Memory Data Flow

Prof. Mikko H. Lipasti
University of Wisconsin-Madison

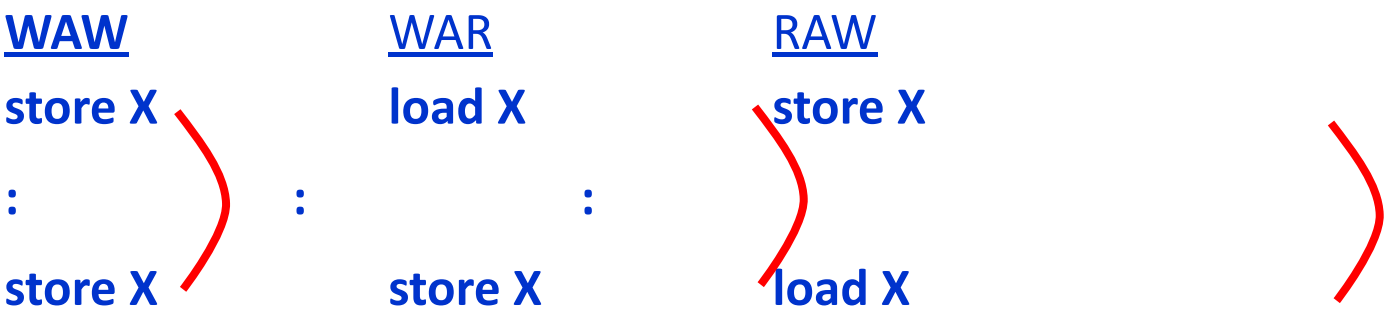
Lecture notes based on notes by John P. Shen
Updated by Mikko Lipasti

Memory Data Flow

- **Memory Data Flow**
  - Memory Data Dependences
  - Load Bypassing
  - Load Forwarding
  - Speculative Disambiguation
  - The Memory Bottleneck
- **Cache Hits and Cache Misses**

Memory Data Dependences

- **Besides branches, long memory latencies are one of the biggest performance challenges today.**
- **To preserve sequential (in-order) state in the data caches and external memory (so that recovery from exceptions is possible) stores are performed in order. This takes care of antidependences and output dependences to memory locations.**
- **However, loads can be issued out of order with respect to stores if the out-of-order loads check for data dependences with respect to previous, pending stores.**

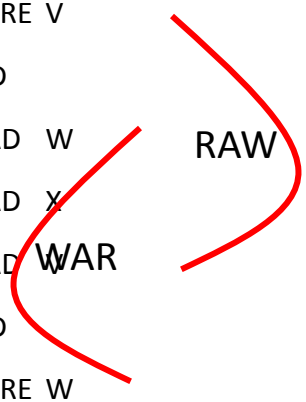


Memory Data Dependences

- **"Memory Aliasing" = Two memory references involving the same memory location (collision of two memory addresses).**
- **"Memory Disambiguation" = Determining whether two memory references will alias or not (whether there is a dependence or not).**
- **Memory Dependency Detection:**
  - Must compute effective addresses of both memory references
  - Effective addresses can depend on run-time data and other instructions
  - Comparison of addresses require much wider comparators

Example code:

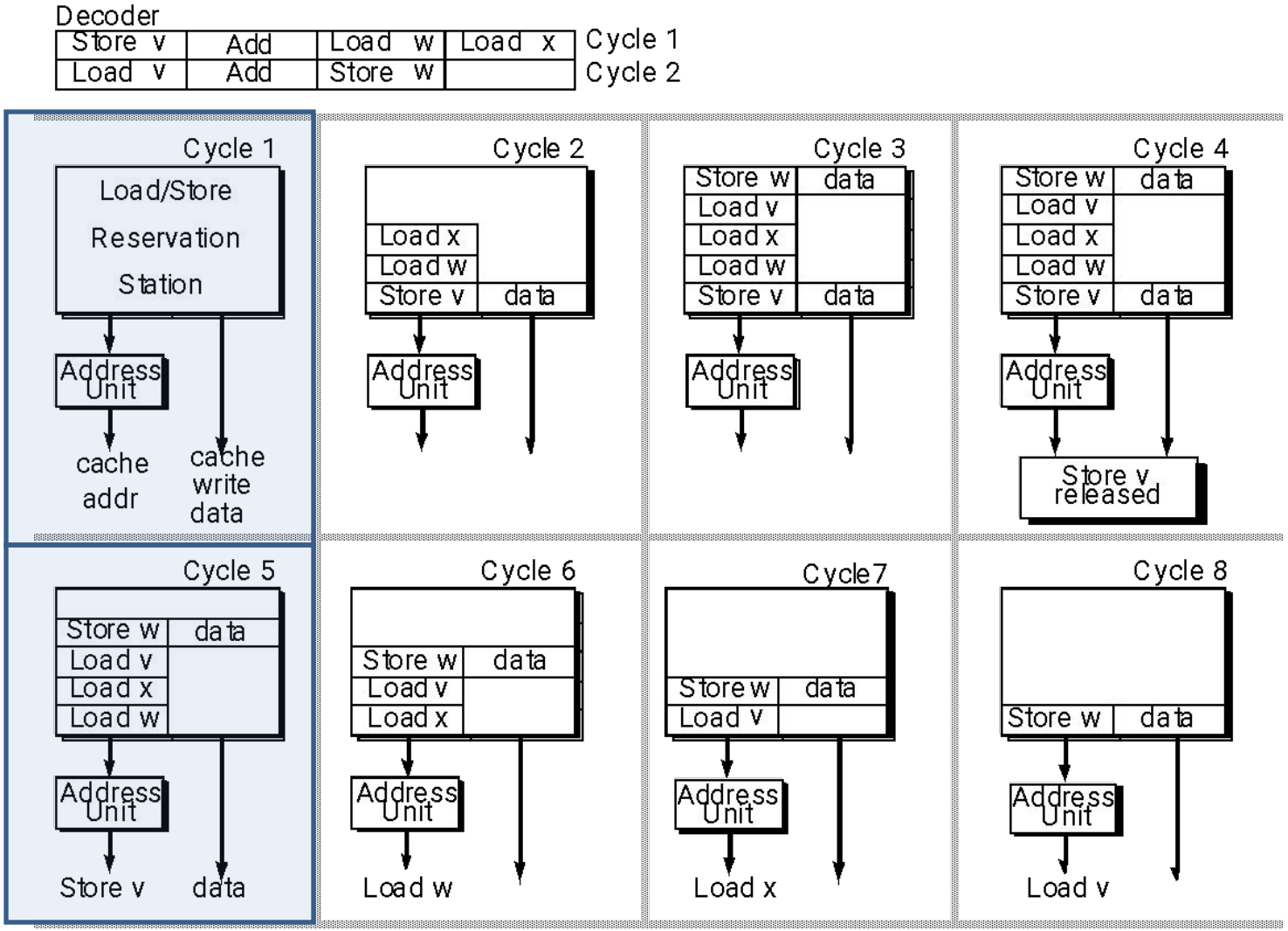
```
(1)   STORE  V
(2)   ADD
(3)   LOAD   W
(4)   LOAD   X
(5)   LOAD   V
(6)   ADD
(7)   STORE  W
```



