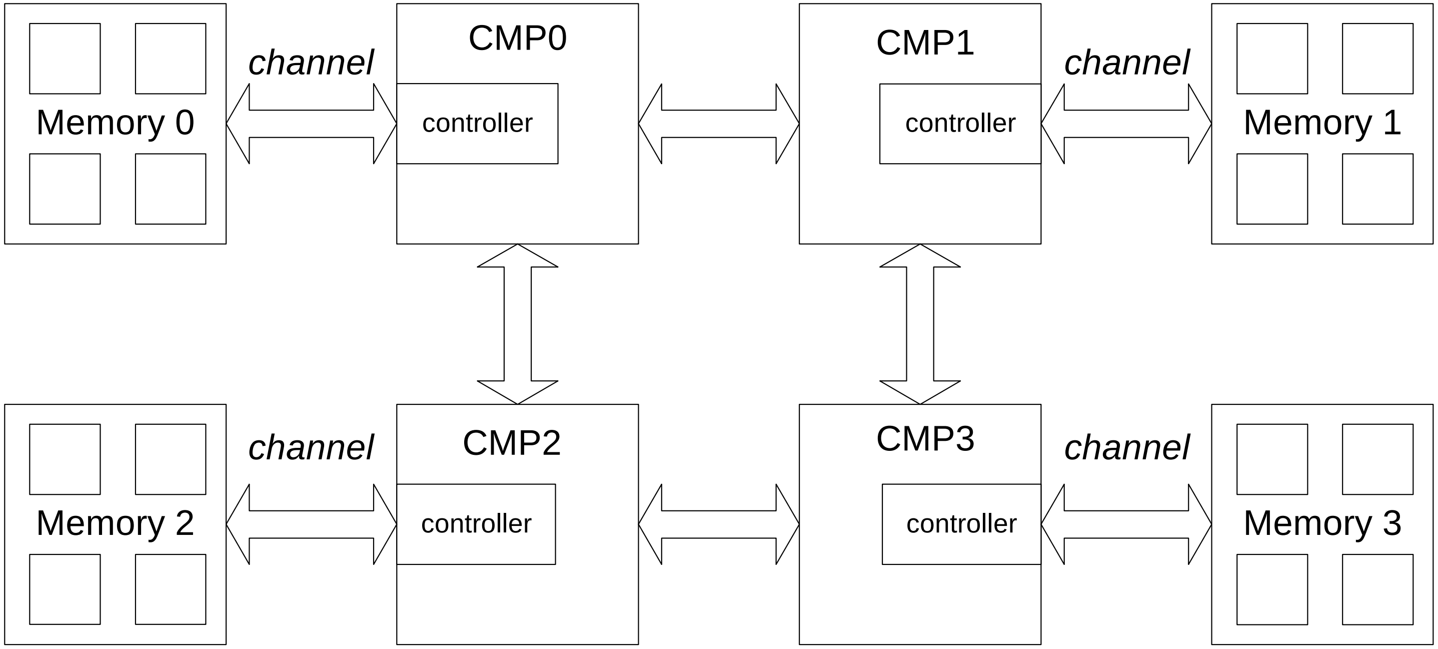
Memory Controller and Channel



🟨 chip (DIMM) select
🟩 data
🟦 address and command

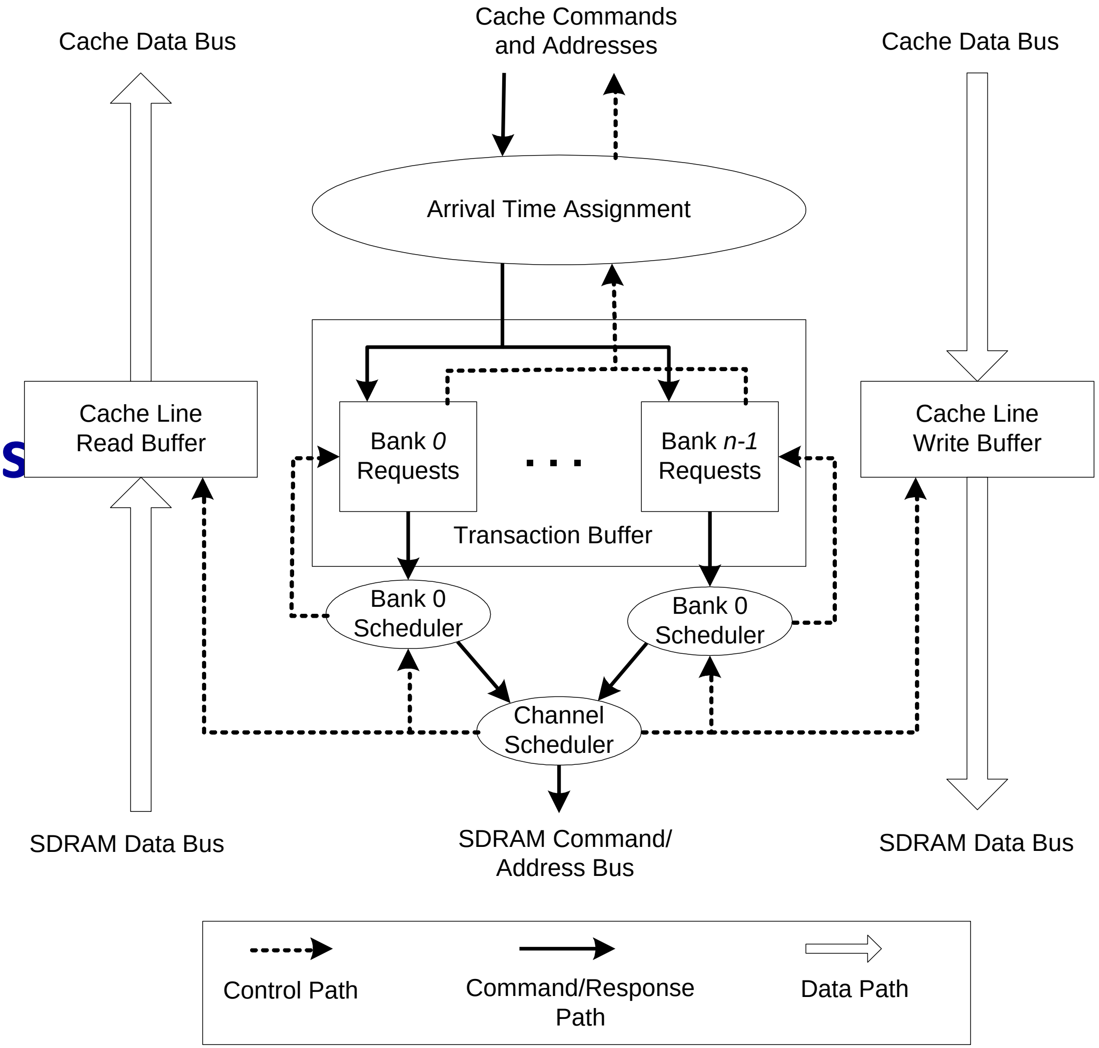
MP Memory Systems

- Memory controller can be centralized
  - Mostly in smaller systems
- More often distributed in larger (Multi-CMP) MP systems



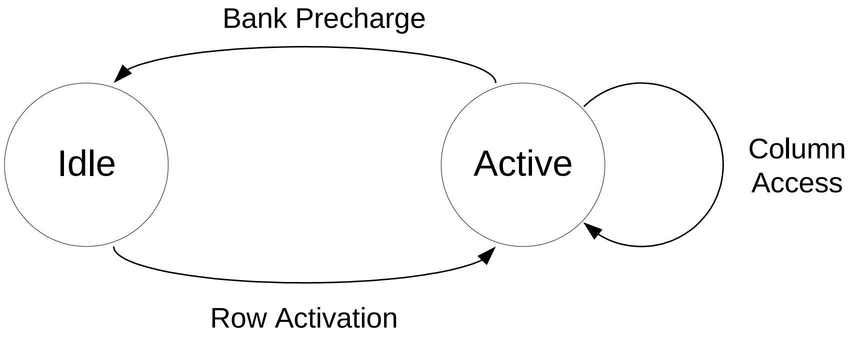
Memory Controllers

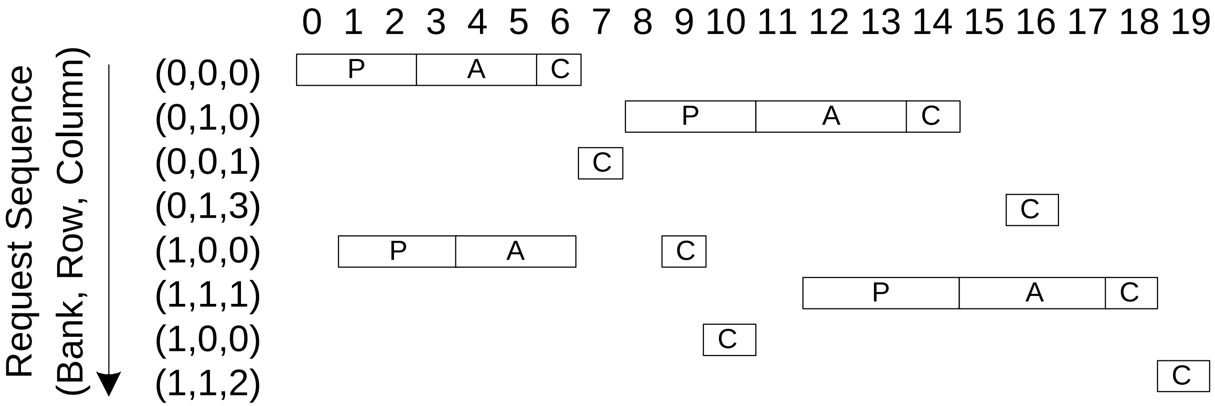
- Contains buffering
  - **In both directions**
- Scheduler's manage resources
  - **Channel and banks**



Resource Scheduling

- An interesting optimization problem
- Example:
  - **Precharge: 3 cycles**
  - **Row activate: 3 cycles**
  - **Column access: 1 cycle**
  - **FR-FCFS: 20 cycles**
  - **StrictFIFO: 56 cycles**





P: bank *P*recharge
A: row *A*ctivation
C: *C*olumn Access

DDR SDRAM Policies

- Goal: try to maximize requests to an open row (page)
- Close row policy
  - Always close row, hides precharge penalty
  - Lost opportunity if next access to same row
- Open row policy
  - Leave row open
  - If an access to a different row, then penalty for precharge
- Also performance issues related to rank interleaving
  - Better dispersion of addresses

Study by Natarajan et al.

[C. Natarajan, B. Christenson, and F. Briggs, “A Study of Performance Impact of Memory Controller Features in Multi-Processor Server Environment,” Proc. of the 3rd Workshop on Memory Perf. Issues, June 2004]

- Intel Servers
  - IA-32 : In-order front-side bus to processor
  - IPF: Out-of-order front-side bus to processor
- Use server Benchmarks
- Open row (page) policy used in desktop systems
  - Performs poorly in Server systems
- “Open-Policy2-0” works much better
  - **Leave open only until there is an idle cycle for the bank; then close**

Results #3

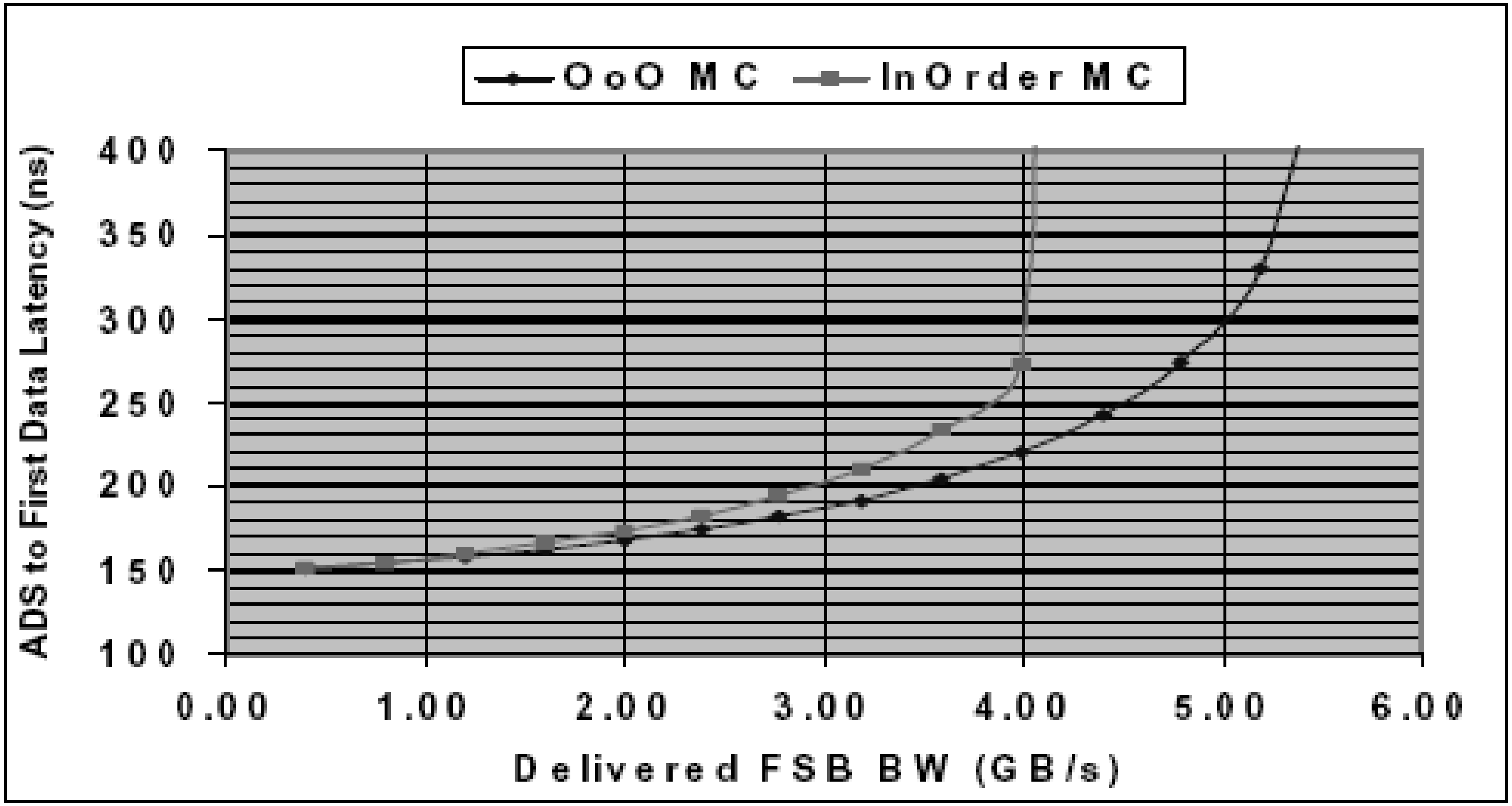
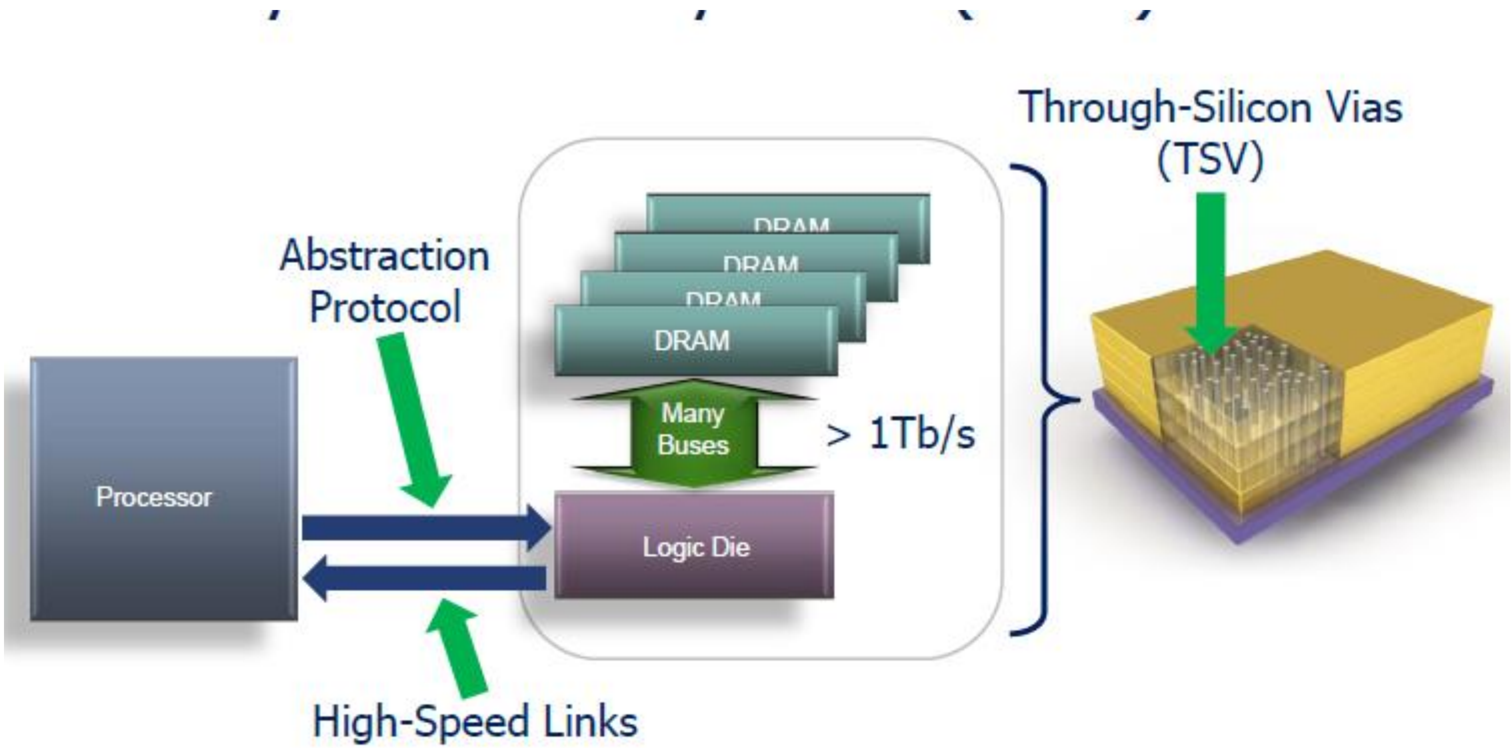


Fig. 4. Memory Controller Feature Impact Study#3 Latency-vs-BW Results

Memory Scheduling Contest

- http://www.cs.utah.edu/~rajeev/jwac12/
- Clean, simple, infrastructure
- Traces provided
- Very easy to make fair comparisons
- Comes with 6 schedulers
- Also targets power-down modes (not just page open/close scheduling)
- Three tracks:
  1. Delay (or Performance),
  2. Energy-Delay Product (EDP)
  3. Performance-Fairness Product (PFP)