Total Order of Loads and Stores

- **Keep all loads and stores totally in order with respect to each other.**
- **However, loads and stores can execute out of order with respect to other types of instructions.**
- **Consequently, stores are held for all previous instructions, and loads are held for stores.**
  - **I.e. stores performed at commit point**
  - **Sufficient to prevent wrong branch path stores since all prior branches now resolved**

Illustration of Total Order

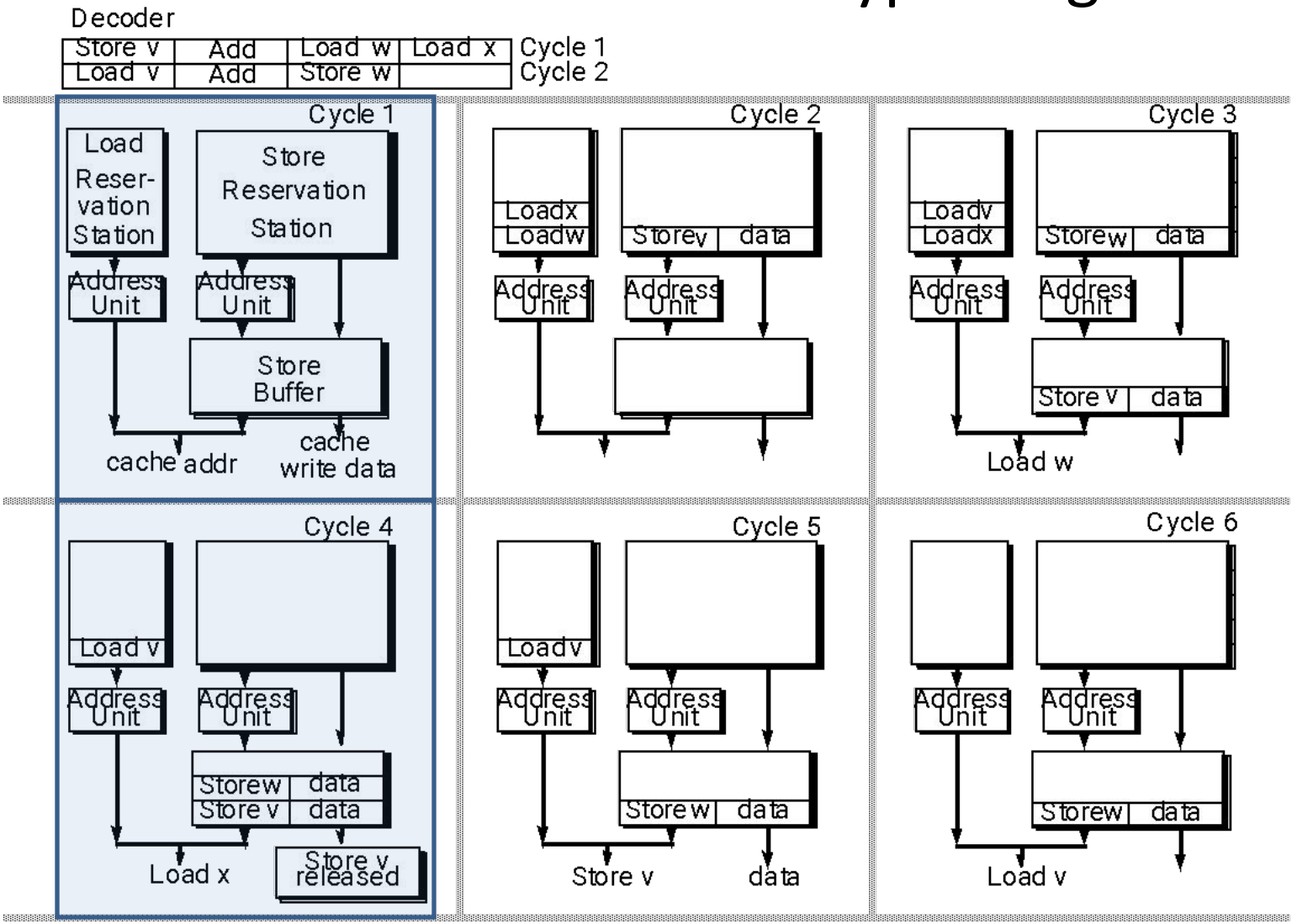


ISSUING LOADS AND STORES WITH TOTAL ORDERING

Load Bypassing

- **Loads can be allowed to bypass stores (if no aliasing).**
- **Two separate reservation stations and address generation units are employed for loads and stores.**
- **Store addresses still need to be computed before loads can be issued to allow checking for load dependences. If dependence cannot be checked, e.g. store address cannot be determined, then all subsequent loads are held until address is valid (conservative).**
- **Stores are kept in ROB until all previous instructions complete; and kept in the store buffer until gaining access to cache port.**
  - Store buffer is “future file” for memory