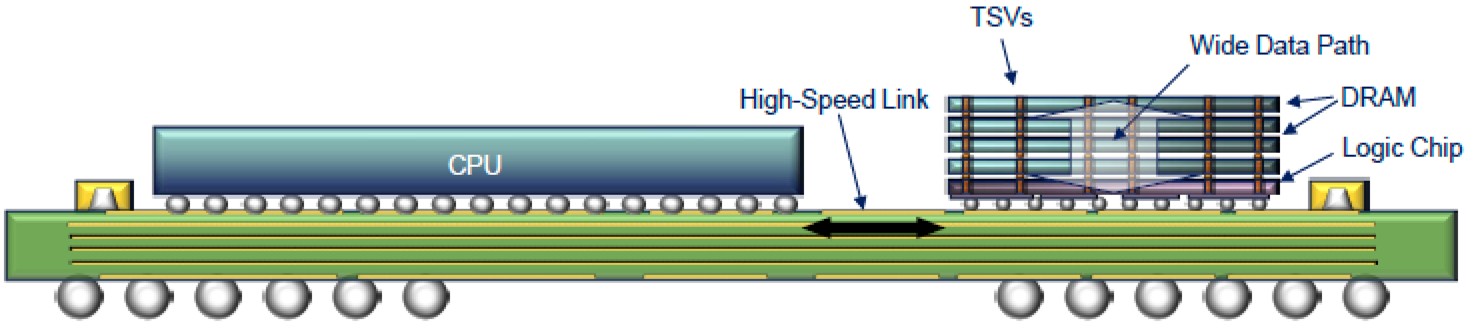
Future: Hybrid Memory Cube



Notes: Tb/s = Terabits / second
HMC height is exaggerated

- Micron proposal [Pawlowski, Hot Chips 11]
  - www.hybridmemorycube.org

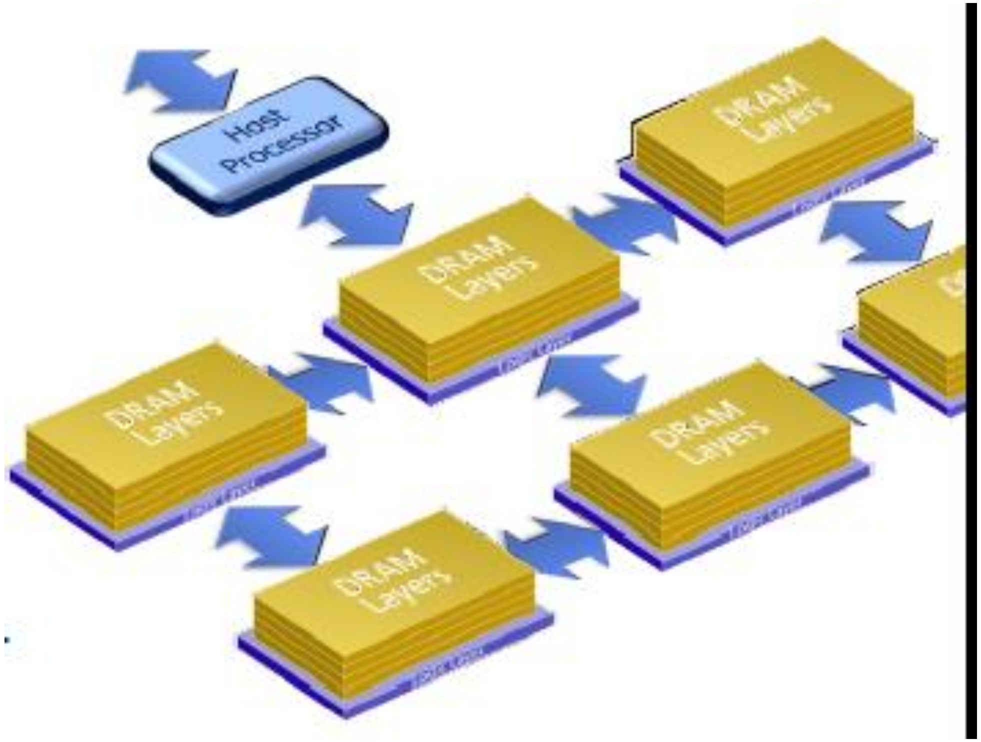
Hybrid Memory Cube MCM



Notes: MCM = multi-chip module
Illustrative purposes only; height is exaggerated

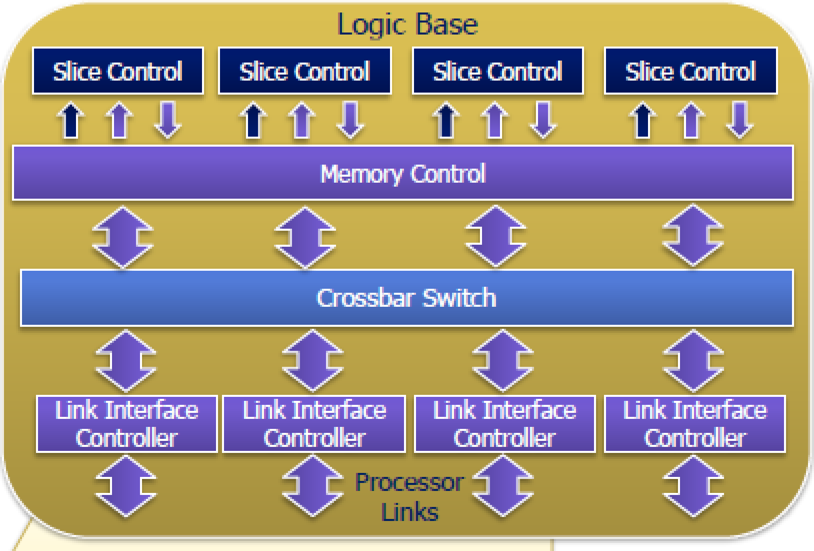
- Micron proposal [Pawlowski, Hot Chips 11]
  - www.hybridmemorycube.org

Network of DRAM



- Traditional DRAM: star topology
- HMC: mesh, etc. are feasible

Hybrid Memory Cube

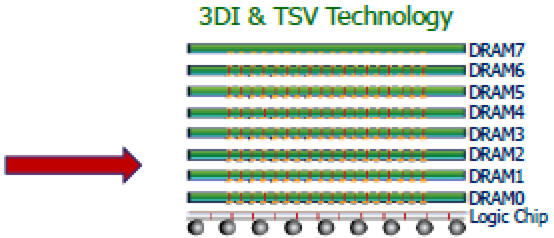


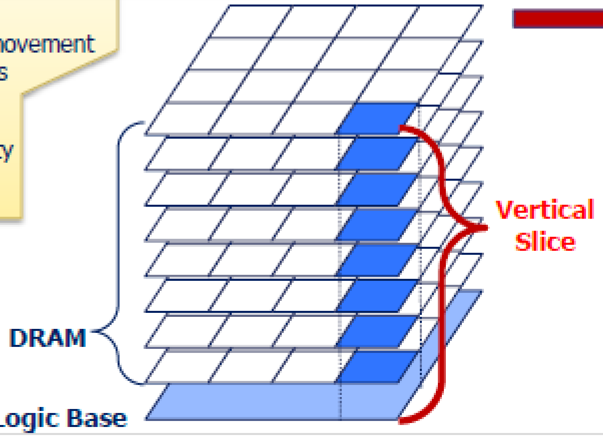
Logic Base

- Wide, high-speed local bus for data movement
- Advanced memory controller functions
- DRAM control at memory rather than distant host controller
- Reduced memory controller complexity and increased efficiency

Add sophisticated switching and optimized memory control...

And now we have a whole new set of capabilities



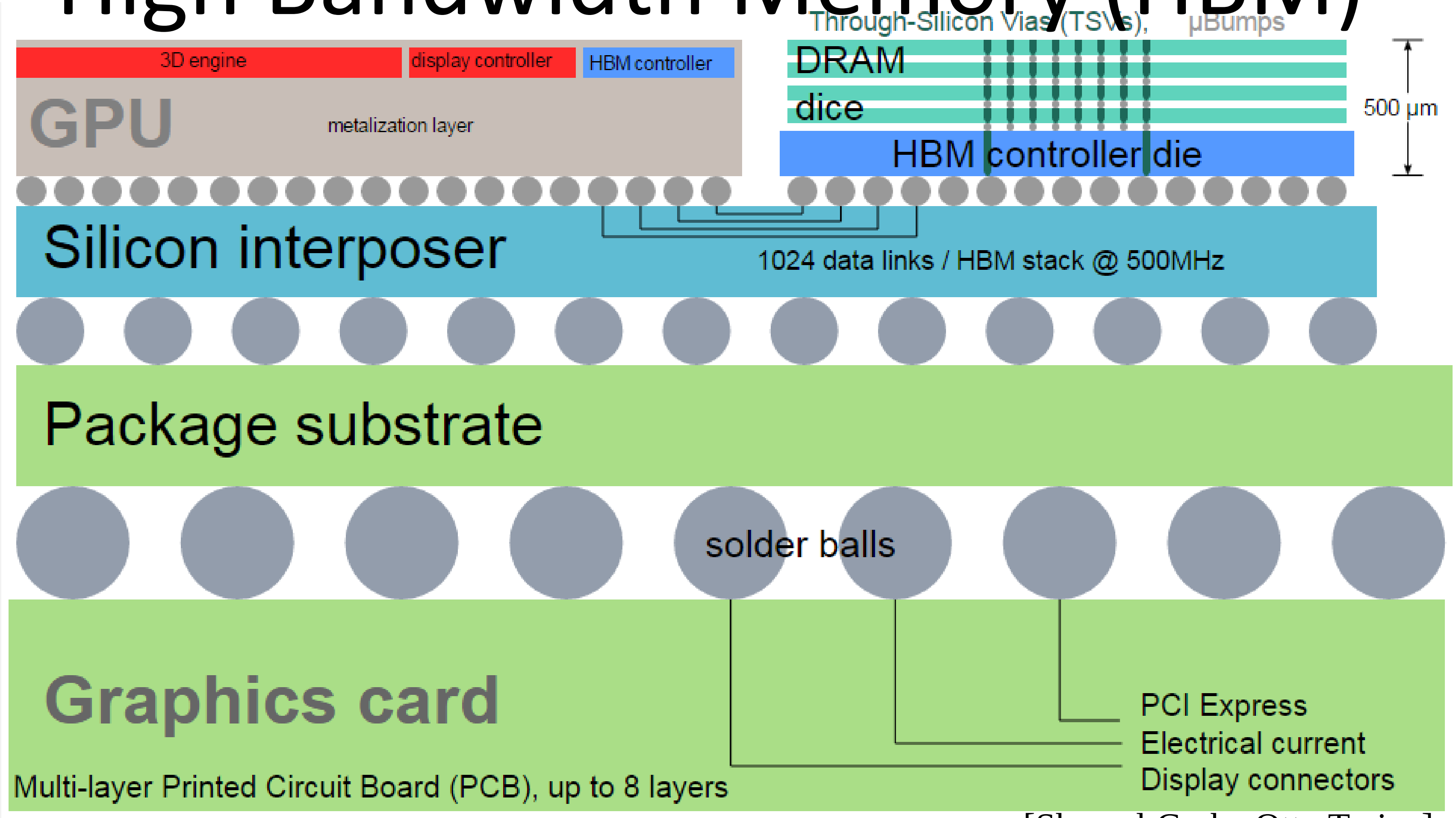


Vertical Slices are managed to maximize overall device availability

- Optimized management of energy and refresh
- Self test, error detection, correction, and repair in the logic base layer

- High-speed logic segregated in chip stack
- 3D TSV for bandwidth

High Bandwidth Memory (HBM)



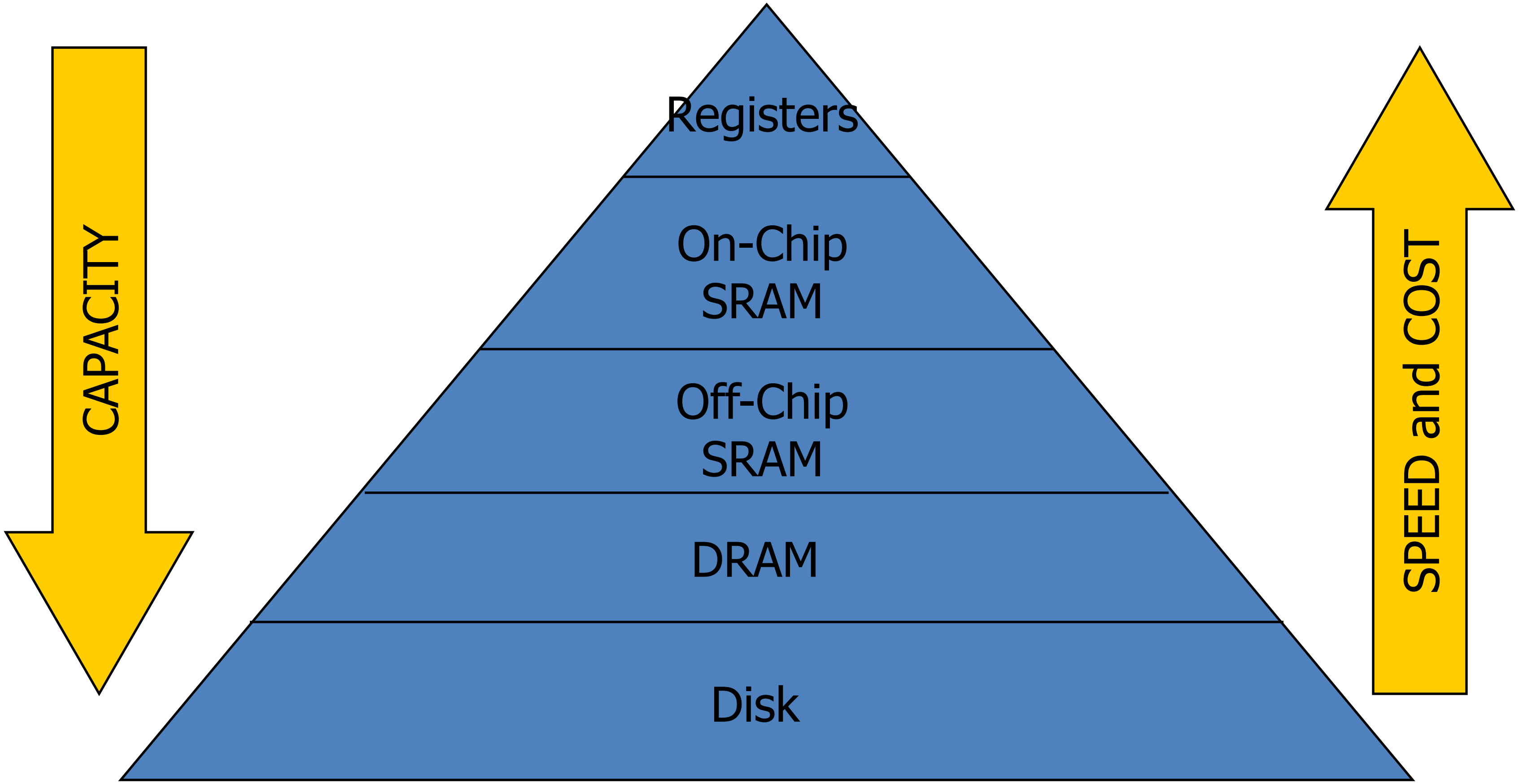
[Shmuel Csaba Otto Traian]

- High-speed serial links vs. 2.5D silicon interposer
- Commercialized, HBM2/HBM3 on the way

Future: Resistive memory

- PCM: store bit in phase state of material
- Alternatives:
  - Memristor (HP Labs)
  - STT-MRAM
- Nonvolatile
- Dense: crosspoint architecture (no access device)
- Relatively fast for read
- Very slow for write (also high power)
- Write endurance often limited
  - Write leveling (also done for flash)
  - Avoid redundant writes (read, cmp, write)
  - Fix individual bit errors (write, read, cmp, fix)

Memory Hierarchy



Why Memory Hierarchy?

- Need lots of bandwidth

$$BW = \frac{1.0 inst}{cycle} \times \left[\frac{1 Ifetch}{inst} \times \frac{4B}{Ifetch} + \frac{0.4 Dref}{inst} \times \frac{4B}{Dref} \right] \times \frac{1 Gcycles}{\sec}$$

$$= \frac{5.6 GB}{\sec}$$

- Need lots of storage
  - 64MB (minimum) to multiple TB
- Must be cheap per bit
  - (TB x anything) is a lot of money!
- These requirements seem incompatible

Why Memory Hierarchy?

- Fast and small memories
  - Enable quick access (fast cycle time)
  - Enable lots of bandwidth (1+ L/S/I-fetch/cycle)
- Slower larger memories
  - Capture larger share of memory
  - Still relatively fast
- Slow huge memories
  - Hold rarely-needed state
  - Needed for correctness
- All together: provide appearance of large, fast memory with cost of cheap, slow memory

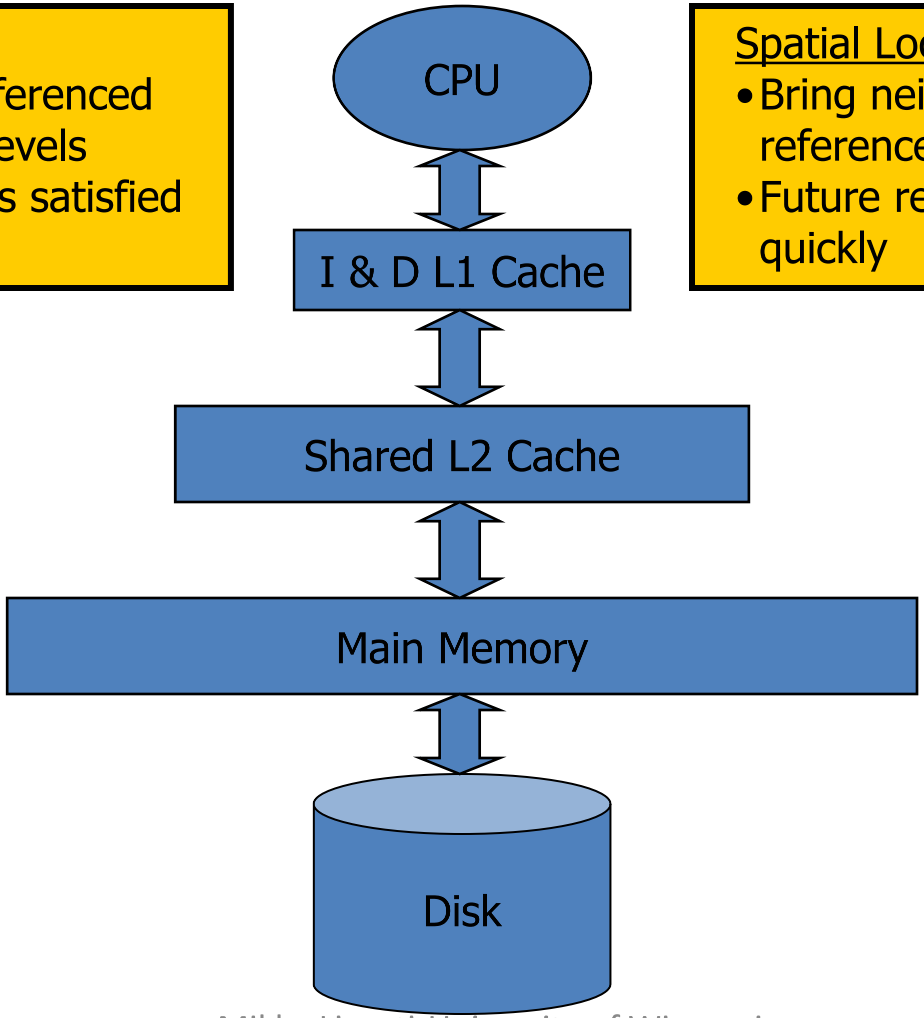
Why Does a Hierarchy Work?