Illustration of Load Bypassing

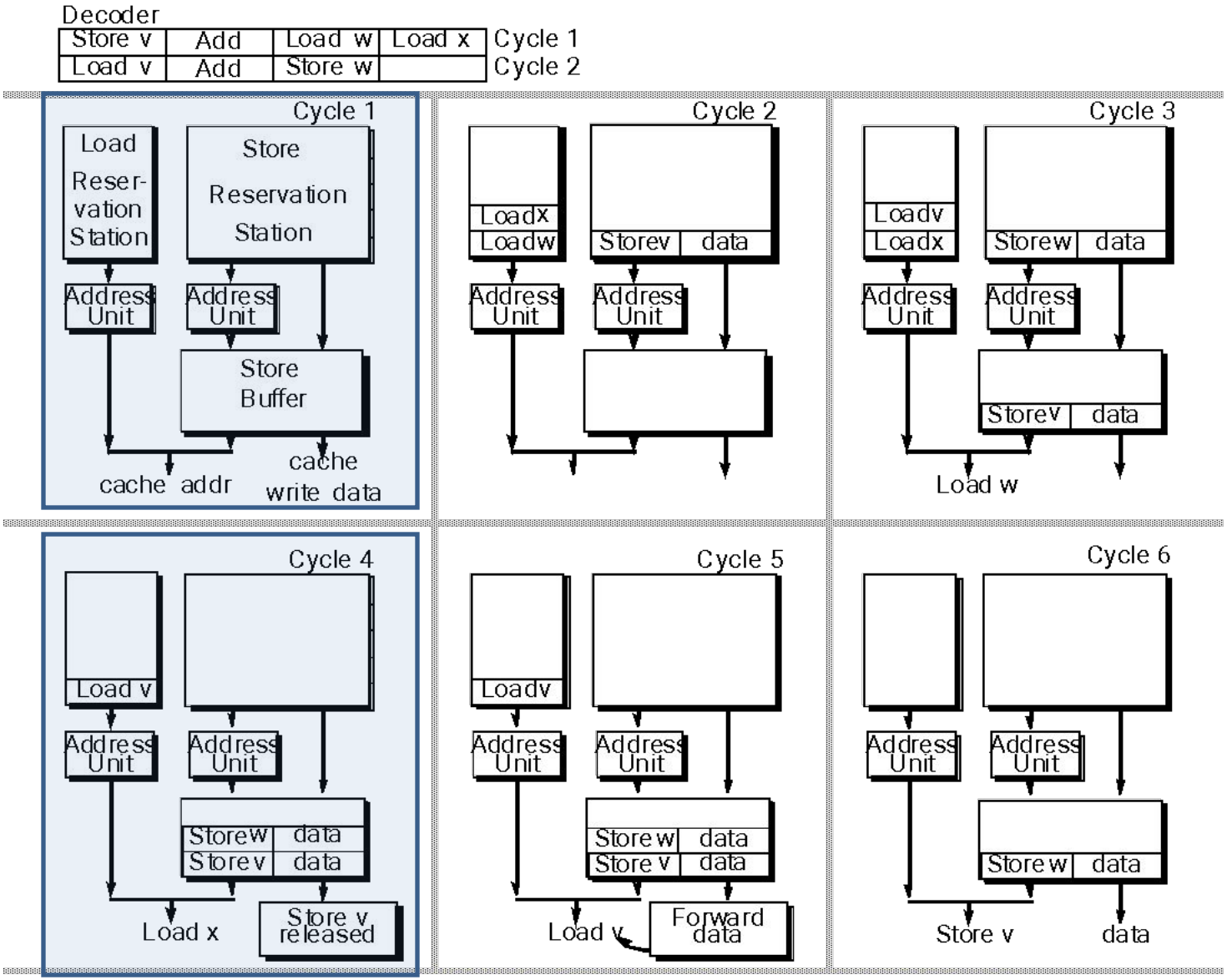


LOAD BYPASSING OF STORES

Load Forwarding

- **If a subsequent load has a dependence on a store still in the store buffer, it need not wait till the store is issued to the data cache.**
- **The load can be directly satisfied from the store buffer if the address is valid and the data is available in the store buffer.**
- **Since data is sourced from the store buffer:**
  - **Could avoid accessing the cache to reduce power/latency**

Illustration of Load Forwarding

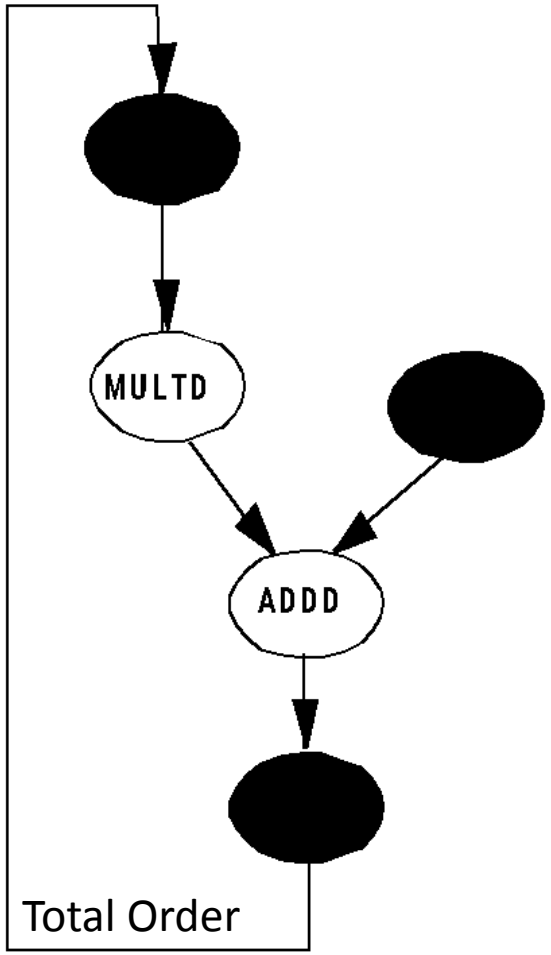


The DAXPY Example

Y(i) = A * X(i) + Y(i)

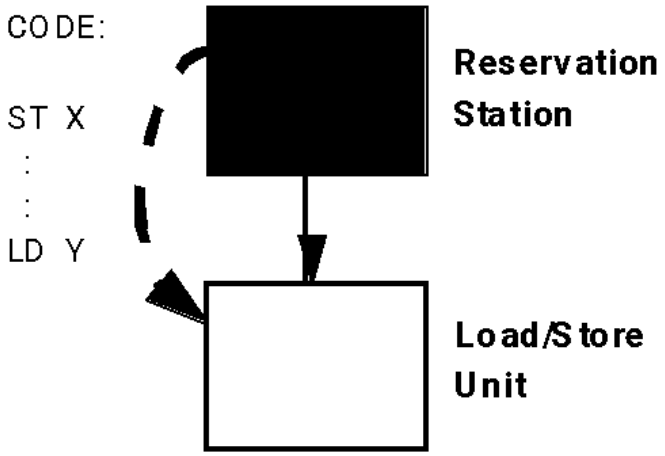
```
        LD      F0, a
        ADDI    R4, Rx, #512        ; last address

Loop:
        LD      F2, 0(Rx)           ; load X(i)
        MULTD   F2, F0, F2          ; A*X(i)
        LD      F4, 0(Ry)           ; load Y(i)
        ADDD    F4, F2, F4          ; A*X(i) + Y(i)
        SD      F4, 0(Ry)           ; store into Y(i)
        ADDI    Rx, Rx, #8          ; inc. index to X
        ADDI    Ry, Ry, #8          ; inc. index to Y
        SUB     R20, R4, Rx         ; compute bound
        BNZ     R20, loop           ; check if done
```

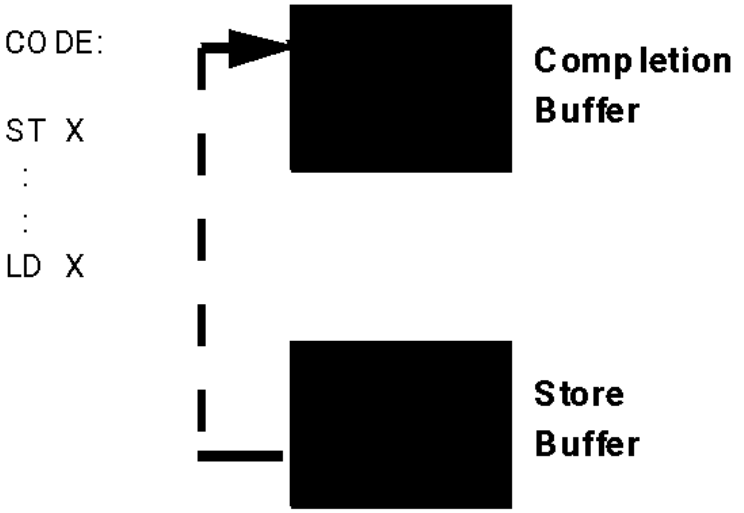


Performance Gains From Weak Ordering

Load Bypassing:



Load Forwarding:



Performance gain:

Load bypassing: 11% -19% increase over total ordering

Load forwarding: 1% -4% increase over load bypassing

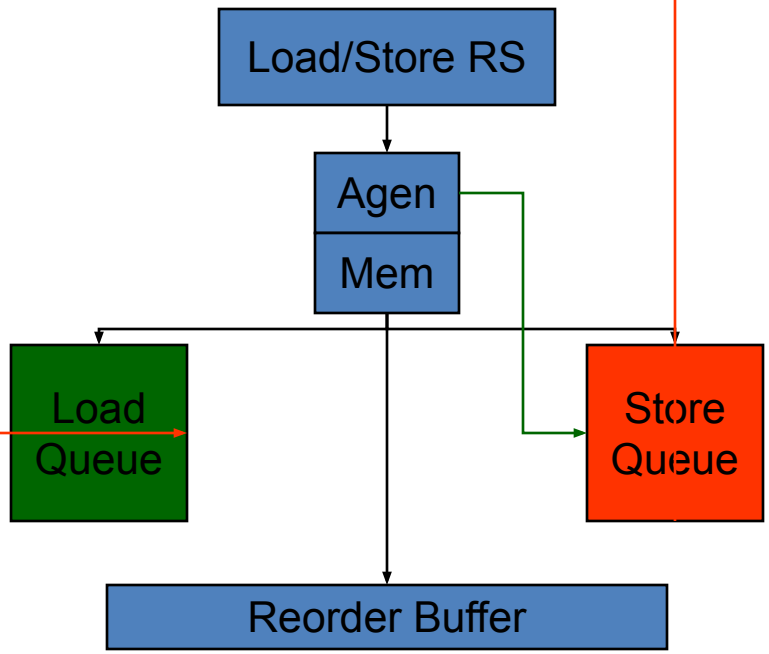
Optimizing Load/Store Disambiguation

- Non-speculative load/store disambiguation
  1. Loads wait for addresses of all prior stores
  2. Full address comparison
  3. Bypass if no match, forward if match
- (1) can limit performance:

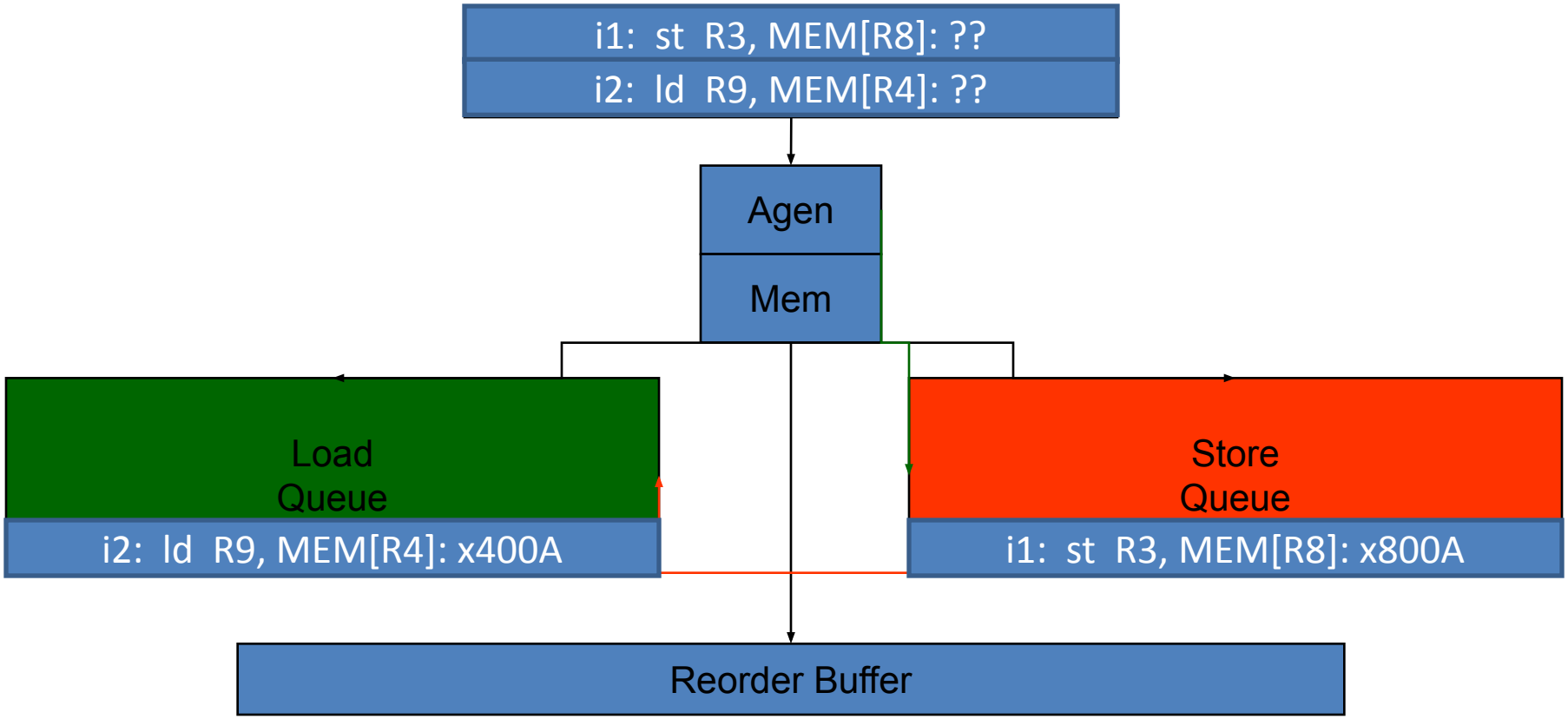
load r5,MEM[r3] ← cache miss
store r7, MEM[r5] ← RAW for agen, stalled
…
load r8, MEM[r9] ← independent load stalled

Speculative Disambiguation

- What if aliases are rare?
  1. Loads don't wait for addresses of all prior stores
  2. Full address comparison of stores that are ready
  3. Bypass if no match, forward if match
  4. Check all store addresses when they commit
     - No matching loads – speculation was correct
     - Matching unbypassed load – incorrect speculation
  5. Replay starting from incorrect load