- Locality of reference
  - Temporal locality
    - Reference same memory location repeatedly
  - Spatial locality
    - Reference near neighbors around the same time
- Empirically observed
  - Significant!
  - Even small local storage (8KB) often satisfies >90% of references to multi-MB data set

Memory Hierarchy

Temporal Locality

- Keep recently referenced items at higher levels
- Future references satisfied quickly



Spatial Locality

- Bring neighbors of recently referenced to higher levels
- Future references satisfied quickly

Four Burning Questions

- These are:
  - Placement
    - Where can a block of memory go?
  - Identification
    - How do I find a block of memory?
  - Replacement
    - How do I make space for new blocks?
  - Write Policy
    - How do I propagate changes?
- Consider these for caches
  - Built from SRAM, EDRAM, stacked DRAM

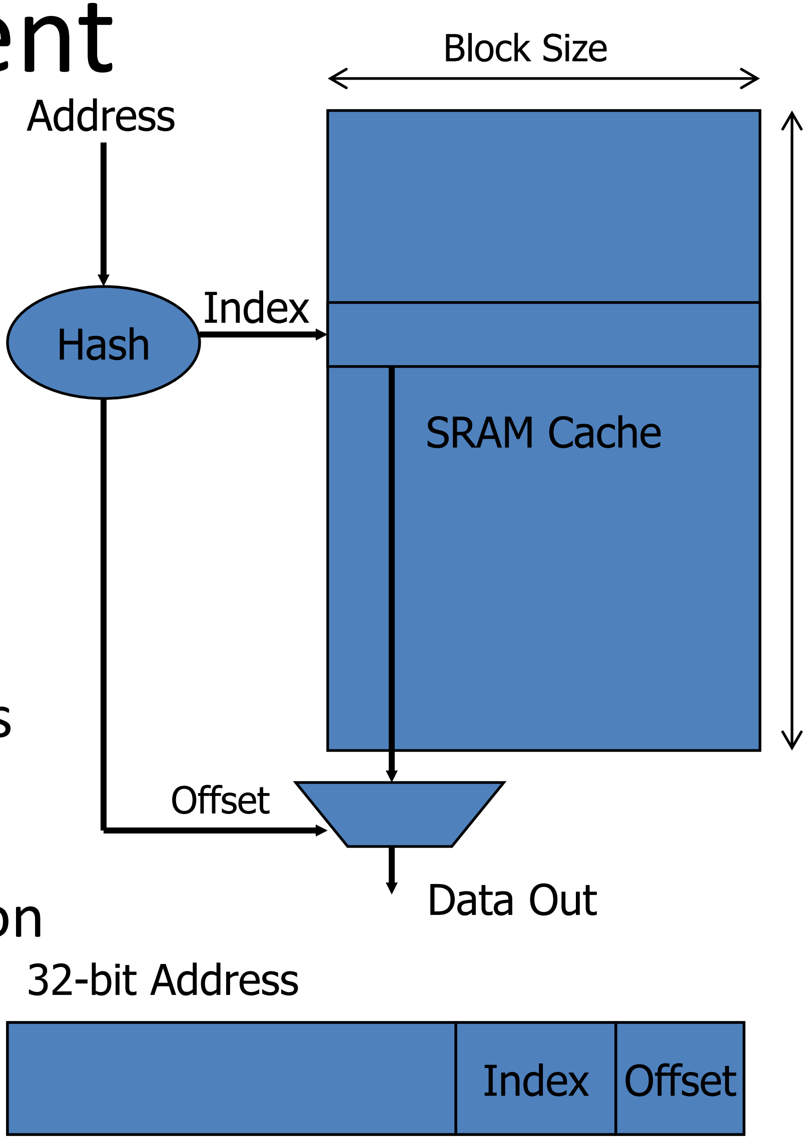
Placement

Memory Type	Placement	Comments
Registers	Anywhere; Int, FP, SPR	Compiler/programmer manages
Cache (SRAM)	Fixed in H/W	*Direct-mapped, set-associative, fully-associative*
DRAM	Anywhere	O/S manages
Disk	Anywhere	O/S manages

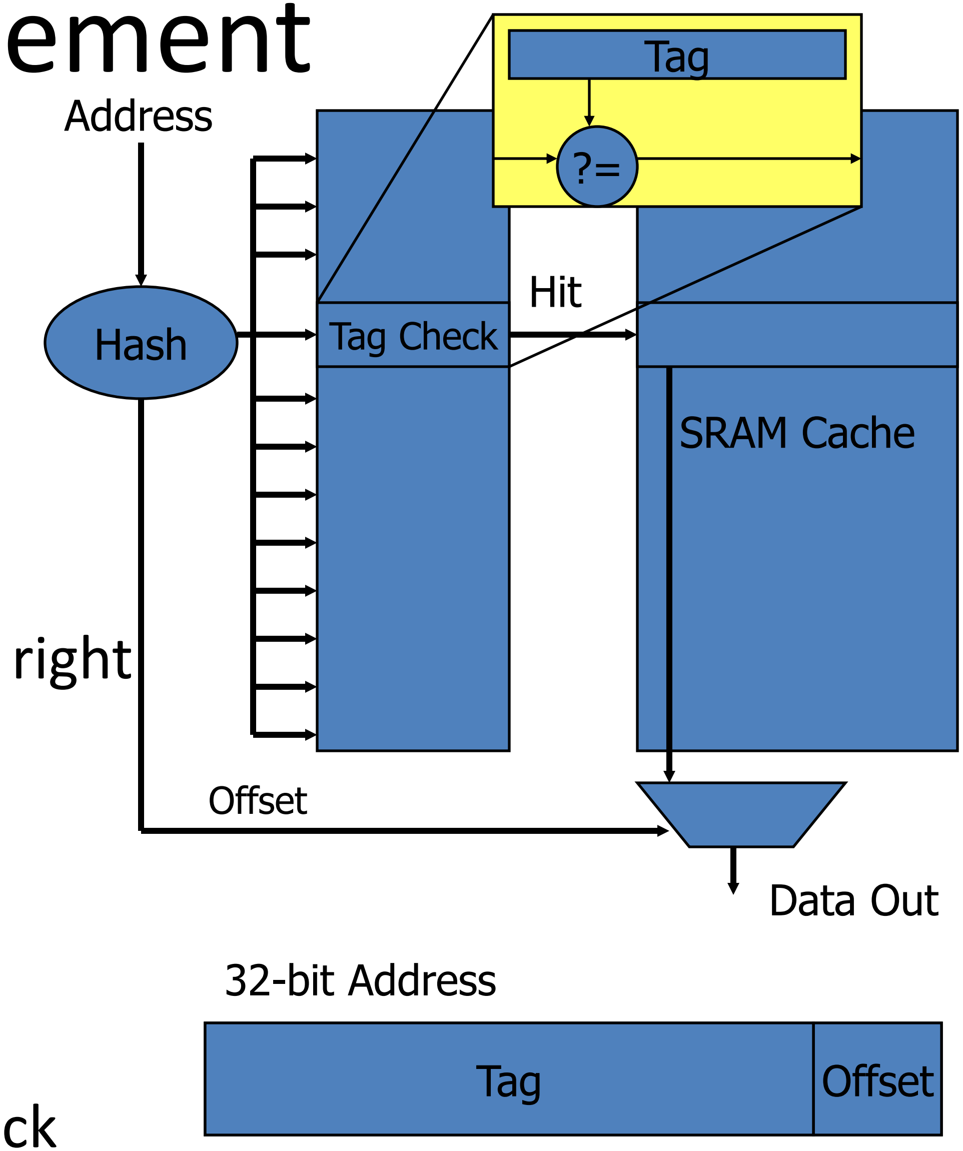
HUH?

Placement

- Address Range
  - Exceeds cache capacity
- Map address to finite capacity
  - Called a *hash*
  - Usually just masks high-order bits
- *Direct-mapped*
  - Block can only exist in one location
  - Hash collisions cause problems



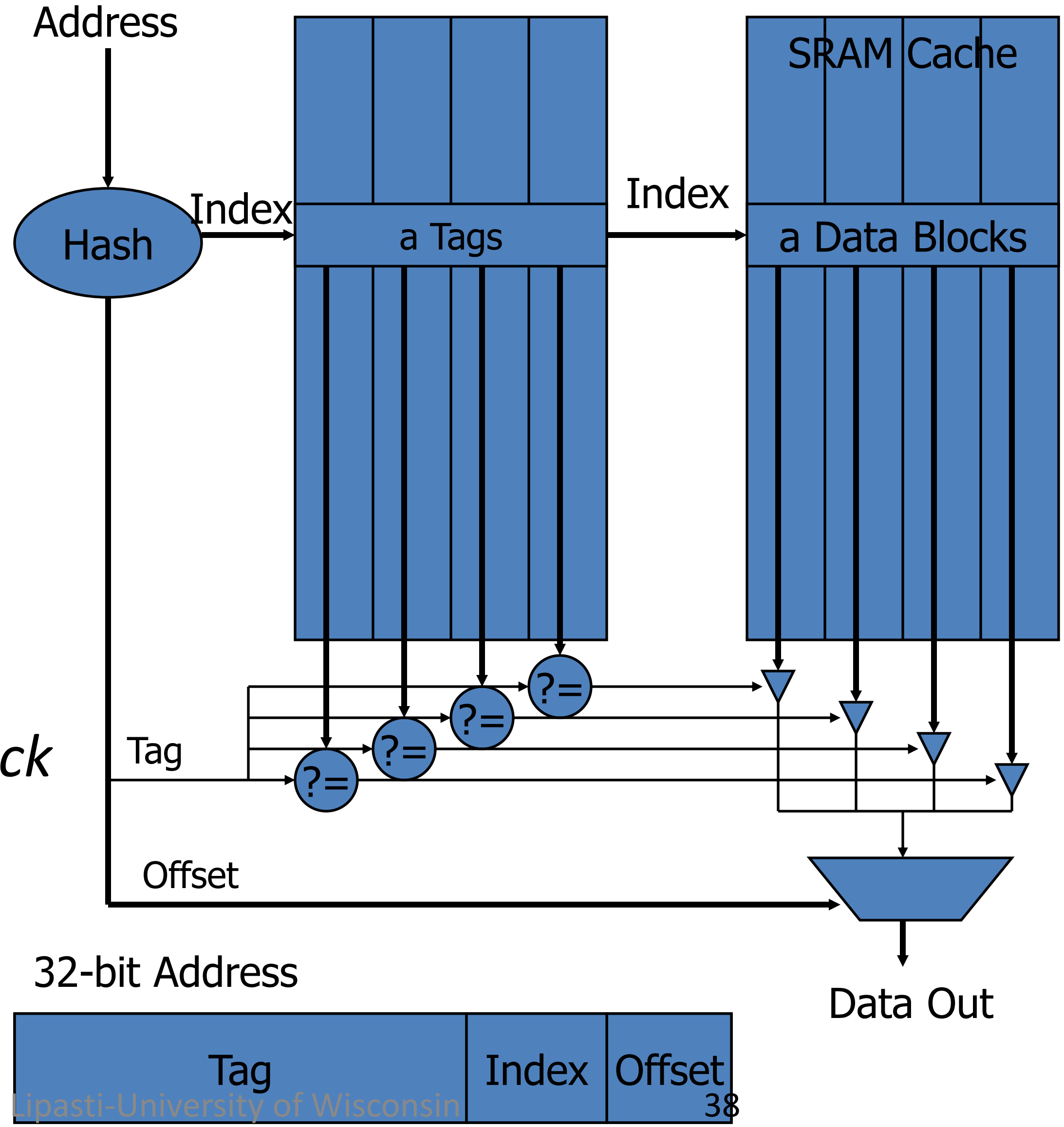
Placement



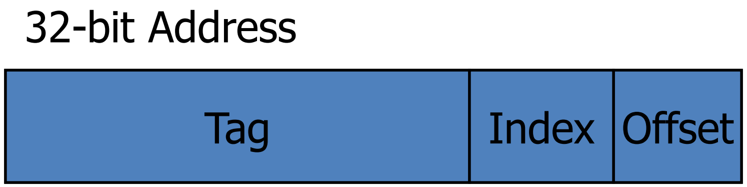
- *Fully-associative*
  - Block can exist anywhere
  - No more hash collisions
- *Identification*
  - How do I know I have the right block?
  - Called a *tag check*
    - Must store address tags
    - Compare against address
- Expensive!
  - Tag & comparator per block

Placement

- *Set-associative*
  - Block can be in *a* locations
  - Hash collisions:
    - *a* still OK
- *Identification*
  - Still perform *tag check*
  - However, only *a* in parallel



Placement and Identification


32-bit Address

Tag

Index

Offset

Portion	Length	Purpose
Offset	$o=\log_2(\text{block size})$	Select word within block
Index	$i=\log_2(\text{number of sets})$	Select set of blocks
Tag	$t=32 - o - i$	ID block within set

- Consider: <BS=block size, S=sets, B=blocks>
  - <64,64,64>: o=6, i=6, t=20: direct-mapped (S=B)
  - <64,16,64>: o=6, i=4, t=22: 4-way S-A (S = B / 4)
  - <64,1,64>: o=6, i=0, t=26: fully associative (S=1)
- Total size = BS x B = BS x S x (B/S)

Replacement

- Cache has finite size
  - What do we do when it is full?
- Analogy: desktop full?
  - Move books to bookshelf to make room
- Same idea:
  - Move blocks to next level of cache

Replacement

- How do we choose *victim*?
  - Verbs: *Victimize, evict, replace, cast out*
- Several policies are possible
  - FIFO (first-in-first-out)
  - LRU (least recently used)
  - NMRU (not most recently used)
  - Pseudo-random (yes, really!)
- Pick victim within *set* where a = *associativity*
  - If a <= 2, LRU is cheap and easy (1 bit)
  - If a > 2, it gets harder
  - Pseudo-random works pretty well for caches

Write Policy

- Memory hierarchy
  - – 2 or more copies of same block
    - Main memory and/or disk
    - Caches
- What to do on a write?
  - – Eventually, all copies must be changed
  - – Write must *propagate* to all levels
    - And other processor’s caches (later)

Write Policy

- Easiest policy: *write-through*
- Every write propagates directly through hierarchy
  - Write in L1, L2, memory, disk (?!?)
- Why is this a bad idea?
  - Very high bandwidth requirement
  - Remember, large memories are slow
- Popular in real systems only to the L2
  - Every write updates L1 and L2
  - Beyond L2, use *write-back* policy

Write Policy

- Most widely used: *write-back*
- Maintain *state* of each line in a cache
  - Invalid – not present in the cache
  - Clean – present, but not written (unmodified)
  - Dirty – present and written (modified)
- Store state in tag array, next to address tag
  - Mark dirty bit on a write
- On eviction, check dirty bit
  - If set, write back dirty line to next level
  - Called a *writeback* or *castout*

Write Policy

- Complications of write-back policy
  - Stale copies lower in the hierarchy
  - Must always check higher level for dirty copies before accessing copy in a lower level
- Not a big problem in uniprocessors
  - In multiprocessors: *the cache coherence problem*
- I/O devices that use DMA (direct memory access) can cause problems even in uniprocessors
  - Called coherent I/O
  - Must check caches for dirty copies before reading main memory

Cache Example

- 32B Cache: <BS=4,S=4,B=8>
  - o=2, i=2, t=2; 2-way set-associative
  - Initially empty
  - Only tag array shown on right
- Trace execution of:

Reference	Binary	Set/Way	Hit/Miss

Tag Array

Tag0	Tag1	LRU
		0
		0
		0
		0

Cache Example

- 32B Cache: <BS=4,S=4,B=8>
  - o=2, i=2, t=2; 2-way set-associative
  - Initially empty
  - Only tag array shown on right
- Trace execution of:

Reference	Binary	Set/Way	Hit/Miss
Load 0x2A	101010	2/0	Miss

Tag Array

Tag0	Tag1	LRU
		0
		0
10		1
		0

Cache Example

- 32B Cache: <BS=4,S=4,B=8>
  - o=2, i=2, t=2; 2-way set-associative
  - Initially empty
  - Only tag array shown on right
- Trace execution of:

Reference	Binary	Set/Way	Hit/Miss
Load 0x2A	101010	2/0	Miss
Load 0x2B	101011	2/0	Hit

Tag Array

Tag0	Tag1	LRU
		0
		0
10		1
		0

Cache Example

- 32B Cache: <BS=4,S=4,B=8>
  - o=2, i=2, t=2; 2-way set-associative
  - Initially empty
  - Only tag array shown on right
- Trace execution of:

Reference	Binary	Set/Way	Hit/Miss
Load 0x2A	101010	2/0	Miss
Load 0x2B	101011	2/0	Hit
Load 0x3C	111100	3/0	Miss

Tag Array

Tag0	Tag1	LRU
		0
		0
10		1
11		1

Cache Example

- 32B Cache: <BS=4,S=4,B=8>
  - o=2, i=2, t=2; 2-way set-associative
  - Initially empty
  - Only tag array shown on right
- Trace execution of:

Reference	Binary	Set/Way	Hit/Miss
Load 0x2A	101010	2/0	Miss
Load 0x2B	101011	2/0	Hit
Load 0x3C	111100	3/0	Miss
Load 0x20	100000	0/0	Miss

Tag Array

Tag0	Tag1	LRU
10		1
		0
10		1
11		1

Cache Example

- 32B Cache: <BS=4,S=4,B=8>
  - o=2, i=2, t=2; 2-way set-associative
  - Initially empty
  - Only tag array shown on right
- Trace execution of:

Reference	Binary	Set/Way	Hit/Miss
Load 0x2A	101010	2/0	Miss
Load 0x2B	101011	2/0	Hit
Load 0x3C	111100	3/0	Miss
Load 0x20	100000	0/0	Miss
Load 0x33	110011	0/1	Miss

Tag Array

Tag0	Tag1	LRU
10	11	0
		0
10		1
11		1

Cache Example

- 32B Cache: <BS=4,S=4,B=8>
  - o=2, i=2, t=2; 2-way set-associative
  - Initially empty
  - Only tag array shown on right
- Trace execution of:

Reference	Binary	Set/Way	Hit/Miss
Load 0x2A	101010	2/0	Miss
Load 0x2B	101011	2/0	Hit
Load 0x3C	111100	3/0	Miss
Load 0x20	100000	0/0	Miss
Load 0x33	110011	0/1	Miss
Load 0x11	010001	0/0 (lru)	Miss/Evict

Tag Array

Tag0	Tag1	LRU
01	11	1
		0
10		1
11		1

Cache Example

- 32B Cache: <BS=4,S=4,B=8>
  - o=2, i=2, t=2; 2-way set-associative
  - Initially empty
  - Only tag array shown on right
- Trace execution of:

Reference	Binary	Set/Way	Hit/Miss
Load 0x2A	101010	2/0	Miss
Load 0x2B	101011	2/0	Hit
Load 0x3C	111100	3/0	Miss
Load 0x20	100000	0/0	Miss
Load 0x33	110011	0/1	Miss
Load 0x11	010001	0/0 (lru)	Miss/Evict
Store 0x29	101001	2/0	Hit/Dirty

Tag Array

Tag0	Tag1	LRU
01	11	1
		0
10 d		1
11		1

Lecture 5 Outline

- Main Memory and Cache Review
- Caches and Replacement Policies
- Cache Coherence
  - Coherence States
  - Snoopy bus-based Invalidate Protocols
  - Invalidate protocol optimizations
  - Update Protocols (Dragon/Firefly)
  - Directory protocols
  - Implementation issues

Cache Misses and Performance

- Miss penalty
  - Detect miss: 1 or more cycles
  - Find victim (replace block): 1 or more cycles
    - Write back if dirty
  - Request block from next level: several cycles
    - May need to **find** line from one of many caches (coherence)
  - Transfer block from next level: several cycles
    - (block size) / (bus width)
  - Fill block into data array, update tag array: 1+ cycles
  - Resume execution
- In practice: 6 cycles to 100s of cycles

Cache Miss Rate

- Determined by:
  - Program characteristics
    - Temporal locality
    - Spatial locality
  - Cache organization
    - Block size, associativity, number of sets

Improving Locality

- Instruction text placement
  - Profile program, place unreferenced or rarely referenced paths "elsewhere"
    - Maximize temporal locality
  - Eliminate taken branches
    - Fall-through path has spatial locality

Improving Locality

- Data placement, access order
  - Arrays: “block” loops to access subarray that fits into cache
    - Maximize temporal locality
  - Structures: pack commonly-accessed fields together
    - Maximize spatial, temporal locality
  - Trees, linked lists: allocate in usual reference order
    - Heap manager usually allocates sequential addresses
    - Maximize spatial locality
- Hard problem, not easy to automate:
  - C/C++ disallows rearranging structure fields
  - OK in Java

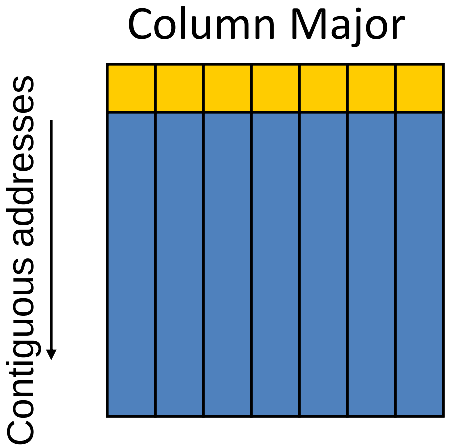
Software Restructuring

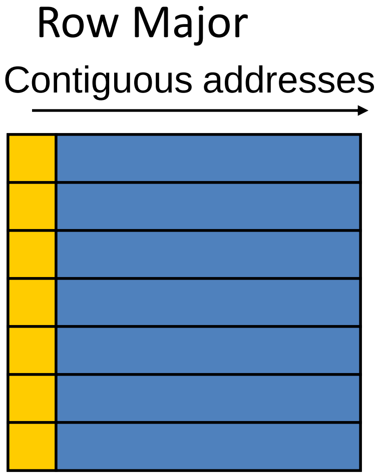
- If column-major (Fortran)
  - x[i+1, j] follows x [i,j] in memory
  - x[i,j+1] long after x[i,j] in memory
- Poor code

```
for i = 1, rows
  for j = 1, columns
    sum = sum + x[i,j]
```

- Conversely, if row-major (C/C++)
- Poor code

```
for j = 1, columns
  for i = 1, rows
    sum = sum + x[i,j]
```

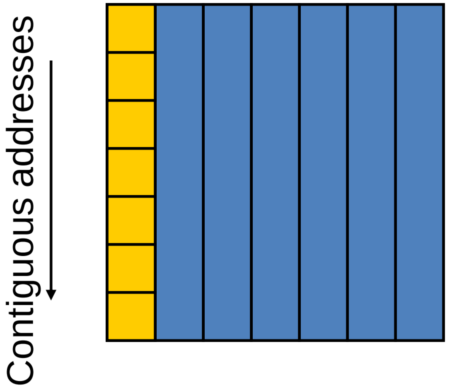




Software Restructuring

- **Better column-major code**

```
for j = 1, columns
  for i = 1, rows
    sum = sum + x[i,j]
```



- **Optimizations - need to check if it is valid to do them**
  - **Loop interchange (used above)**
  - **Merging arrays**
  - **Loop fusion**
  - **Blocking**

Cache Miss Rates: 3 C's [Hill]

- Compulsory miss
  - First-ever reference to a given block of memory
  - *Cold misses = m_c : number of misses for FA infinite cache*
- Capacity
  - Working set exceeds cache capacity
  - Useful blocks (with future references) displaced
  - *Capacity misses = m_f - m_c : add'l misses for finite FA cache*
- Conflict
  - Placement restrictions (not fully-associative) cause useful blocks to be displaced
  - Think of as *capacity within set*
  - *Conflict misses = m_a - m_f : add'l misses in actual cache*

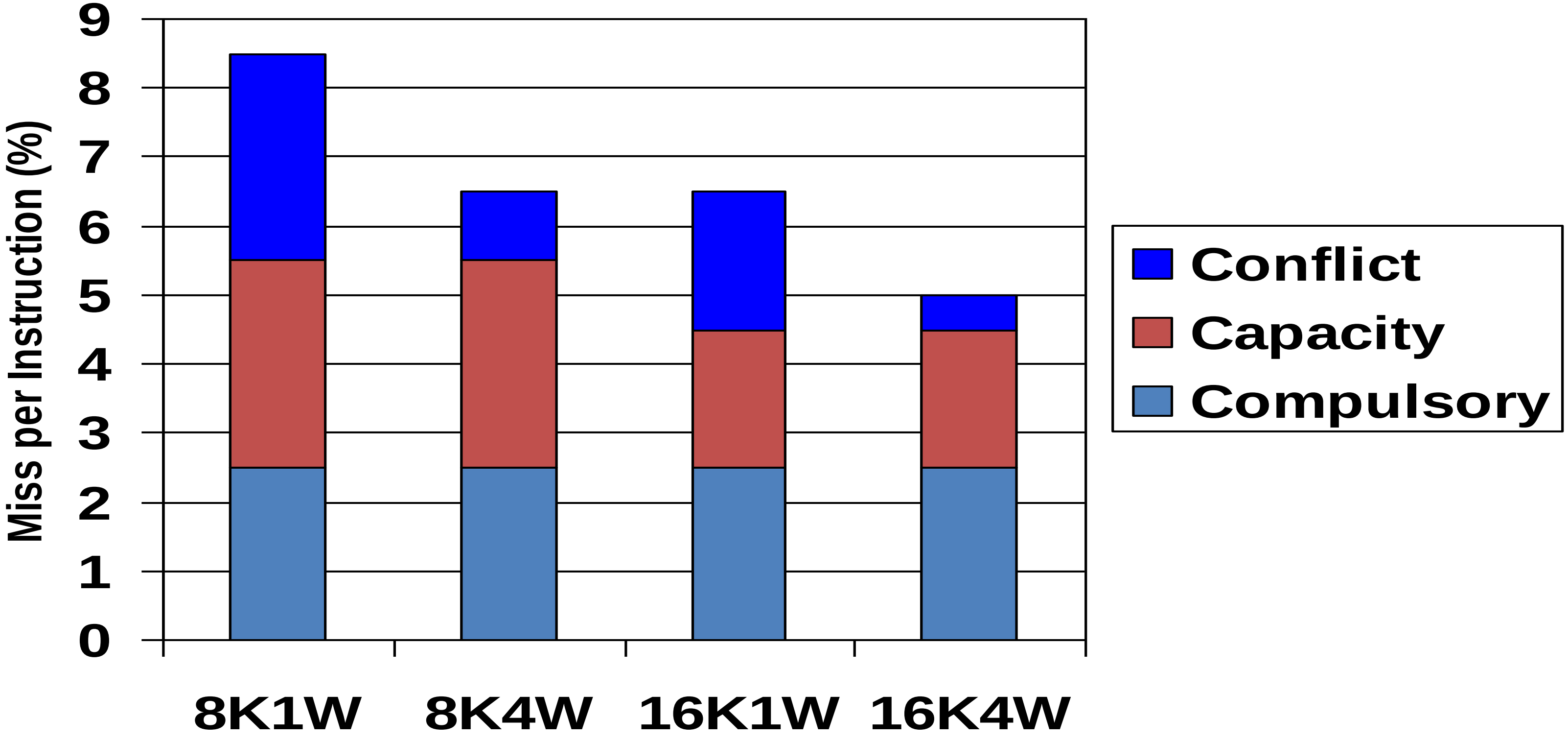
Cache Miss Rate Effects

- Number of blocks (sets x associativity)
  - Bigger is better: fewer conflicts, greater capacity
- Associativity
  - Higher associativity reduces conflicts
  - Very little benefit beyond 8-way set-associative
- Block size
  - Larger blocks exploit spatial locality
  - Usually: miss rates improve until 64B-256B
  - 512B or more miss rates get worse
    - Larger blocks less efficient: more capacity misses
    - Fewer placement choices: more conflict misses

Cache Miss Rate

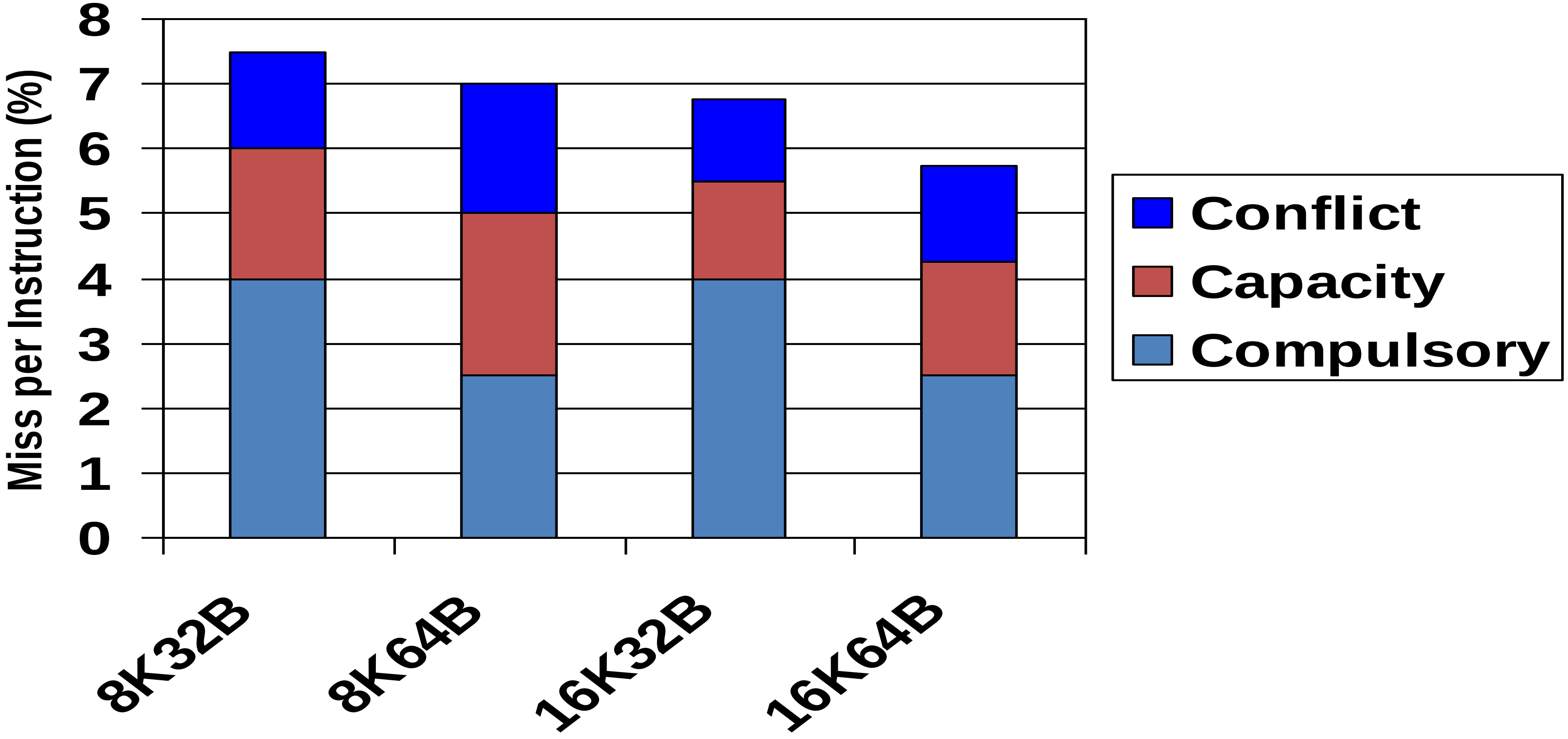
- Subtle tradeoffs between cache organization parameters
  - Large blocks reduce compulsory misses but increase miss penalty
    - #compulsory ~= (working set) / (block size)
    - #transfers = (block size)/(bus width)
  - Large blocks increase conflict misses
    - #blocks = (cache size) / (block size)
  - Associativity reduces conflict misses
  - Associativity increases access time
- Can associative cache ever have higher miss rate than direct-mapped cache of same size?

Cache Miss Rates: 3 C’s



- Vary size and associativity
  - Compulsory misses are constant
  - Capacity and conflict misses are reduced

Cache Miss Rates: 3 C's

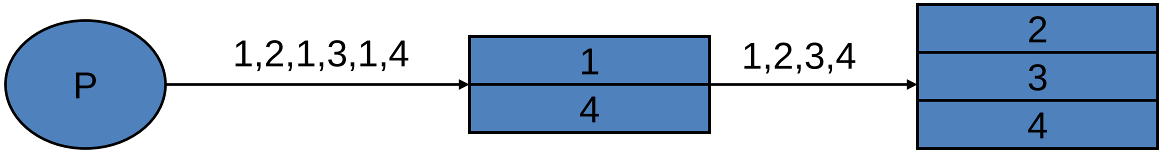


- Vary size and block size
  - Compulsory misses drop with increased block size
  - Capacity and conflict can increase with larger blocks

Multilevel Caches

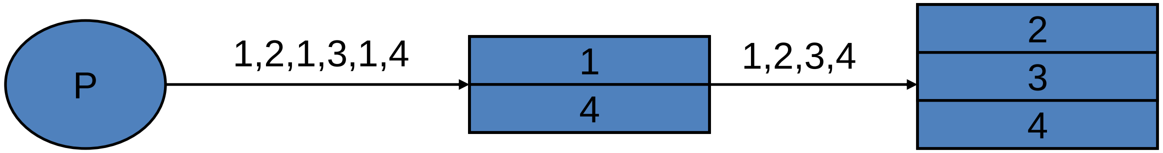
- Ubiquitous in high-performance processors
  - Gap between L1 (core frequency) and main memory too high
  - Level 2 usually on chip, level 3 on or off-chip, level 4 off chip
- Inclusion in multilevel caches
  - Multi-level inclusion holds if L2 cache is superset of L1
  - Can handle virtual address synonyms
  - Filter coherence traffic: if L2 misses, L1 needn’t see snoop
  - Makes L1 writes simpler
    - For both write-through and write-back

Multilevel Inclusion



- Example: local LRU not sufficient to guarantee inclusion
  - Assume L1 holds two and L2 holds three blocks
  - Both use local LRU
- Final state: L1 contains 1, L2 does not
  - Inclusion not maintained
- Different block sizes also complicate inclusion

Multilevel Inclusion



- Inclusion takes effort to maintain
  - Make L2 cache have bits or pointers giving L1 contents
  - Invalidate from L1 before replacing from L2
  - In example, removing 1 from L2 also removes it from L1
- Number of pointers per L2 block
  - L2 blocksize/L1 blocksize
- Supplemental reading: [Wang, Baer, Levy ISCA 1989]

Multilevel Miss Rates

- Miss rates of lower level caches
  - Affected by upper level filtering effect
  - LRU becomes LRM, since “use” is “miss”
  - Can affect miss rates, though usually not important
- Miss rates reported as:
  - Miss per instruction
  - Global miss rate
  - Local miss rate
  - “Solo” miss rate
    - L2 cache sees all references (unfiltered by L1)

Cache Design: Four Key Issues

- These are:
  - Placement
    - Where can a block of memory go?
  - Identification
    - How do I find a block of memory?
  - **Replacement**
    - **How do I make space for new blocks?**
  - Write Policy
    - How do I propagate changes?
- Consider these for caches
  - Usually SRAM
- Also apply to main memory, disks

Replacement

- Cache has finite size
  - What do we do when it is full?
- Analogy: desktop full?
  - Move books to bookshelf to make room
  - Bookshelf full? Move least-used to library
  - Etc.
- Same idea:
  - Move blocks to next level of cache

Replacement

- How do we choose *victim*?
  - Verbs: *Victimize, evict, replace, cast out*
- Many policies are possible
  - FIFO (first-in-first-out)
  - LRU (least recently used), pseudo-LRU
  - LFU (least frequently used)
  - NMRU (not most recently used)
  - NRU
  - Pseudo-random (yes, really!)
  - Optimal
  - Etc

Optimal Replacement Policy?

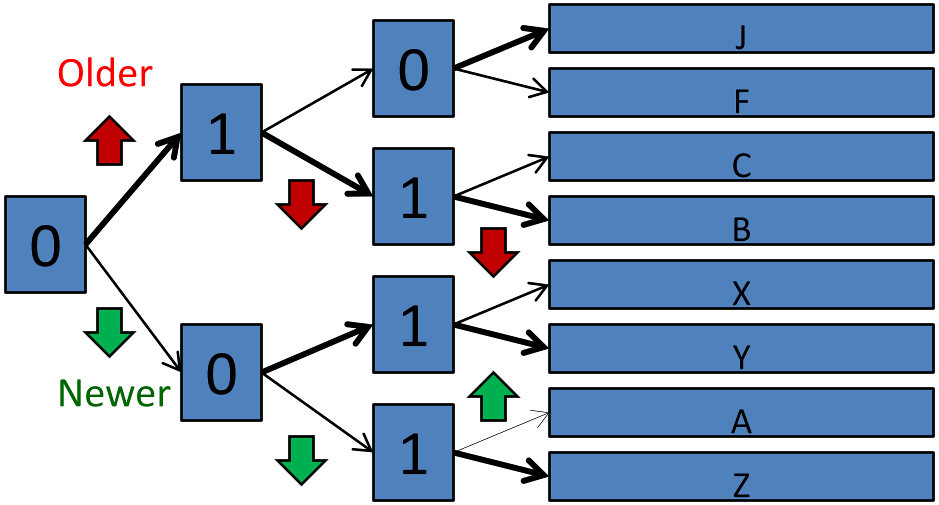
[Belady, IBM Systems Journal, 1966]

- Evict block with longest reuse distance
  - i.e. next reference to block is farthest in future
  - Requires knowledge of the future!
- Can’t build it, but can model it with trace
  - Process trace in reverse
  - [Sugumar&Abraham] describe how to do this in one pass over the trace with some lookahead (Cheetah simulator)
- Useful, since it reveals ***opportunity***
  - (X,A,B,C,D,X): LRU 4-way SA $, 2nd X will miss

Least-Recently Used

- For a=2, LRU is equivalent to NMRU
  - Single bit per set indicates LRU/MRU
  - Set/clear on each access
- For a>2, LRU is difficult/expensive
  - Timestamps? How many bits?
    - Must find min timestamp on each eviction
  - Sorted list? Re-sort on every access?
- List overhead: $log_2(a)$ bits /block
  - Shift register implementation

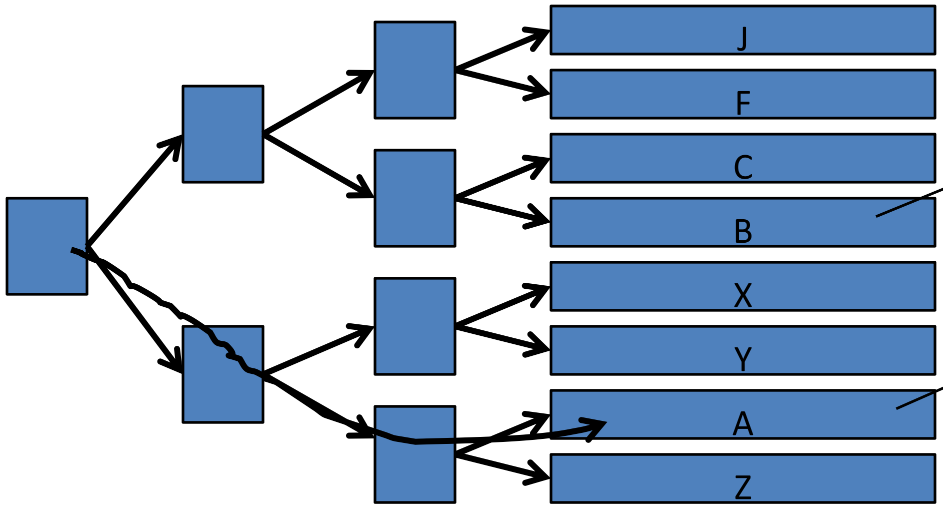
Practical Pseudo-LRU



- Rather than true LRU, use binary tree
- Each node records which half is older/newer
- Update nodes on each reference
- Follow older pointers to find LRU victim

Practical Pseudo-LRU In Action

J Y X Z B C F A



011: PLRU
Block B is here

110: MRU
block is here

Partial Order Encoded in Tree:

Z < A

Y < X

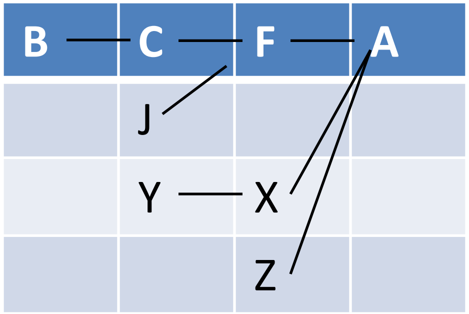
B < C

J < F

A > X

C < F

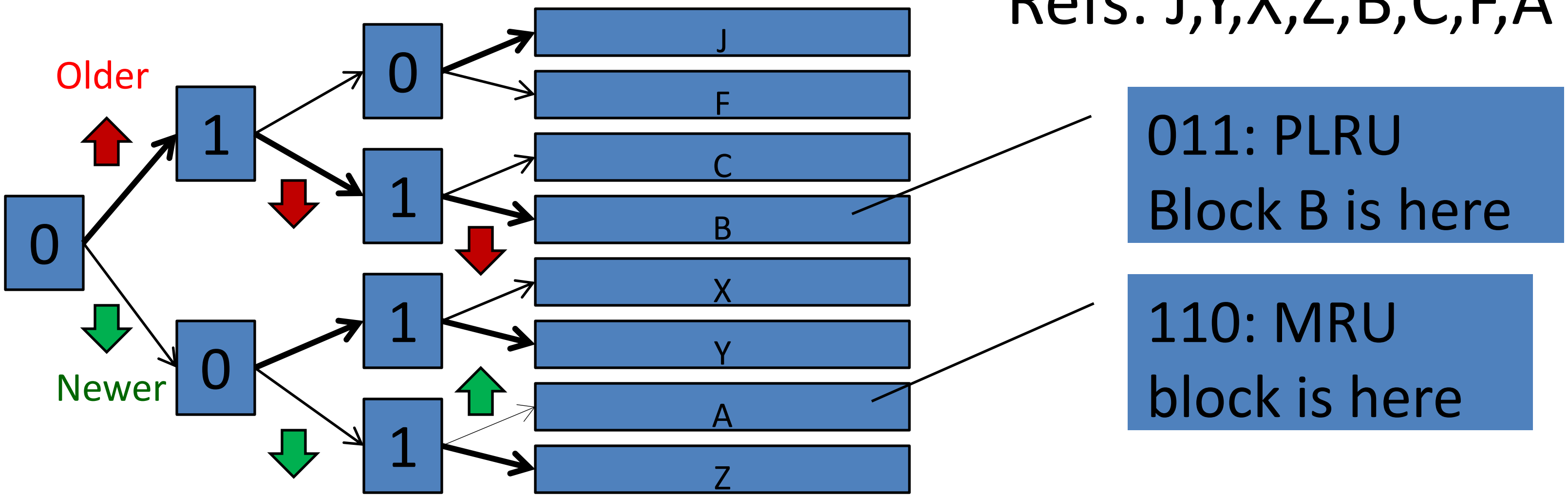
A > F



B	C	F	A
	J		
	Y	X	
		Z	

Practical Pseudo-LRU

Refs: J,Y,X,Z,B,C,F,A



011: PLRU
Block B is here

110: MRU
block is here

- Binary tree encodes PLRU *partial order*
  - At each level point to LRU half of subtree
- Each access: flip nodes along path to block
- Eviction: follow LRU path
- Overhead: *(a-1)/a* bits per block