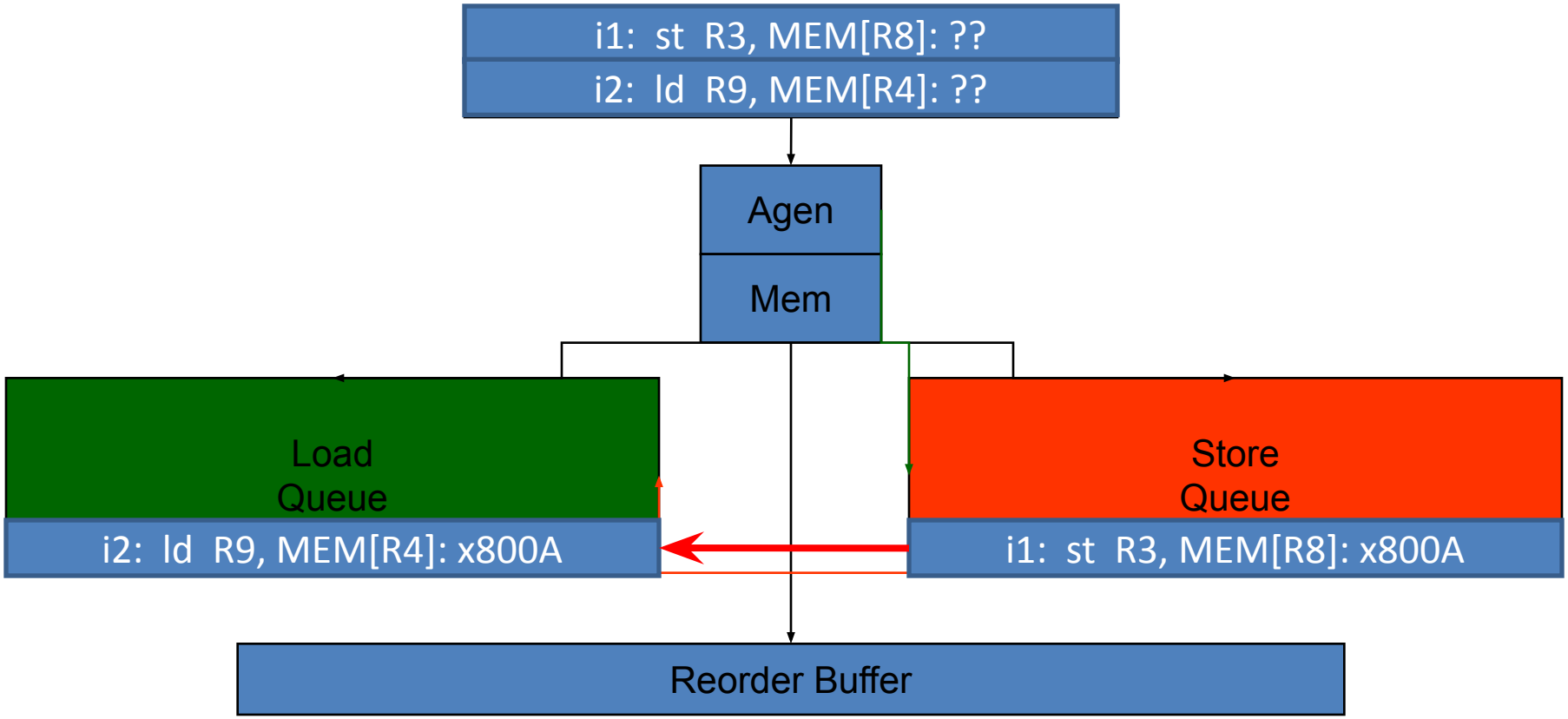


Speculative Disambiguation: Load Bypass



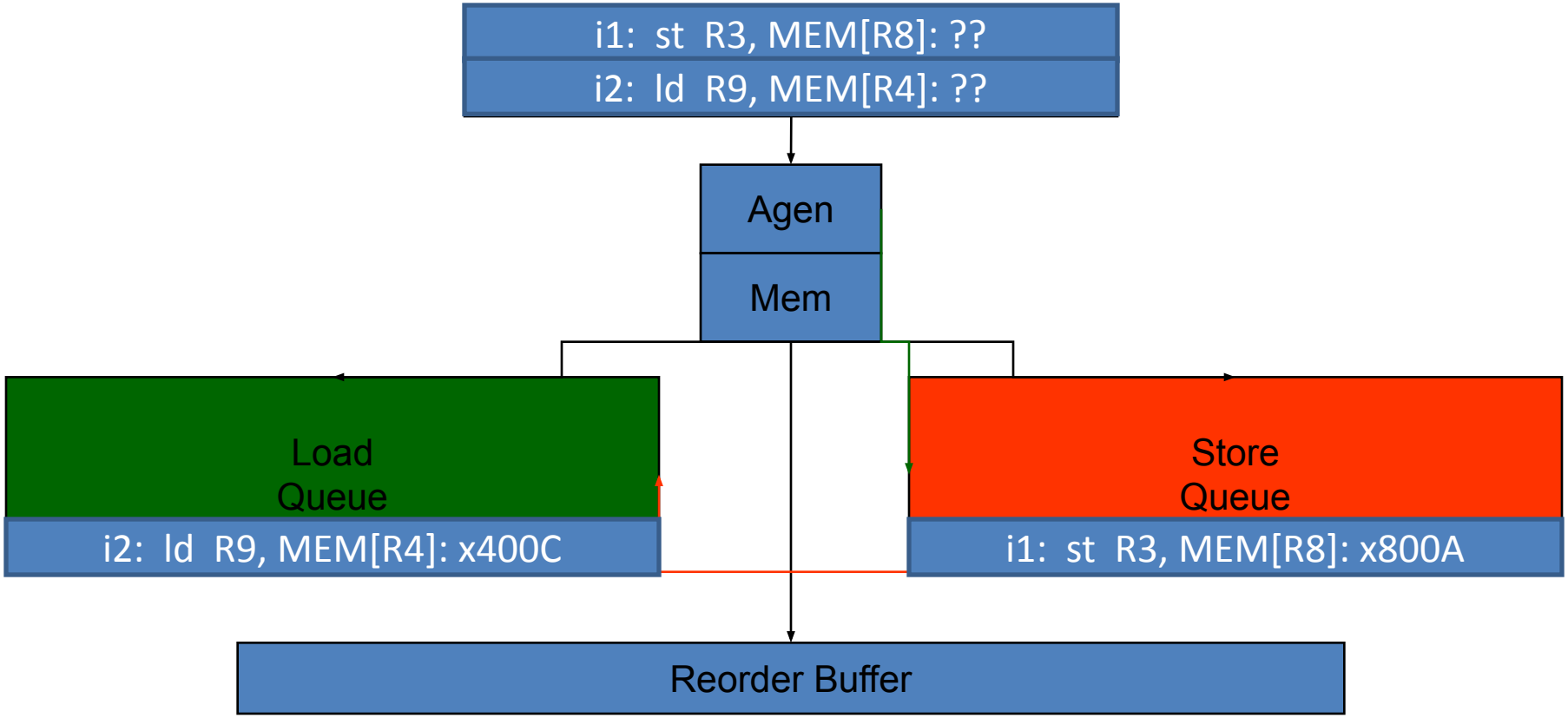
- i1 and i2 issue in program order
- i2 checks store queue (no match)