True LRU Shortcomings

- Streaming data/scans: x_0, x_1, …, x_n
  - Effectively no temporal reuse
- Thrashing: *reuse distance > a*
  - Temporal reuse exists but LRU fails
- All blocks march from MRU to LRU
  - Other conflicting blocks are pushed out
- For *n>a* no blocks remain after scan/thrash
  - Incur many conflict misses after scan ends
- Pseudo-LRU sometimes helps a little bit

Segmented or Protected LRU

[I/O: Karedla, Love, Wherry, IEEE Computer 27(3), 1994]

[Cache: Wilkerson, Wade, US Patent 6393525, 1999]

- Partition LRU list into *filter* and *reuse* lists
- On insert, block goes into *filter* list
- On reuse (hit), block promoted into *reuse* list
- Provides scan & some thrash resistance
  - Blocks without reuse get evicted quickly
  - Blocks with reuse are protected from scan/thrash blocks
- No storage overhead, but LRU update slightly more complicated

Protected LRU: LIP

- Simplified variant of this idea: LIP
  - Qureshi et al. ISCA 2007
- Insert new blocks into LRU position, not MRU position
  - *Filter list* of size 1, *reuse list* of size (a-1)
- Do this adaptively: DIP
- Use *set dueling* to decide LIP vs. LRU
  - 1 (or a few) set uses LIP vs. 1 that uses LRU
  - Compare hit rate for sets
  - Set policy for all other sets to match best set

Not Recently Used (NRU)

- Keep NRU state in 1 bit/block
  - Bit is set to 0 when installed (assume reuse)
  - Bit is set to 0 when referenced (reuse observed)
  - Evictions favor NRU=1 blocks
  - If all blocks are NRU=0
    - Eviction forces all blocks in set to NRU=1
    - Picks one as victim (can be pseudo-random, or rotating, or fixed left-to-right)
- Simple, similar to virtual memory clock algorithm
- Provides some scan and thrash resistance
  - Relies on “randomizing” evictions rather than strict LRU order
- Used by Intel Itanium, Sparc T2

Least Frequently Used

- Counter per block, incremented on reference
- Evictions choose lowest count
  - Logic not trivial (a^2 comparison/sort)
- Storage overhead
  - 1 bit per block: same as NRU
  - How many bits are helpful?

Pitfall: Cache Filtering Effect

- Upper level caches (L1, L2) hide reference stream from lower level caches
- Blocks with “no reuse” @ LLC could be very hot (never evicted from L1/L2)
- Evicting from LLC often causes L1/L2 eviction (due to inclusion)
- Could hurt performance even if LLC miss rate improves

Replacement Policy Summary

- Replacement policies affect *capacity* and *conflict* misses
- Policies covered:
  - Belady’s optimal replacement
  - Least-recently used (LRU)
  - Practical pseudo-LRU (tree LRU)
  - Protected LRU
    - LIP/DIP variant
    - *Set dueling* to dynamically select policy
  - Not-recently-used (NRU) or *clock* algorithm
  - Least frequently used (LFU)

Cache Replacement Championships

- First: held at ISCA 2010
  - http://www.jilp.org/jwac-1
- Second: to be held at ISCA 2017
  - http://crc2.ece.tamu.edu
- Good option for a course project
  - Focus on MP cases

Replacement Policy References

S. Bansal and D. S. Modha. “CAR: Clock with Adaptive Replacement”, In FAST, 2004.

A. Basu et al. “Scavenger: A New Last Level Cache Architecture with Global Block Priority”. In Micro-40, 2007.

L. A. Belady. A study of replacement algorithms for a virtual-storage computer. In IBM Systems journal, pages 78–101, 1966.

M. Chaudhuri. “Pseudo-LIFO: The Foundation of a New Family of Replacement Policies for Last-level Caches”. In Micro, 2009.

F. J. Corbat´o, “A paging experiment with the multics system,” In Honor of P. M. Morse, pp. 217–228, MIT Press, 1969.

A. Jaleel, et al. “Adaptive Insertion Policies for Managing Shared Caches”. In PACT, 2008.

S. Jiang and X. Zhang, “LIRS: An efficient low inter-reference recency set replacement policy to improve buffer cache performance,” in Proc. ACM SIGMETRICS Conf., 2002.

T. Johnson and D. Shasha, “2Q: A low overhead high performance buffer management replacement algorithm,” in VLDB Conf., 1994.

S. Kaxiras et al. Cache decay: exploiting generational behavior to reduce cache leakage power. In ISCA-28, 2001.

A. Lai, C. Fide, and B. Falsafi. Dead-block prediction & dead-block correlating prefetchers. In ISCA-28, 2001

D. Lee et al. “LRFU: A spectrum of policies that subsumes the least recently used and least frequently used policies,” IEEE Trans.Computers, vol. 50, no. 12, pp. 1352–1360, 2001.

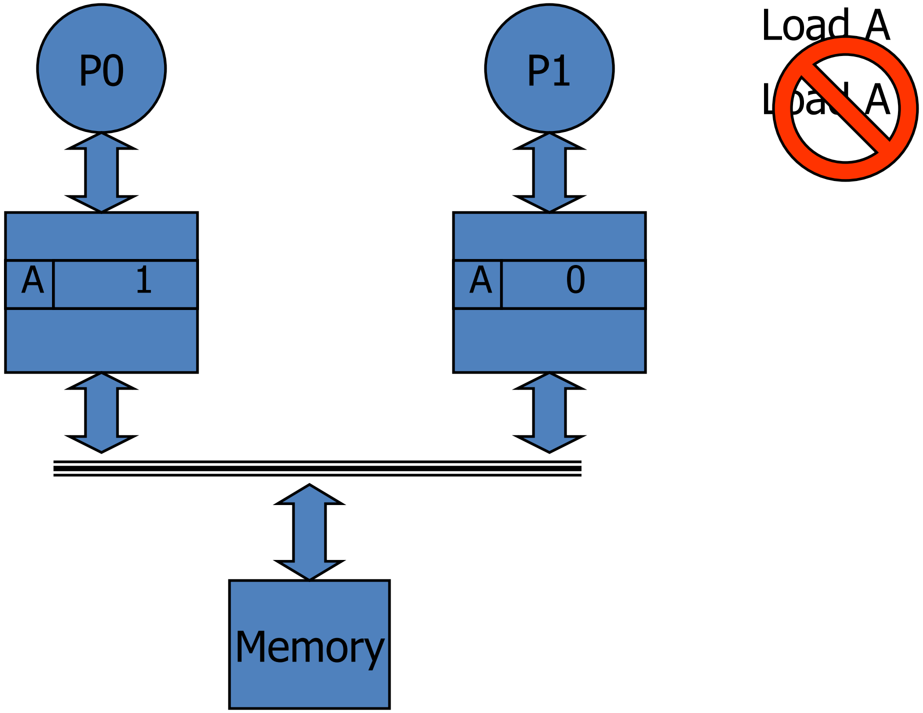
Lecture 5 Outline

- Main Memory and Cache Review
- Caches and Replacement Policies
- Cache Coherence
  - Coherence States
  - Snoopy bus-based Invalidate Protocols
  - Invalidate protocol optimizations
  - Update Protocols (Dragon/Firefly)
  - Directory protocols
  - Implementation issues

Cache Coherence Problem

Load A

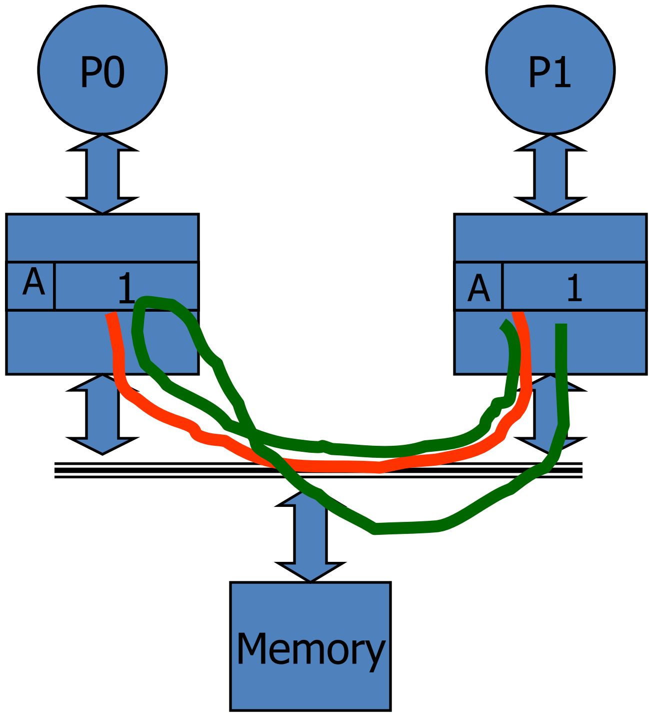
Store A<= 1



Cache Coherence Problem

Load A

Store A<= 1



Load A

Load A

Possible Causes of Incoherence

- Sharing of writeable data
  - Cause most commonly considered
- Process migration
  - Can occur even if independent jobs are executing
- I/O
  - Often fixed via O/S cache flushes

Cache Coherence

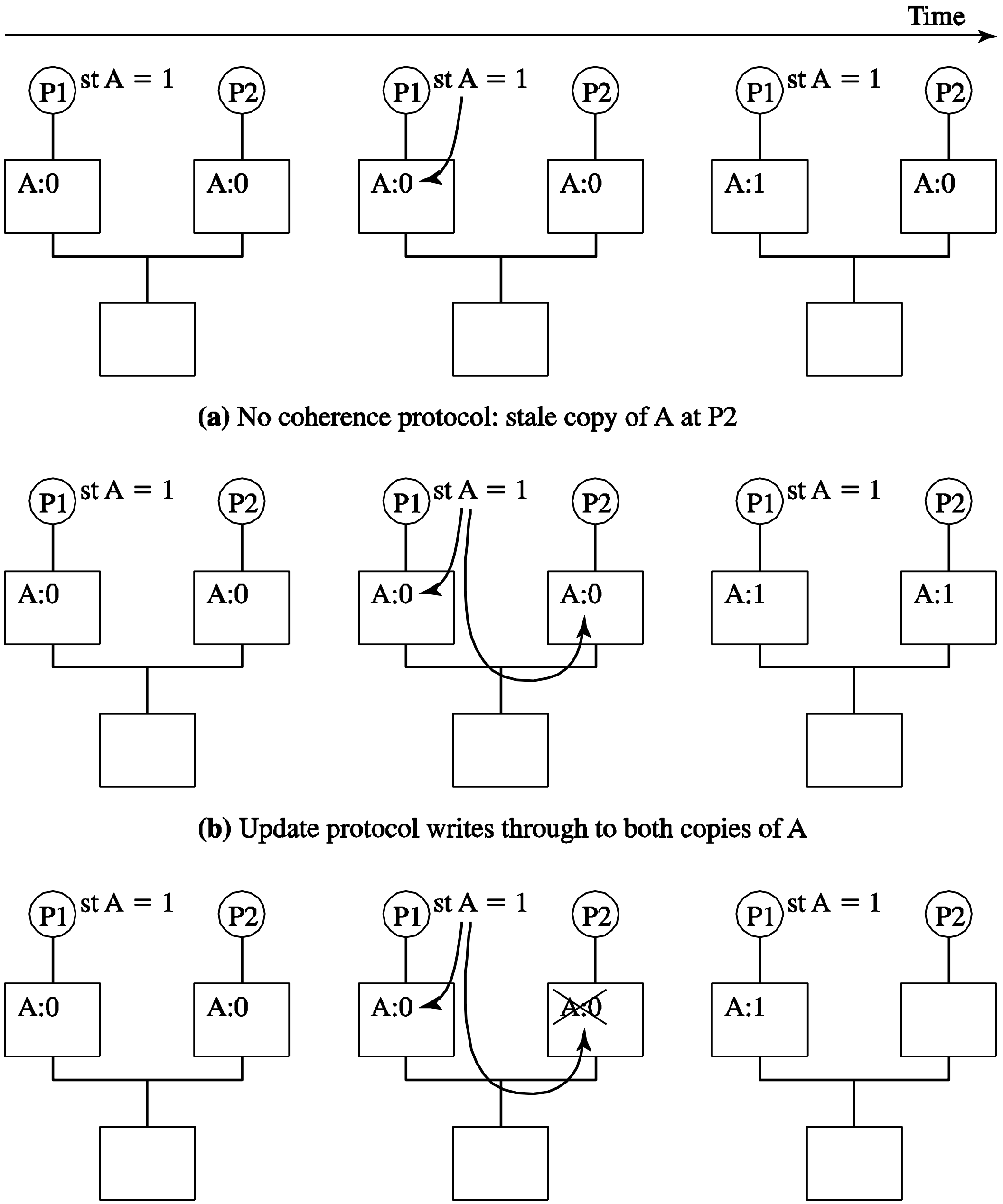
- Informally, with coherent caches: accesses to a memory location *appear* to occur simultaneously in all copies of the memory location

  “copies” ⇒ caches

- Cache coherence suggests an absolute time scale -- this is not necessary
  - What is required is the "appearance" of coherence... not absolute coherence
  - E.g. temporary incoherence between memory and a write-back cache may be OK.

Update vs. Invalidation Protocols

- Coherent Shared Memory
  - All processors see the effects of others' writes
- How/when writes are propagated
  - Determine by coherence protocol



(a) No coherence protocol: stale copy of A at P2

(b) Update protocol writes through to both copies of A

(c) Invalidate protocol eliminates stale remote copy

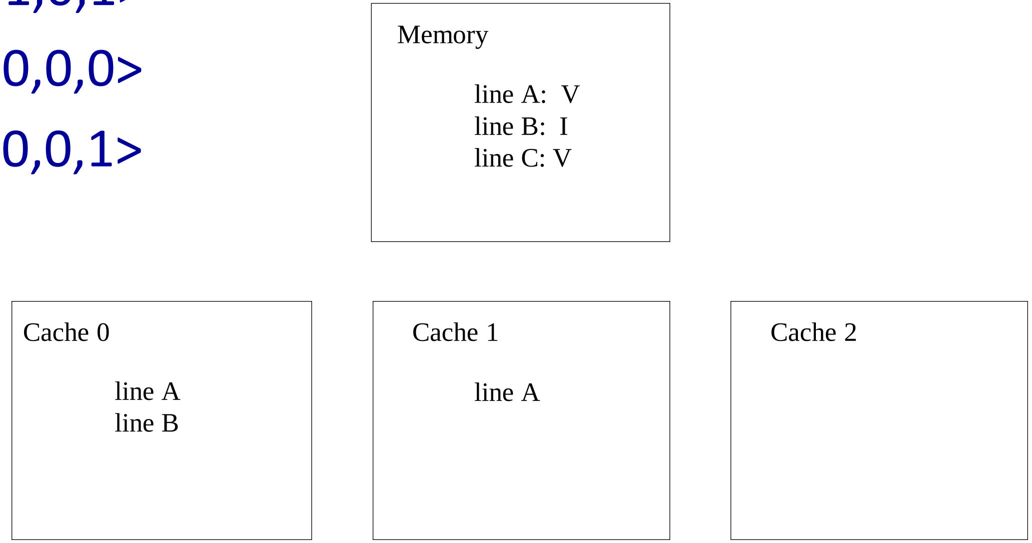
Global Coherence States

- A memory line can be present (valid) in any of the caches and/or memory
- Represent global state with an N+1 element vector
  - First N components => cache states (valid/invalid)
  - N+1st component => memory state (valid/invalid)
- Example:

  Line A: <1,1,0,1>

  Line B: <1,0,0,0>

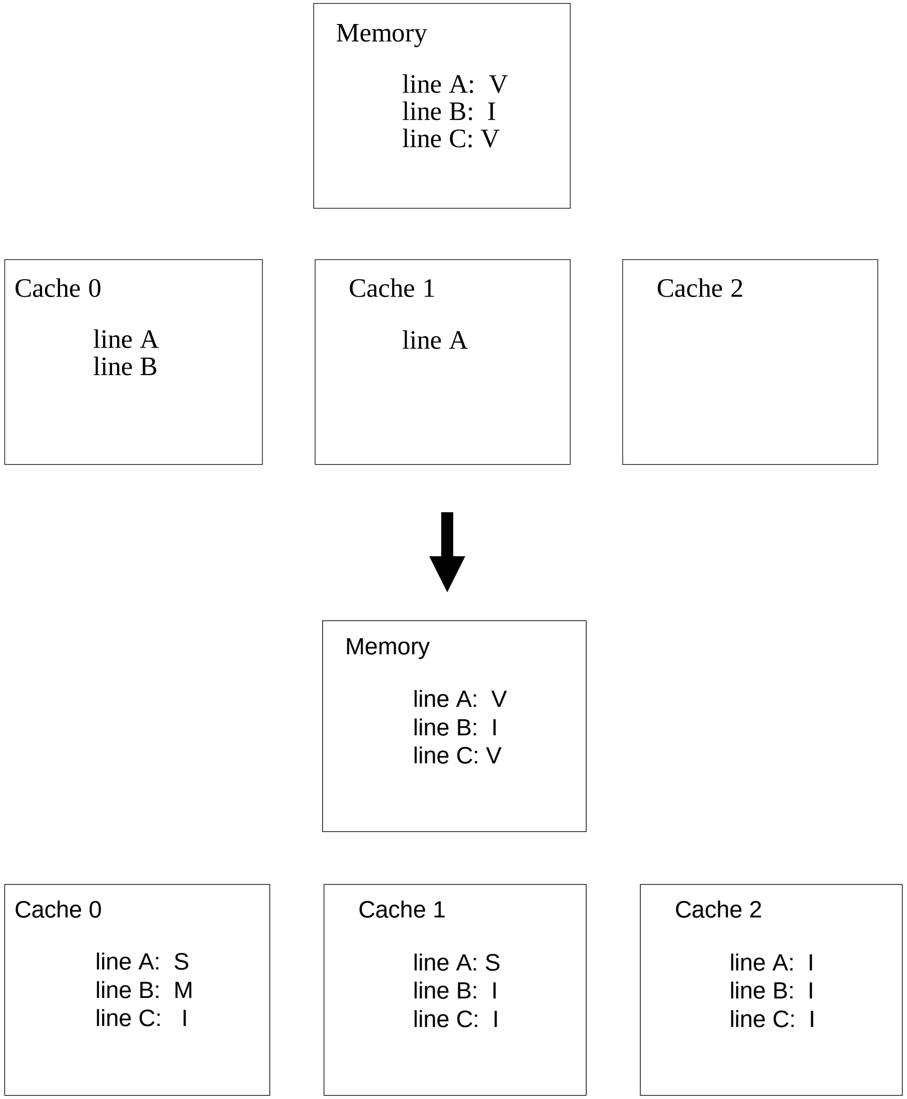
  Line C: <0,0,0,1>



Local Coherence States

- Individual caches can maintain a summary of the state of memory lines, from a “local” perspective
  - Reduces storage for maintaining state
  - May have only partial information
- Invalid (I): <0,X,X,X....X> -- local cache does not have a valid copy; (cache miss)
  - Don’t confuse invalid state with empty frame
- Shared (S): <1,X,X,X,…,1> -- local cache has a valid copy, main memory has a valid copy, other caches ??
- Modified(M): <1,0,0,..0,…0> -- local cache has only valid copy.
- Exclusive(E): <1,0,0,..0,…1> -- local cache has a valid copy, no other caches do, main memory has a valid copy.
- Owned(O): <1,X,X,X,….X> -- local cache has a valid copy, all other caches and memory may have a valid copy.
  - Only one cache can be in O state
  - <1,X,1,X,… 0> is included in O, but not included in any of the others.

Example



Memory

line A: V
line B: I
line C: V

Cache 0

line A
line B

Cache 1

line A

Cache 2

Memory

line A: V
line B: I
line C: V

Cache 0

line A: S
line B: M
line C: I

Cache 1

line A: S
line B: I
line C: I

Cache 2

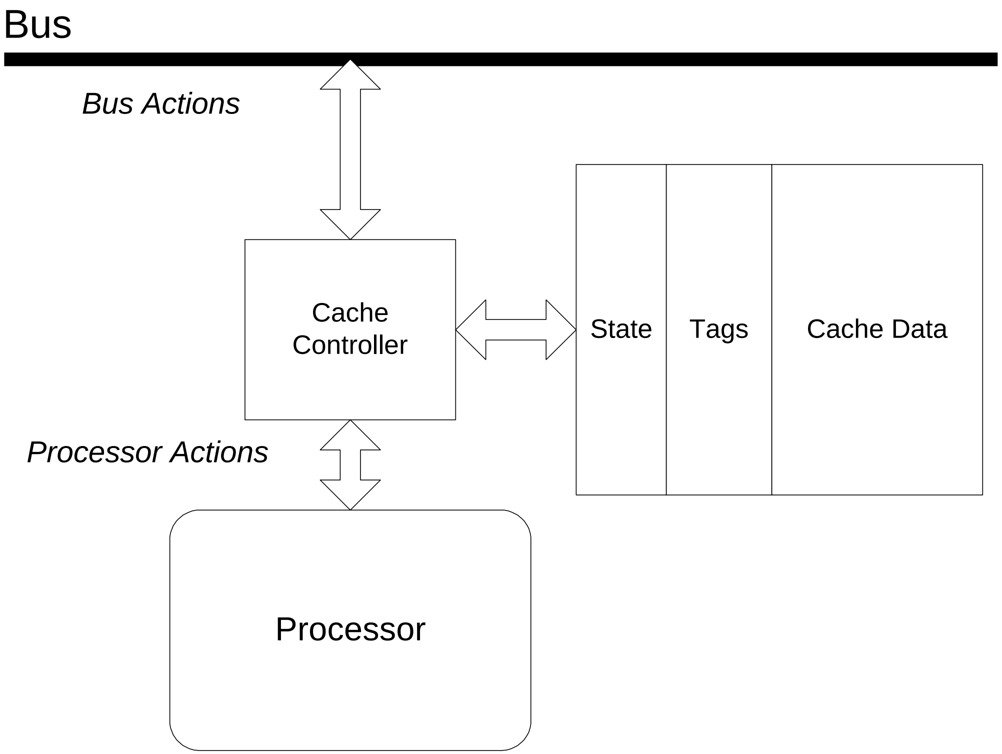
line A: I
line B: I
line C: I

Snoopy Cache Coherence

- All requests broadcast on bus
- All processors and memory snoop and respond
- Cache blocks writeable at one processor or read-only at several
  - Single-writer protocol
- Snoops that hit dirty lines?
  - Flush modified data out of cache
  - Either write back to memory, then satisfy remote miss from memory, or
  - Provide dirty data directly to requestor
  - Big problem in MP systems
    - Dirty/coherence/sharing misses

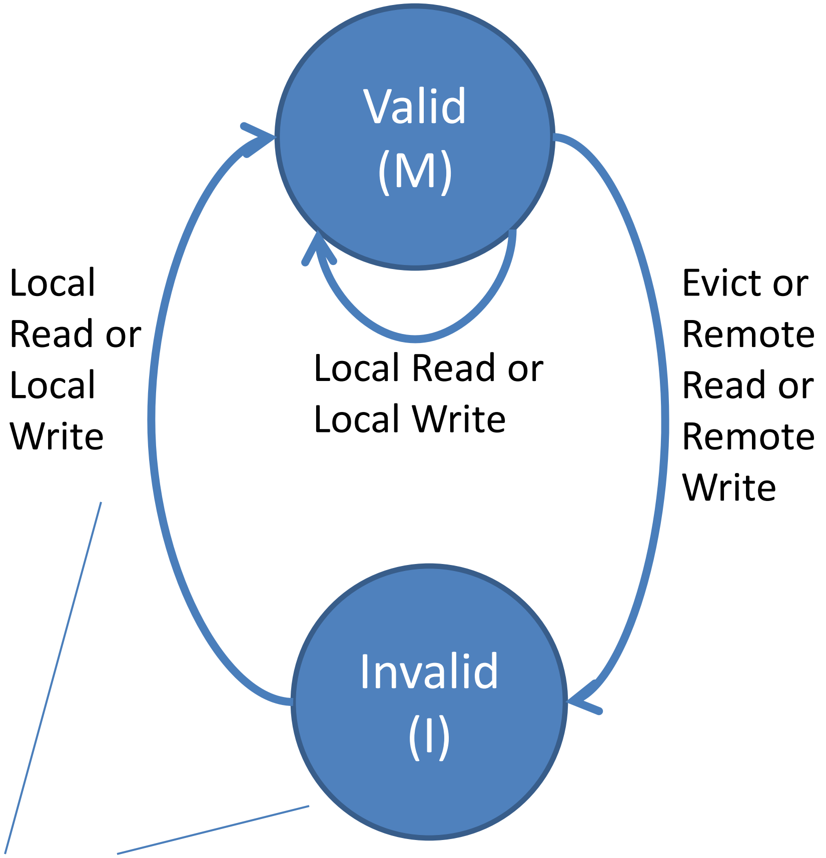
Bus-Based Protocols

- Protocol consists of states and actions (state transitions)
- Actions can be invoked from processor or bus



Minimal Coherence Protocol

- Blocks are always private or exclusive
- State transitions:
  - Local read: I->M, fetch, invalidate other copies
  - Local write: I->M, fetch, invalidate other copies
  - Evict: M->I, write back data
  - Remote read: M->I, write back data
  - Remote write: M->I, write back data


Valid (M)
Local Read or Local Write
Local Read or Local Write
Evict or Remote Read or Remote Write
Invalid (I)

Cache

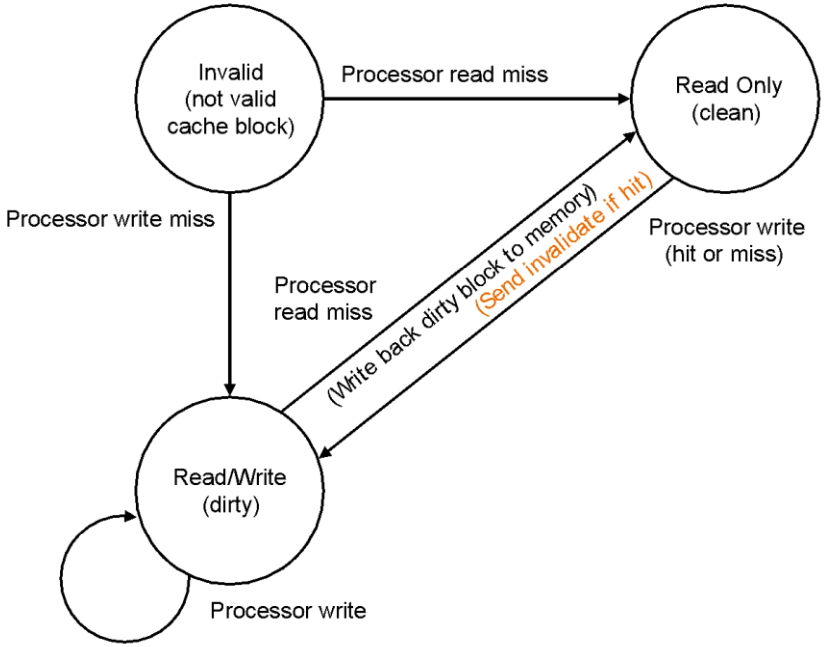
Tag	State	Data
A	M	…
B	I	…

Invalidate Protocol Optimization

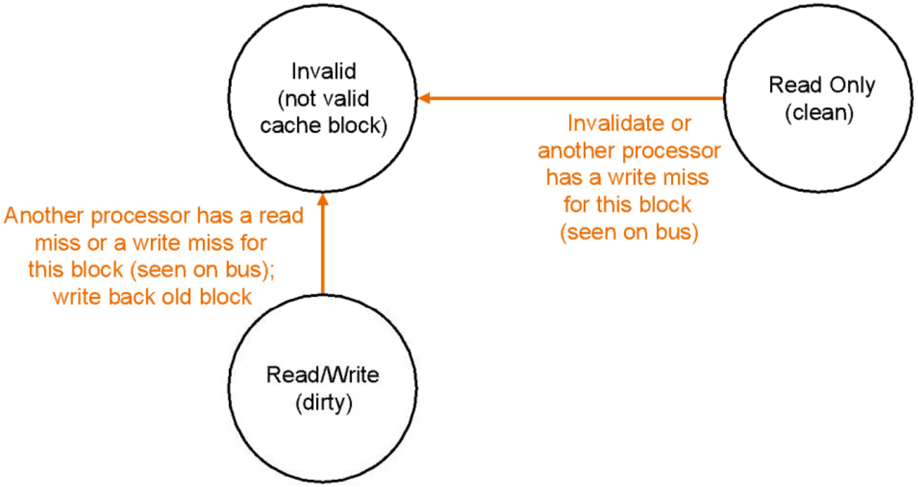
- Observation: data often read shared by multiple CPUs
  - Add S (shared) state to protocol: MSI
- State transitions:
  - Local read: I->S, fetch shared
  - Local write: I->M, fetch modified; S->M, invalidate other copies
  - Remote read: M->I, write back data
  - Remote write: M->I, write back data

Simple Coherence Protocol FSM

[Source: Patterson/Hennessy, Comp. Org. & Design]



a. Cache state transitions using signals from the processor


Invalid (not valid cache block)
Read Only (clean)
Invalidate or another processor has a write miss for this block (seen on bus)
Another processor has a read miss or a write miss for this block (seen on bus); write back old block
Read/Write (dirty)

b. Cache state transitions using signals from the bus

MSI Protocol

	Action and Next State						
Current State	***Processor Read***	***Processor Write***	***Eviction***		***Cache Read***	***Cache Read&M***	***Cache Upgrade***
I	***Cache Read*** **Acquire Copy** **→ S**	***Cache Read&M*** **Acquire Copy** **→ M**			**No Action** **→ I**	**No Action** **→ I**	**No Action** **→ I**
S	**No Action** **→ S**	***Cache Upgrade*** **→ M**	**No Action** **→ I**		**No Action** **→ S**	***Invalidate Frame*** **→ I**	***Invalidate Frame*** **→ I**
M	**No Action** **→ M**	**No Action** **→ M**	***Cache Write back*** **→ I**		**Memory inhibit; Supply data;** **→ S**	***Invalidate Frame;*** **Memory inhibit; Supply data;** **→ I**	

MSI Example

Thread Event	Bus Action	Data From	Global State	Local States:		
				C0	C1	C2
0. Initially:			<0,0,0,1>	I	I	I
1. T0 read→	CR	Memory	<1,0,0,1>	S	I	I
2. T0 write→	CU		<1,0,0,0>	M	I	I
3. T2 read→	CR	C0	<1,0,1,1>	S	I	S
4. T1 write→	CRM	Memory	<0,1,0,0>	I	M	I

- If line is in no cache
  - Read, modify, Write requires 2 bus transactions
  - Optimization: add **E**xclusive state

Invalidate Protocol Optimizations

- Observation: data can be write-private (e.g. stack frame)
  - Avoid invalidate messages in that case
  - Add E (exclusive) state to protocol: MESI
- State transitions:
  - Local read: I->E if only copy, I->S if other copies exist
  - Local write: E->M silently, S->M, invalidate other copies

MESI Protocol

- Variation used in many Intel processors
- 4-State Protocol
  - **M**odified: <1,0,0…0>
  - **E**xclusive: <1,0,0,…,1>
  - **S**hared: <1,X,X,…,1>
  - **I**nvalid: <0,X,X,…X>
- Bus/Processor Actions
  - Same as MSI
- Adds *shared* signal to indicate if other caches have a copy

MESI Protocol

	Action and Next State						
Current State	***Processor Read***	***Processor Write***	***Eviction***		***Cache Read***	***Cache Read&M***	***Cache Upgrade***
I	*Cache Read* If no sharers: → E If sharers: → S	*Cache Read&M* → M			No Action → I	No Action → I	No Action → I
S	No Action → S	*Cache Upgrade* → M	No Action → I		Respond Shared: → S	No Action → I	No Action → I
E	No Action → E	No Action → M	No Action → I		Respond Shared; → S	No Action → I	

M	No Action
	→ M
		No Action
		→ M
			Cache Write-back
			→ I
					Respond dirty;
					Write back data;
					→ S
						Respond dirty;
						Write back data;
						→ I	

MESI Example

Thread Event	Bus Action	Data From	Global State	Local States: C0	C1	C2
0. Initially:			**<0,0,0,1>**	**I**	**I**	**I**
1. T0 read→	**CR**	**Memory**	**<1,0,0,1>**	**E**	**I**	**I**
2. T0 write→	**none**		**<1,0,0,0>**	**M**	**I**	**I**

Cache-to-cache Transfers

- Common in many workloads:
  - T0 writes to a block: <1,0,…,0> (block in M state in T0)
  - T1 reads from block: T0 must write back, then T1 reads from memory

- In shared-bus system
  - T1 can *snarf* data from the bus during the writeback
  - Called *cache-to-cache transfer* or *dirty miss* or *intervention*

- Without shared bus
  - Must explicitly send data to requestor and to memory (for writeback)

- Known as the 4th C (cold, capacity, conflict, communication)

MESI Example 2

Thread Event	Bus Action	Data From	Global State	Local States:		
				C0	C1	C2
0. Initially:			<0,0,0,1>	I	I	I
1. T0 read→	CR	Memory	<1,0,0,1>	E	I	I
2. T0 write→	none		<1,0,0,0>	M	I	I
3. T1 read→	CR	C0	<1,1,0,1>	S	S	I
4. T2 read→	CR	Memory	<1,1,1,1>	S	S	S

MOESI Optimization

- Observation: shared ownership prevents cache-to-cache transfer, causes unnecessary memory read
  - Add O (owner) state to protocol: MOSI/MOESI
  - Last requestor (or last writer) becomes the owner
  - Avoid writeback (to memory) of dirty data
  - Also called *shared-dirty* state, since memory is stale

MOESI Protocol

- Used in AMD Opteron
- 5-State Protocol
  - **M**odified: <1,0,0…0>
  - **E**xclusive: <1,0,0,…,1>
  - **S**hared: <1,X,X,…,1>
  - **I**nvalid: <0,X,X,…X>
  - **O**wned: <1,X,X,X,0> ; only one owner, memory not up to date
- Owner can supply data, so memory does not have to
  - Avoids lengthy memory access

MOESI Protocol

	Action and Next State						
Current State	***Processor Read***	***Processor Write***	***Eviction***		***Cache Read***	***Cache Read&M***	***Cache Upgrade***
I	***Cache Read*** **If no sharers:** **→ E** **If sharers:** **→ S**	***Cache Read&M*** **→ M**			**No Action** **→ I**	**No Action** **→ I**	**No Action** **→ I**
S	**No Action** **→ S**	***Cache Upgrade*** **→ M**	**No Action** **→ I**		**Respond shared;** **→ S**	**No Action** **→ I**	**No Action** **→ I**

E	**No Action**
	→ E
		No Action
		→ M
			No Action
			→ I
					Respond shared;
					Supply data;
					→ S
						Respond shared;
						Supply data;
						→ I	

O	**No Action**
	→ O
		Cache Upgrade
		→ M
			Cache Write-back
			→ I
					Respond shared;
					Supply data;
					→ O
						Respond shared;
						Supply data;
						→ I	

M	**No Action**
	→ M
		No Action
		→ M
			Cache Write-back
			→ I
					Respond shared;
					Supply data;
					→ O
						Respond shared;
						Supply data;
						→ I	

MOESI Example

Thread Event	Bus Action	Data From	Global State	local states		
				C0	C1	C2
0. Initially:			<0,0,0,1>	I	I	I
1. T0 read→	CR	Memory	<1,0,0,1>	E	I	I
2. T0 write→	none		<1,0,0,0>	M	I	I
3. T2 read→	CR	C0	<1,0,1,0>	O	I	S
4. T1 write→	CRM	C0	<0,1,0,0>	I	M	I

Further Optimizations

- Observation: Shared blocks should only be fetched from memory once
  - If I find a shared block on chip, forward the block
  - Problem: multiple shared blocks possible, who forwards?
    - Everyone? Power/bandwidth wasted
  - Single forwarder, but who?
    - Last one to receive block: F state
    - I->F for requestor, F->S for forwarder
  - What if F block is evicted?
    - Favor F blocks in replacement?
    - Don’t allow silent eviction (force some other node to be F)
    - Fall back on memory copy if can’t find F copy
- IBM protocols do something very similar
- Intel has also adopted F state in recent designs (QPI)

Further Optimizations

- Observation: migratory data often “flies by”
  - Add T (transition) state to protocol
  - Tag is still valid, data isn’t
  - Data can be snarfed as it flies by
  - Only works with certain kinds of interconnect networks
  - Replacement policy issues
- Many other optimizations are possible
  - Literature extends 25 years back
  - Many unpublished (but implemented) techniques as well

Update Protocols

- Basic idea:
  - All writes (updates) are made visible to all caches:
    - (address,value) tuples sent “everywhere”
    - Similar to write-through protocol for uniprocessor caches
  - Obviously not scalable beyond a few processors
  - No one actually builds machines this way
- Simple optimization
  - Send updates to memory/directory
  - Directory propagates updates to all known copies: less bandwidth
- Further optimizations: combine & delay
  - Write-combining of adjacent updates (if consistency model allows)
  - Send write-combined data
  - Delay sending write-combined data until requested
- Logical end result
  - Writes are combined into larger units, updates are delayed until needed
  - Effectively the same as invalidate protocol
- Of historical interest only (Firefly and Dragon protocols)

Update vs Invalidate

- [Weber & Gupta, ASPLOS3]
  - Consider sharing patterns
- No Sharing
  - Independent threads
  - Coherence due to thread migration
  - Update protocol performs many wasteful updates
- Read-Only
  - No significant coherence issues; most protocols work well
- Migratory Objects
  - Manipulated by one processor at a time
  - Often protected by a lock
  - Usually a write causes only a single invalidation
  - E state useful for Read-modify-Write patterns
  - Update protocol could proliferate copies

Update vs Invalidate, contd.