Speculative Disambiguation: Load Forward



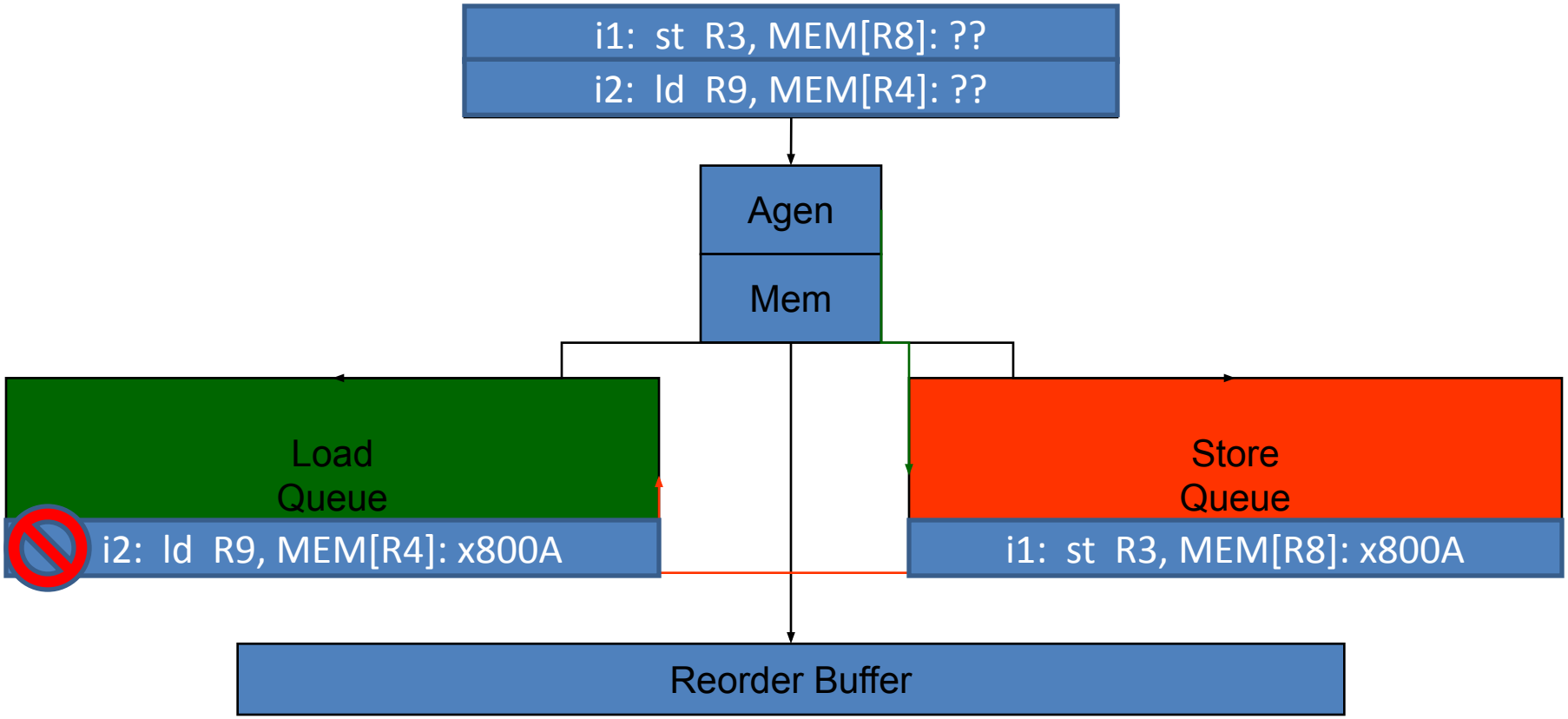
- i1 and i2 issue in program order
- i2 checks store queue (match=>forward)

Speculative Disambiguation: Safe Speculation



- i1 and i2 issue out of program order
- i1 checks load queue at commit (no match)

Speculative Disambiguation: Violation



- i1 and i2 issue out of program order
- i1 checks load queue at commit (match)
  - i2 marked for replay