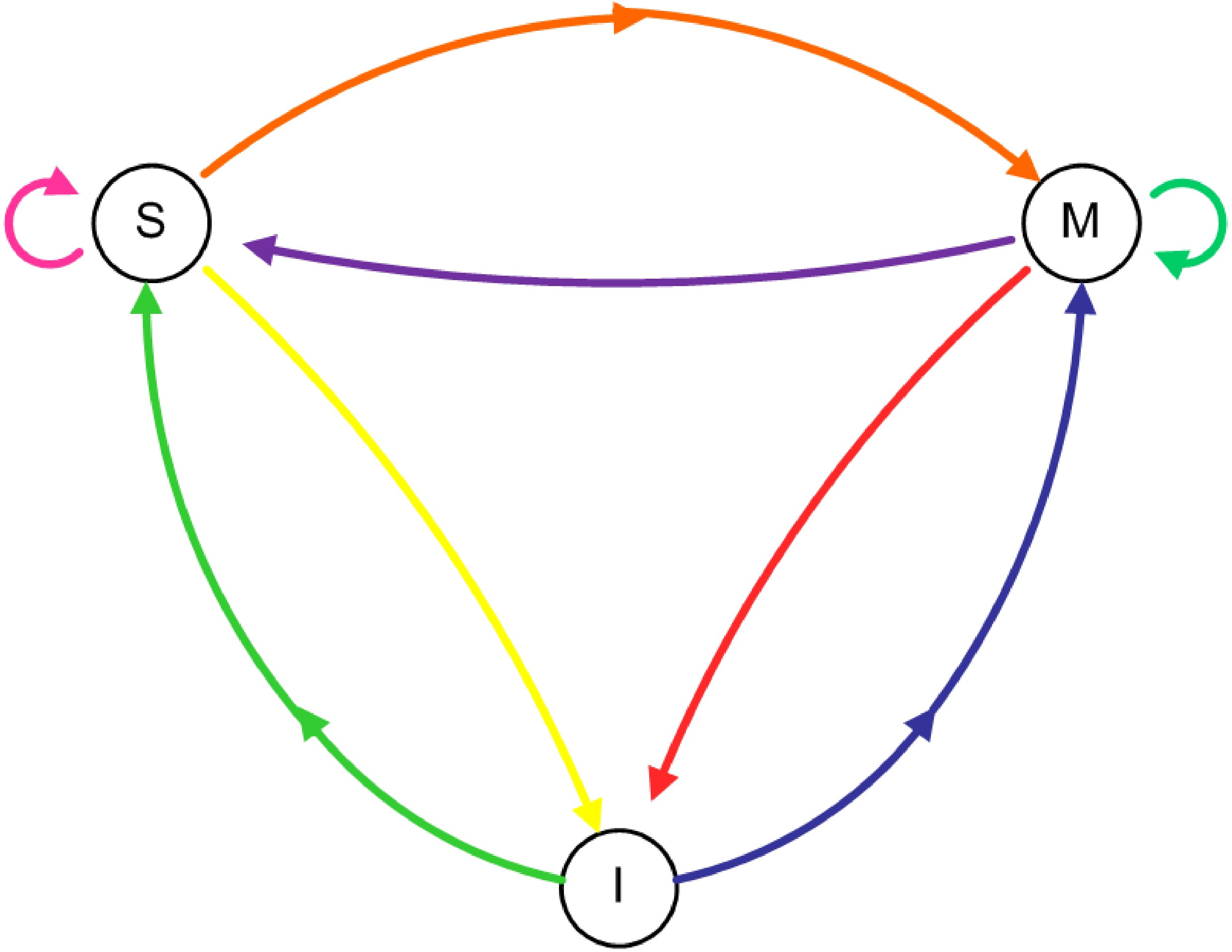
- Synchronization Objects
  - Locks
  - Update could reduce spin traffic invalidations
  - Test&Test&Set w/ invalidate protocol would work well
- Many Readers, One Writer
  - Update protocol may work well, but writes are relatively rare
- Many Writers/Readers
  - Invalidate probably works better
  - Update will proliferate copies
- What is used today?
  - Invalidate is dominant
  - CMP *has not changed* this assessment
    - Even with plentiful on-chip bandwidth

Nasty Realities

- State diagram is for (ideal) protocol assuming instantaneous and actions
- In reality controller implements more complex diagrams
  - A protocol state transition may be started by controller when bus activity changes local state
  - Example: an upgrade pending (for bus) when an invalidate for same line arrives

Example: MSI (SGI-Origin-like, directory, invalidate)

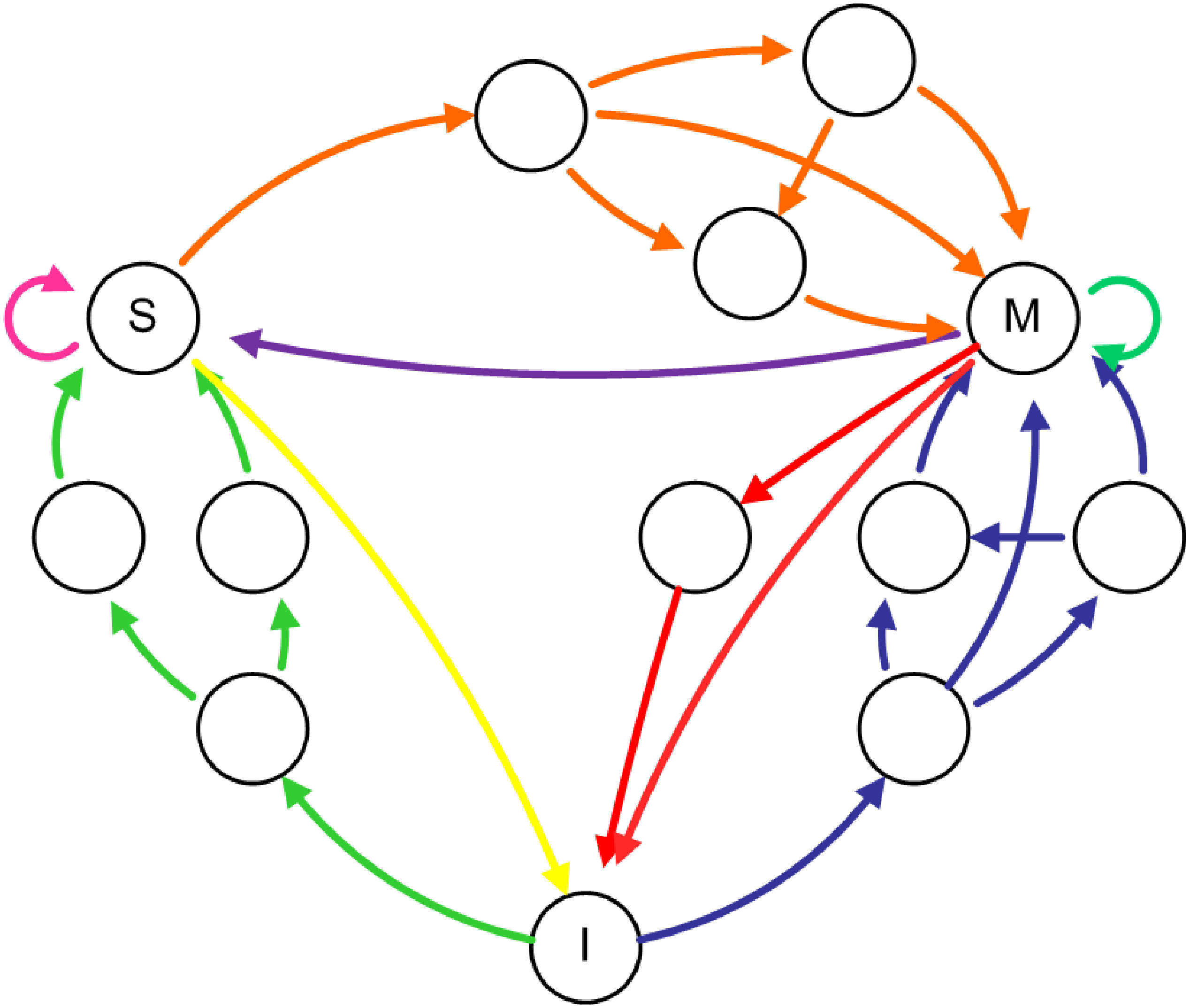
Stable States



Example: MSI (SGI-Origin-like, directory, invalidate)

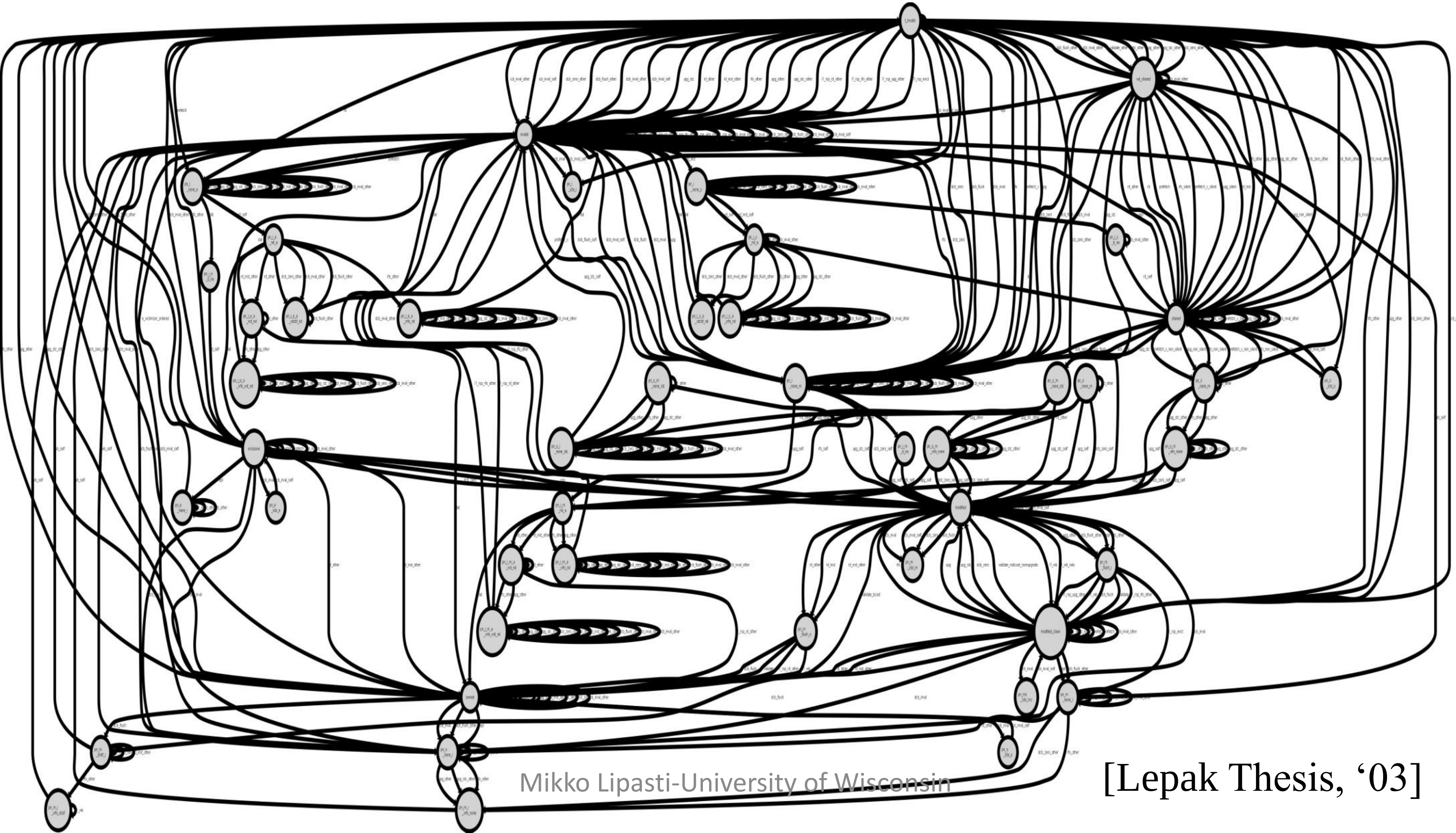
Stable States

Busy States



Cache coherence complexity

L2 MOETSI Transitions



[Lepak Thesis, ‘03]

Implementing Cache Coherence

- Snooping implementation
  - Origins in shared-memory-bus systems
  - All CPUs could observe all other CPUs requests on the bus; hence “snooping”
    - Bus Read, Bus Write, Bus Upgrade
  - React appropriately to snooped commands
    - Invalidate shared copies
    - Provide up-to-date copies of dirty lines
      - Flush (writeback) to memory, or
      - Direct intervention (*modified intervention* or *dirty miss*)
- Snooping suffers from:
  - Scalability: shared busses not practical
  - Ordering of requests without a shared bus
  - Lots of recent and on-going work on scaling snoop-based systems

Snooping Cache Coherence

- Basic idea: broadcast snoop to all caches to find owner
- Not scalable?
  - Address traffic roughly proportional to square of number of processors
  - Current implementations scale to 64/128-way (Sun/IBM) with multiple address-interleaved broadcast networks
- Inbound snoop bandwidth: big problem

$$OutboundSnoopRate = s_o = \langle CacheMissRate \rangle + \langle BusUpgradeRate \rangle$$

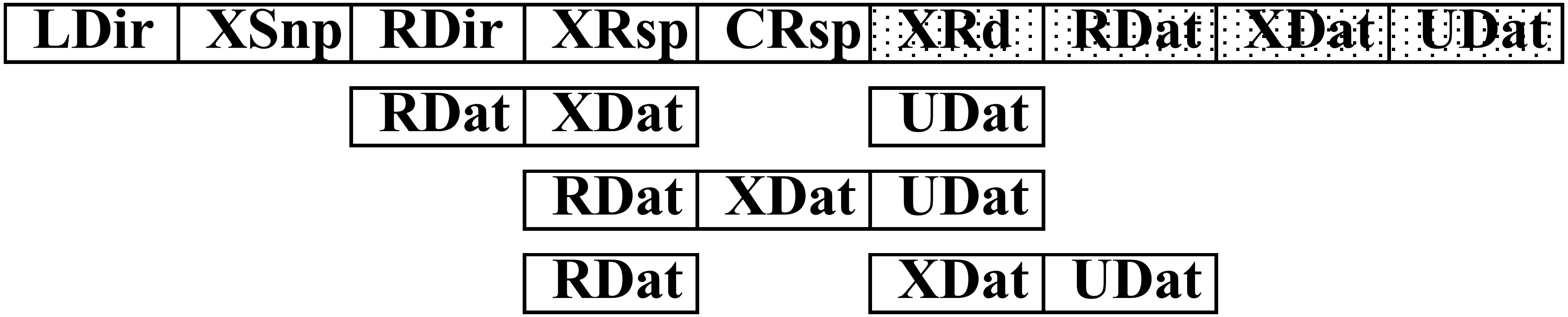
$$InboundSnoopRate = s_i = n \times s_o$$

Snoop Bandwidth

- Snoop filtering of various kinds is possible
- Filter snoops at sink: Jetty filter [Moshovos et al., HPCA 2001]
  - Check small "filter cache" that summarizes contents of local cache
  - Avoid power-hungry lookups in each tag array
- Filter snoops at source: Multicast snooping [Bilir et al., ISCA 1999]
  - Predict likely sharing set, snoop only predicted sharers
  - Double-check at directory to make sure
- Filter snoops at source: Region coherence
  - Concurrent work: [Cantin/Smith/Lipasti, ISCA 2005; Moshovos, ISCA 2005]
  - Check larger region of memory on every snoop; remember when no sharers
  - Snoop only on first reference to region, or when region is shared
  - Eliminate 60%+ of all snoops

Snoop Latency

- Snoop latency:
  - Must reach all nodes, return and combine responses
  - Topology matters: ring, mesh, torus, hypercube
  - No obvious solutions
- Parallelism: fundamental advantage of snooping
  - Broadcast exposes parallelism, enables speculative latency reduction



LDir	XSnp	RDir	XRsp	CRsp	XRd	RDat	XDat	UDat
		RDat	XDat		UDat			
			RDat	XDat	UDat			
			RDat		XDat	UDat		

Scaleable Cache Coherence

- No physical bus but still snoop
  - Point-to-point tree structure (indirect) or ring
  - Root of tree or ring provide ordering point
  - Use some scalable network for data (ordering less important)
- Or, use level of indirection through directory
  - Directory at memory remembers:
    - Which processor is “single writer”
    - Which processors are “shared readers”
  - Level of indirection has a price
    - Dirty misses require 3 hops instead of two
      - Snoop: Requestor->Owner->Requestor
      - Directory: Requestor->Directory->Owner->Requestor

Implementing Cache Coherence

- Directory implementation
  - Extra bits stored in memory (directory) record state of line
  - Memory controller maintains coherence based on the current state
  - Other CPUs' commands are not snooped, instead:
    - Directory forwards relevant commands
  - Powerful filtering effect: only observe commands that you need to observe
  - Meanwhile, bandwidth at directory scales by adding memory controllers as you increase size of the system
    - Leads to very scalable designs (100s to 1000s of CPUs)
- Directory shortcomings
  - Indirection through directory has latency penalty
  - If shared line is dirty in other CPU's cache, directory must forward request, adding latency
  - This can severely impact performance of applications with heavy sharing (e.g. relational databases)

Directory Protocol Implementation

- Basic idea: Centralized directory keeps track of data location(s)
- Scalable
  - Address traffic roughly proportional to number of processors
  - Directory & traffic can be distributed with memory banks (interleaved)
  - Directory cost (SRAM) or latency (DRAM) can be prohibitive
- Presence bits track sharers
  - Full map (N processors, N bits): cost/scalability
  - Limited map (limits number of sharers)
  - Coarse map (identifies board/node/cluster; must use broadcast)
- Vectors track sharers
  - Point to shared copies
  - Fixed number, linked lists (SCI), caches chained together
  - Latency vs. cost vs. scalability

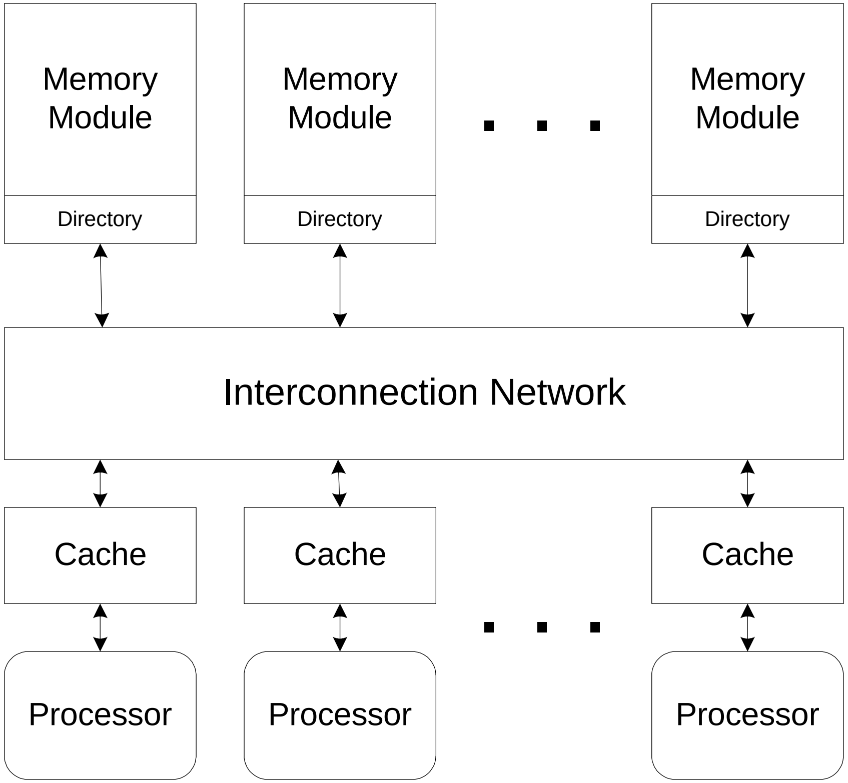
Directory Protocol Latency

LDir	**XSnp**	**RDir**	**XRd**	**RDat**	**XDat**	**UDat**

- Access to non-shared data
  - Overlap directory read with data read
  - Best possible latency given distributed memory
- Access to shared data
  - Dirty miss, modified intervention
  - Shared intervention?
    - If DRAM directory, no gain
    - If directory cache, possible gain (use F state)
  - No inherent parallelism
  - Indirection adds latency
  - Minimum 3 hops, often 4 hops

Directory-based Cache Coherence

- An alternative for large, scalable MPs
- Can be based on any of the protocols discussed thus far
  - We will use MSI
- Memory Controller becomes an active participant
- Sharing info held in memory directory
  - Directory may be distributed
- Use point-to-point messages
- Network is not totally ordered

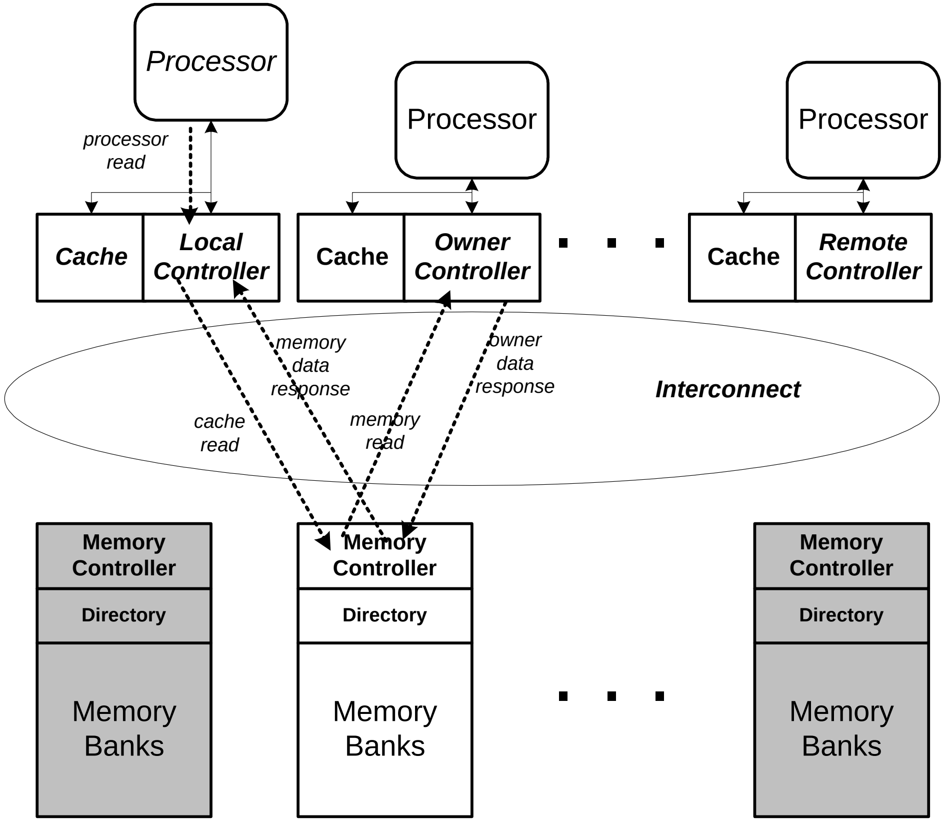


Example: Simple Directory Protocol

- Local cache controller states
  - M, S, I as before
- Local directory states
  - **S**hared: <1,X,X,…1>; one or more proc. has copy; memory is up-to-date
  - **M**odified: <0,1,0,….,0> one processor has copy; memory does not have a valid copy
  - **U**ncached: <0,0,…0,1> none of the processors has a valid copy
- Directory also keeps track of sharers
  - Can keep global state vector in full
  - e.g. via a bit vector

Example

- *Local cache* suffers load miss
- Line in *remote cache* in M state
  - It is the *owner*
- Four messages send over network
  - Cache read from local controller to home memory controller
  - Memory read to remote cache controller
  - Owner data back to memory controller; change state to S
  - Memory data back to local cache; change state to S



(c) 2007 Jim Smith

Cache Controller State Table

	Cache Controller Actions and Next States								
	from Processor Side				**from Memory Side**				
Current State	***Processor Read***	***Processor Write***	***Eviction***		***Memory Read***	***Memory Read&M***	***Memory Invalidate***	***Memory Upgrade***	***Memory Data***
I	***Cache Read*** → I'	***Cache Read&M*** → I''					**No Action** → I		
S	**No Action** → S	***Cache Upgrade*** → S'	**No Action*** → I				**Invalidate Frame;** ***Cache ACK*;** → I		
M	**No Action** → M	**No Action** → M	***Cache Write-back*** → I		***Owner Data;*** → S	***Owner Data;*** → I	**Invalidate Frame;** ***Cache ACK*;** → I		
I'									**Fill Cache** → S
I''									**Fill Cache** → M
S'								**No Action** → M	

Memory Controller State Diagram

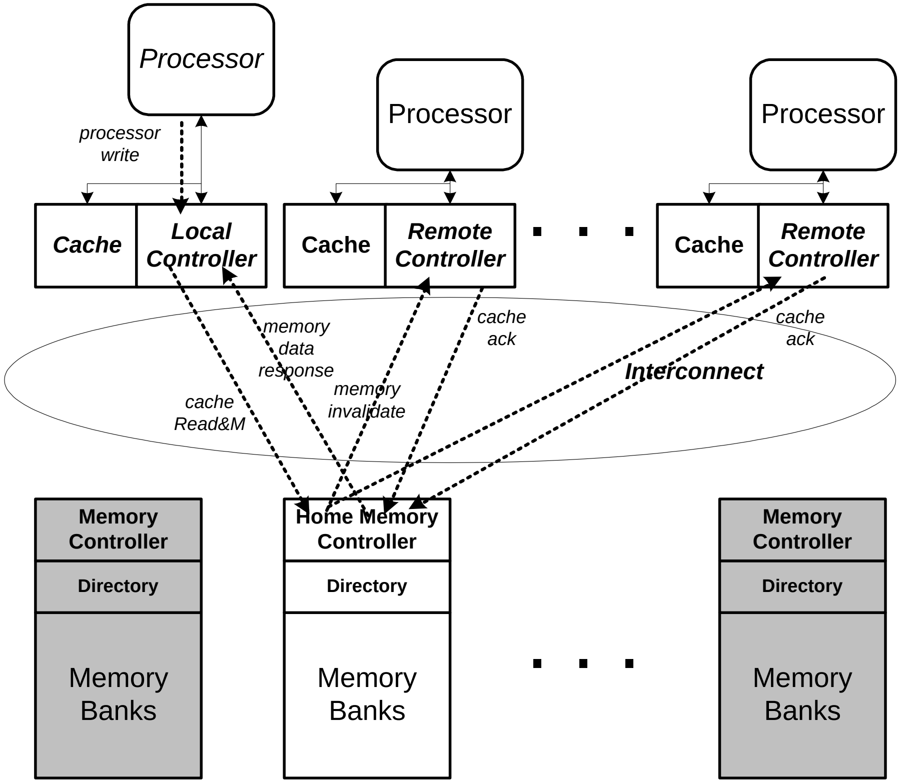
	Memory Controller Actions and Next States						
	command from Local Cache Controller				response from Remote Cache Controller		
Current Directory State	*Cache Read*	*Cache Read&M*	*Cache Upgrade*		*Data Write-back*	*Cache ACK*	*Owner Data*
U	*Memory Data*; Add Requestor to Sharers; → S	*Memory Data*; Add Requestor to Sharers; → M					
S	*Memory Data*; Add Requestor to Sharers; → S	*Memory Invalidate* All Sharers; → M'	*Memory Upgrade* All Sharers; → M''		No Action → I		
M	*Memory Read* from Owner; → S'	*Memory Read&M*; to Owner → M'			Make Sharers Empty; → U		

S'							*Memory Data*
							to Requestor;
							Write memory;
							Add Requestor to Sharers;
							→ S

M'						When all ACKS *Memory Data*; → M	*Memory Data* to Requestor; → M
M''						When all ACKS then → M	

Another Example

- Local write (miss) to shared line
- Requires invalidations and acks



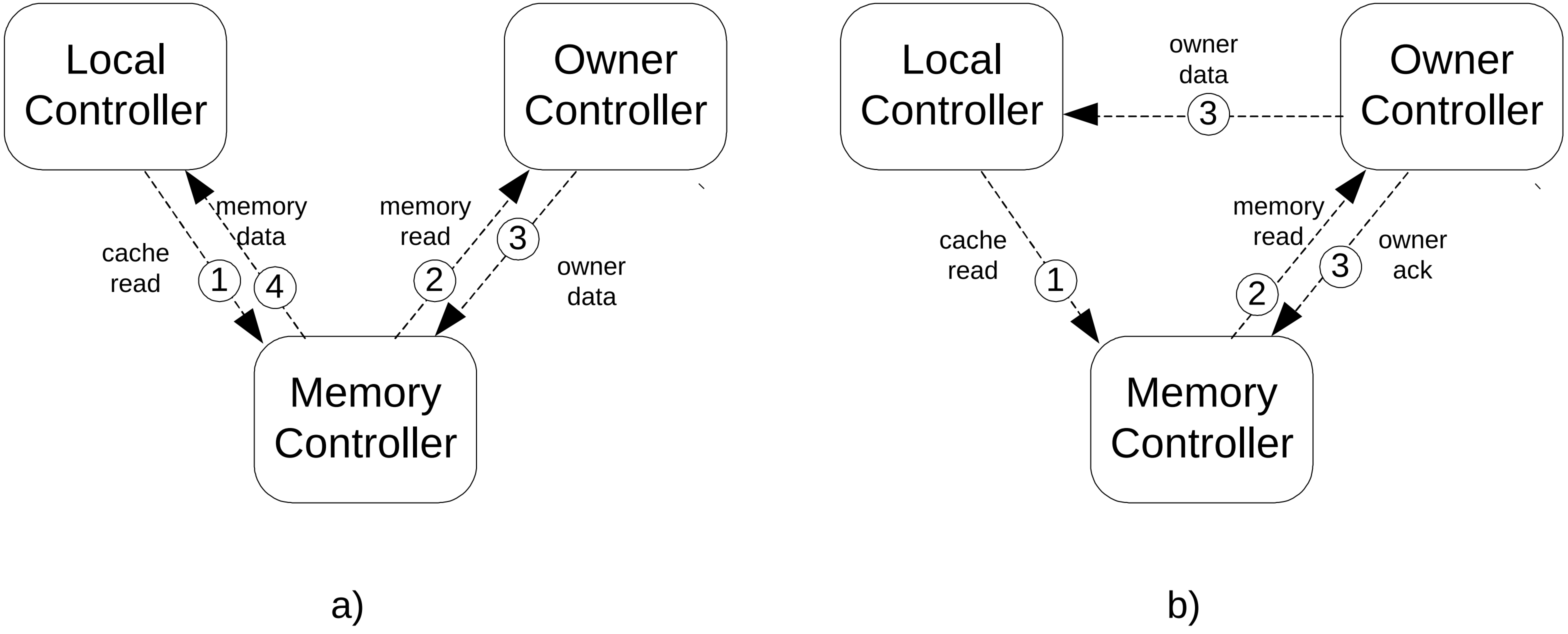
Example Sequence

- Similar to earlier sequences

Thread Event	Controller Actions	Data From	global state	local states: C0	C1	C2
0. Initially:			<0,0,0,1>	I	I	I
1. T0 read→	CR,MD	Memory	<1,0,0,1>	S	I	I
2. T0 write→	CU, MU*,MD		<1,0,0,0>	M	I	I
3. T2 read→	CR,MR,MD	C0	<1,0,1,1>	S	I	S
4. T1 write→	CRM,MI,CA,MD	Memory	<0,1,0,0>	I	M	I

Variation: Three Hop Protocol

- Have owner send data directly to local controller
- Owner Acks to Memory Controller in parallel



Directory Protocol Optimizations

- Remove dead blocks from cache:
  - Eliminate 3- or 4-hop latency
  - Dynamic Self-Invalidation [Lebeck/Wood, ISCA 1995]
  - Last touch prediction [Lai/Falsafi, ISCA 2000]
  - Dead block prediction [Lai/Fide/Falsafi, ISCA 2001]
- Predict sharers
  - Prediction in coherence protocols [Mukherjee/Hill, ISCA 1998]
  - Instruction-based prediction [Kaxiras/Goodman, ISCA 1999]
  - Sharing prediction [Lai/Falsafi, ISCA 1999]
- Hybrid snooping/directory protocols
  - Improve latency by snooping, conserve bandwidth with directory
  - Multicast snooping [Bilir et al., ISCA 1999; Martin et al., ISCA 2003]
  - Bandwidth-adaptive hybrid [Martin et al., HPCA 2002]
  - Token Coherence [Martin et al., ISCA 2003]
  - Virtual Tree Coherence [Enright Jerger MICRO 2008]

Protocol Races

- Atomic bus
  - Only stable states in protocol (e.g. M, S, I)
  - All state transitions are atomic (I->M)
  - No conflicting requests can interfere since bus is held till transaction completes
    - Distinguish coherence transaction from data transfer
    - Data transfer can still occur much later; easier to handle this case
- Atomic buses don’t scale
  - At minimum, separate bus request/response
- Large systems have broadly variable delays
  - Req/resp separated by dozens of cycles
  - Conflicting requests can and do get issued
  - Messages may get reordered in the interconnect
- How do we resolve them?

Example: MSI (SGI-Origin-like, directory, invalidate)

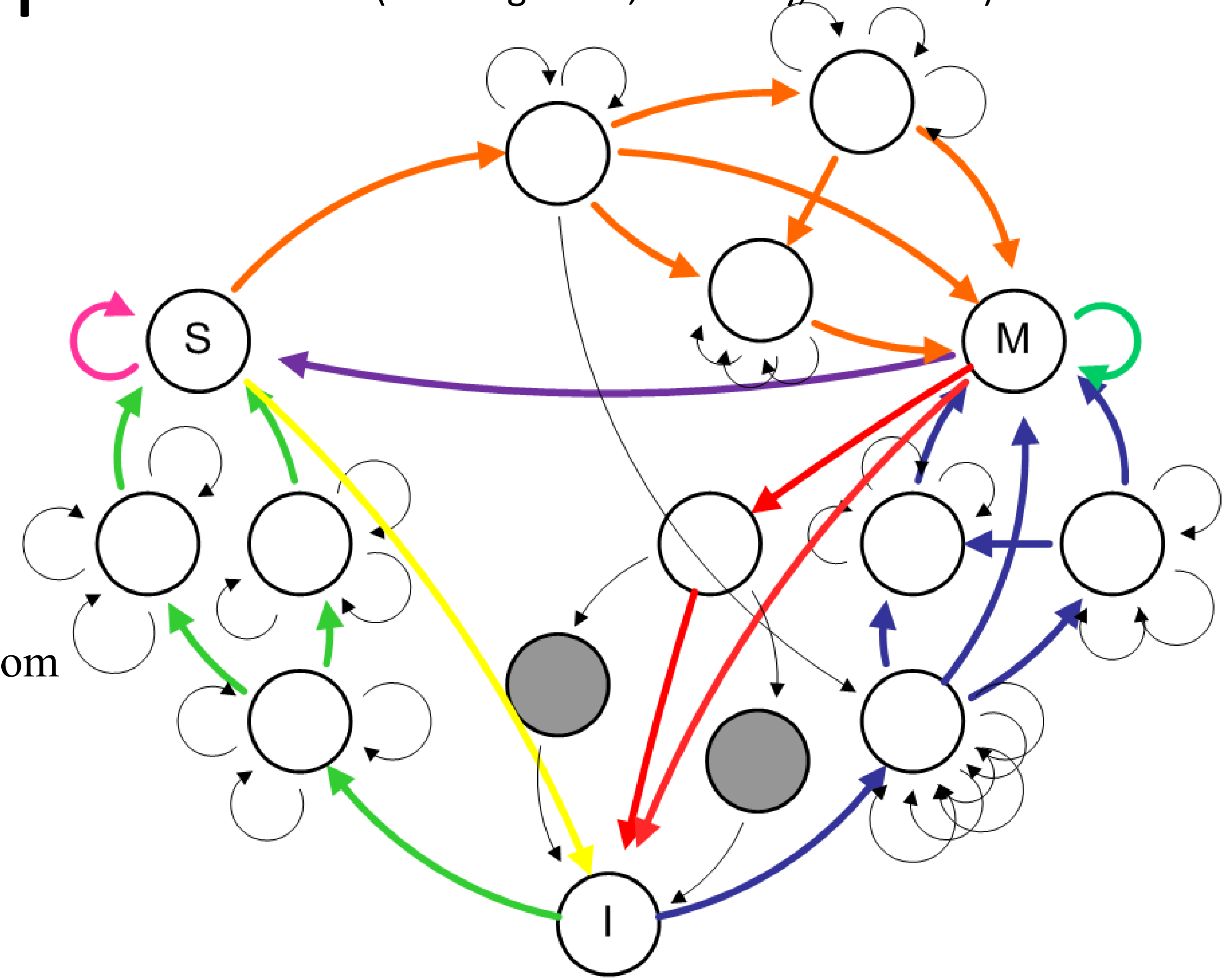
Stable States

Busy States

Races

↑

"unexpected" events from concurrent requests to same block



Resolving Protocol Races

- Req/resp decoupling introduces transient states
  - E.g. I->S is now I->ItoX->ItoS_nodata->S
- Conflicting requests to blocks in transient states
  - NAK – ugly; livelock, starvation potential
  - Keep adding more transient states …
- Directory protocol makes this a bit easier
  - Can order at directory, which has full state info
  - Even so, messages may get reordered

Common Protocol Races

- Read strings: P0 read, P1 read, P2 read
  - Easy, since read is nondestructive
  - Can rely on F state to reduce DRAM accesses
  - Forward reads to previous requestor (F)
- Write strings: P0 write, P1 write, P2 write
  - Forward P1 write req to P0 (M)
  - P0 completes write then forwards M block to P1
  - Build string of writes (write string forwarding)
- Read after write (similar to prev. WAW)
- Writeback race: P0 evicts dirty block, P1 reads
  - Dirty block is in the network (no copy at P0 or at dir)
  - NAK P1, or force P0 to keep copy till dir ACKs WB
- Many others crop up, esp. with optimizations

Lecture 5 Outline

- Main Memory and Cache Review
- Caches and Replacement Policies
- Cache Coherence
  - Coherence States
  - Snoopy bus-based Invalidate Protocols
  - Invalidate protocol optimizations
  - Update Protocols (Dragon/Firefly)
  - Directory protocols
  - Implementation issues

Additional Slides

- For reference only

Update Protocol: Dragon

- Dragon (developed at Xerox PARC)
- 5-State Protocol
  - **I**nvalid:<0,X,X,…X>
    - Some say no invalid state – due to confusion regarding empty frame versus invalid line state
  - **E**xclusive: <1,0,0,…,1>
  - **S**hared-**C**lean (Sc): <1,X,X,…X> memory may not be up-to-date
  - **S**hared-**M**odified (Sm): <1,X,X,X…0> memory not up-to-date; only one copy in Sm
  - **M**odified: <1,0,0,…0>
- Includes Cache Update action
- Includes Cache Writeback action
- Bus includes Shared flag
  - Appears to also require memory inhibit signal
  - Distinguish shared case where cache (not memory) supplies data

Dragon State Diagram

	Action and Next State					
Current State	***Processor Read***	***Processor Write***	***Eviction***		***Cache Read***	***Cache Update***
I	*Cache Read* If no sharers: → E If sharers: → Sc	*Cache Read* If no sharers: → M If sharers: *Cache Update* → Sm			→ I	→ I

Sc	No Action
	→ Sc
		Cache Update
		If no sharers:
		→M
		If sharers:
		→ Sm
			No Action
			→ I
					Respond Shared;
					→ Sc
						Respond shared;
						Update copy;
						→ Sc

E	No Action → E	No Action → M	No Action → I		Respond shared; Supply data → Sc	

Sm	No Action
	→ Sm
		Cache Update
		If no sharers:
		→M
		If sharers:
		→ Sm
			Cache Write-back
			→ I
					Respond shared;
					Supply data;
					→ Sm
						Respond shared;
						Update copy;
						→ Sc

M	No Action
	→ M
		No Action
		→ M
			Cache Write-back
			→ I
					Respond shared;
					Supply data;
					→ Sm	

Example

Thread Event	Bus Action	Data From	Global State	local states		
				C0	C1	C2
0. Initially:			<0,0,0,1>	I	I	I
1. T0 read→	CR	Memory	<1,0,0,1>	E	I	I
2. T0 write→	none		<1,0,0,0>	M	I	I
3. T2 read→	CR	C0	<1,0,1,0>	Sm	I	Sc
4. T1 write→	CR,CU	C0	<1,1,1,0>	Sc	Sm	Sc
5. T0 read→	none (hit)	C0	<1,1,1,0>	Sc	Sm	Sc

- Appears to require atomic bus cycles CR,CU on write to invalid line

Update Protocol: Firefly

- Develped at DEC by ex-Xerox people
- 5-State Protocol
  - Similar to Dragon – different state naming based on shared/exclusive and clean/dirty
  - **I**nvalid:<0,X,X,…X>
  - **EC**: <1,0,0,…,1>
  - **SC**: <1,X,X,…X> memory may not be up-to-date
  - **EM**: <1,0,0,…0>
  - **SM**: <1,X,X,X…0> memory not up-to-date; only one copy in Sm
- Performs write-through updates (different from Dragon)