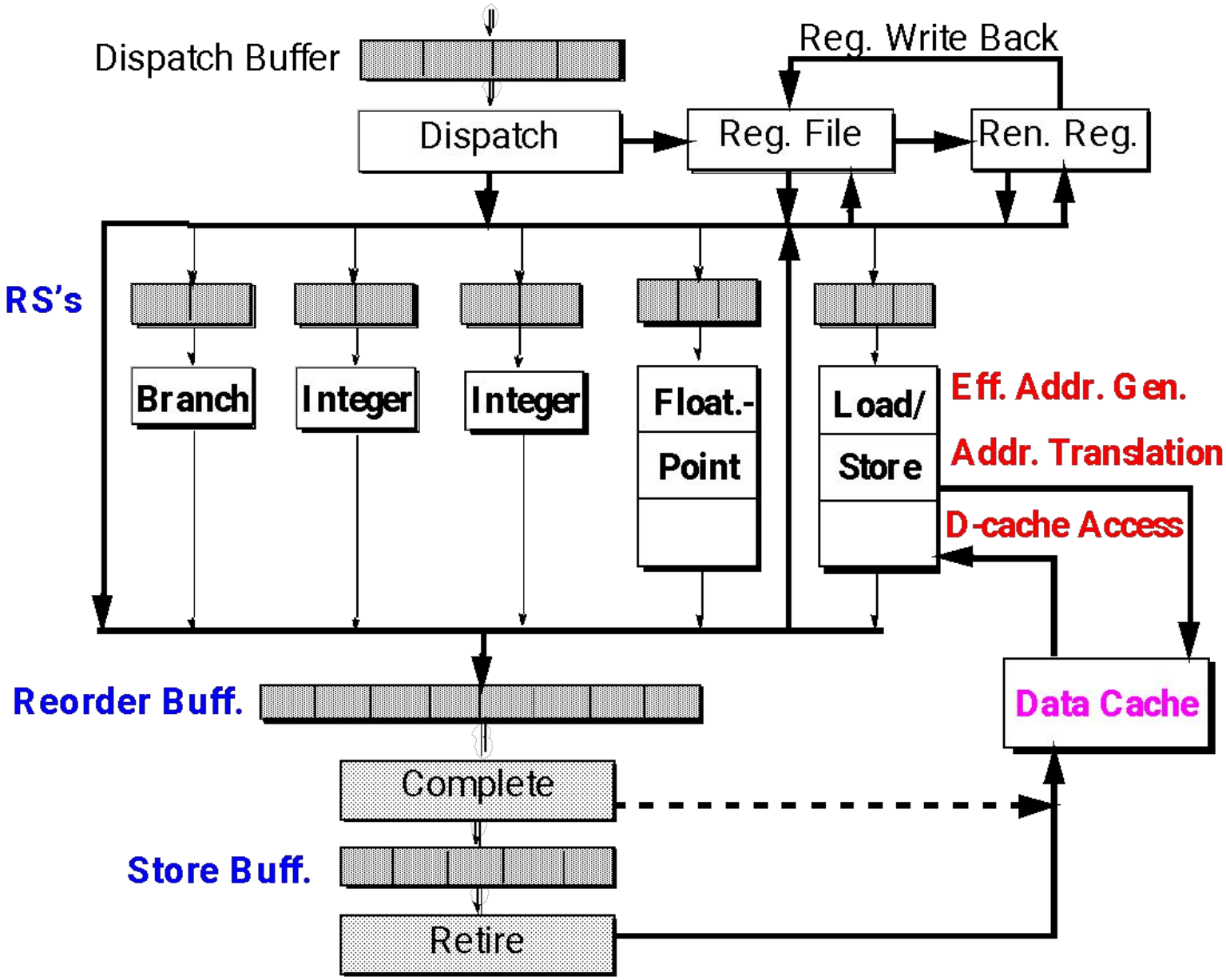
Use of Prediction

- If aliases are rare: static prediction
  - Predict no alias every time
    - Why even implement forwarding? PowerPC 620 doesn't
  - Pay misprediction penalty rarely
- If aliases are more frequent: dynamic prediction
  - Use PHT-like history table for loads
    - If alias predicted: delay load
    - If aliased pair predicted: forward from store to load
      - More difficult to predict pair [store sets, Alpha 21264]
  - Pay misprediction penalty rarely
- Memory cloaking [Moshovos, Sohi]
  - Predict load/store pair
  - Directly copy store data register to load target register
  - Reduce data transfer latency to absolute minimum

Load/Store Disambiguation Discussion

- RISC ISA:
  - Many registers, most variables allocated to registers
  - Aliases are rare
  - Most important to not delay loads (bypass)
  - Alias predictor may/may not be necessary
- CISC ISA:
  - Few registers, many operands from memory
  - Aliases much more common, forwarding necessary
  - Incorrect load speculation should be avoided
  - If load speculation allowed, predictor probably necessary
- Address translation:
  - Can't use virtual address (must use physical)
  - Wait till after TLB lookup is done
  - Or, use subset of untranslated bits (page offset)
    - Safe for proving inequality (bypassing OK)
    - Not sufficient for showing equality (forwarding not OK)

The Memory Bottleneck



Load/Store Processing

For both Loads and Stores:

1. **Effective Address Generation:**

   Must wait on register value

   Must perform address calculation

2. **Address Translation:**

   Must access TLB

   Can potentially induce a page fault (exception)

For Loads: D-cache Access (Read)

Can potentially induce a D-cache miss

Check aliasing against store buffer for possible load forwarding

If bypassing store, must be flagged as “speculative” load until completion

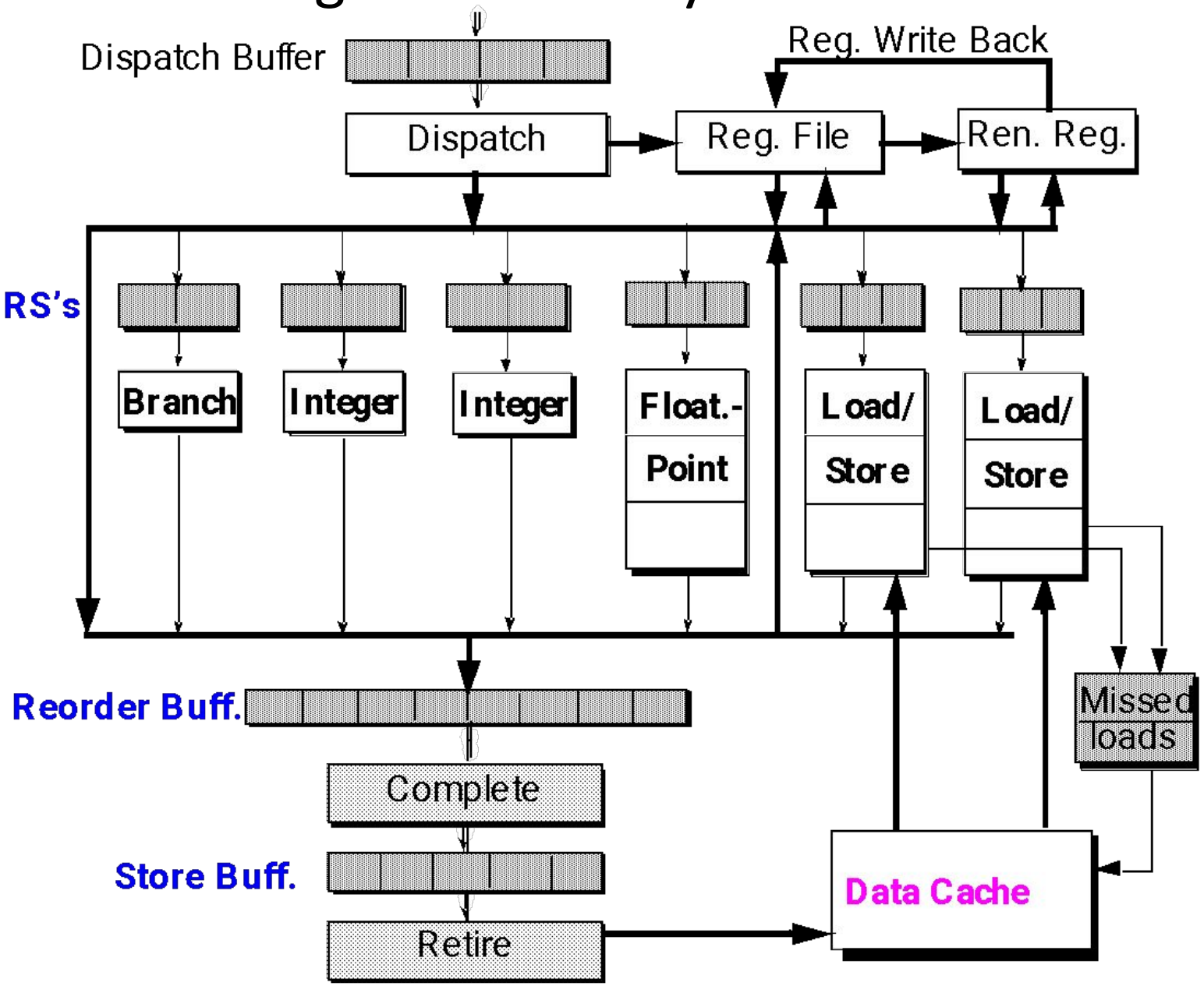
For Stores: D-cache Access (Write)

When completing must check aliasing against “speculative” loads

After completion, wait in store buffer for access to D-cache

Can potentially induce a D-cache miss

Easing The Memory Bottleneck



Memory Bottleneck Techniques

Dynamic Hardware (Microarchitecture):

- Use Multiple Load/Store Units (need multiported D-cache)
- Use More Advanced Caches (victim cache, stream buffer)
- Use Hardware Prefetching (need load history and stride detection)
- Use Non-blocking D-cache (need missed-load buffers/MSHRs)
- Large instruction window (memory-level parallelism)

Static Software (Code Transformation):

- Insert Prefetch or Cache-Touch Instructions (mask miss penalty)
- Array Blocking Based on Cache Organization (minimize misses)
- Reduce Unnecessary Load/Store Instructions (redundant loads)
- Software Controlled Memory Hierarchy (expose it to above DSI)

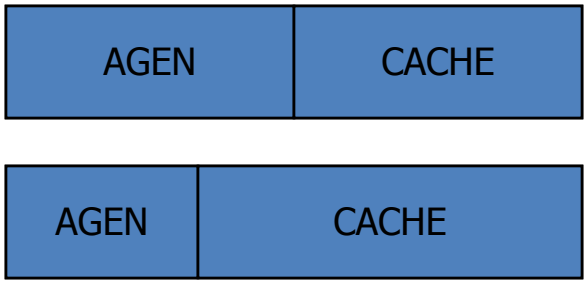
Caches and Performance

- Caches
  - Enable design for common case: cache hit
    - Cycle time, pipeline organization
    - Recovery policy
  - Uncommon case: cache miss
    - Fetch from next level
      - Apply recursively if multiple levels
    - What to do in the meantime?
- What is performance impact?
- Various optimizations are possible

Performance Impact

- Cache hit latency
  - Included in “pipeline” portion of CPI
    - E.g. IBM study: 1.15 CPI with 100% cache hits
  - Typically 1-3 cycles for L1 cache
    - Intel/HP McKinley: 1 cycle
      - Heroic array design
      - No address generation: load r1, (r2)
    - IBM Power4: 3 cycles
      - Address generation
      - Array access
      - Word select and align
      - Register file write (no bypass)

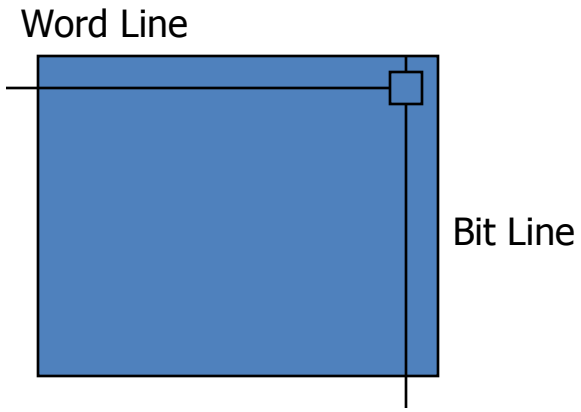
Cache Hit continued



- Cycle stealing common
  - Address generation < cycle
  - Array access > cycle
  - Clean, FSD cycle boundaries violated
- Speculation rampant
  - “Predict” cache hit
  - Don’t wait for (full) tag check
  - Consume fetched word in pipeline
  - Recover/flush when miss is detected
    - Reportedly 7 (!) cycles later in Pentium 4

Cache Hits and Performance

- Cache hit latency determined by:
  - Cache organization
    - Associativity
      - Parallel tag checks expensive, slow
      - Way select slow (fan-in, wires)
    - Block size
      - Word select may be slow (fan-in, wires)
    - Number of block (sets x associativity)
      - Wire delay across array
      - “Manhattan distance” = width + height
      - Word line delay: width
      - Bit line delay: height



- Array design is an art form
  - Detailed analog circuit/wire delay modeling

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

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*

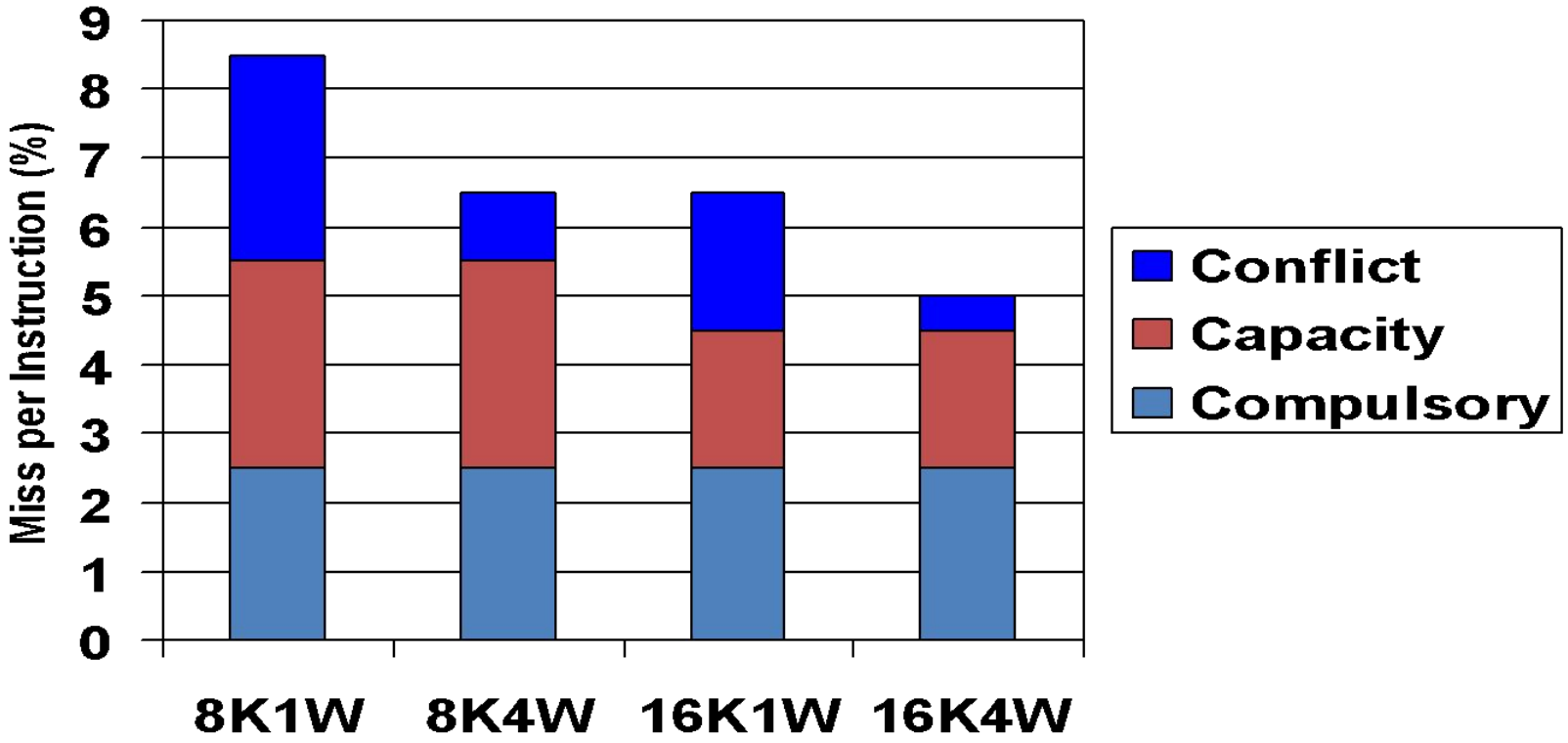
Cache Miss Rate Effects

- 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

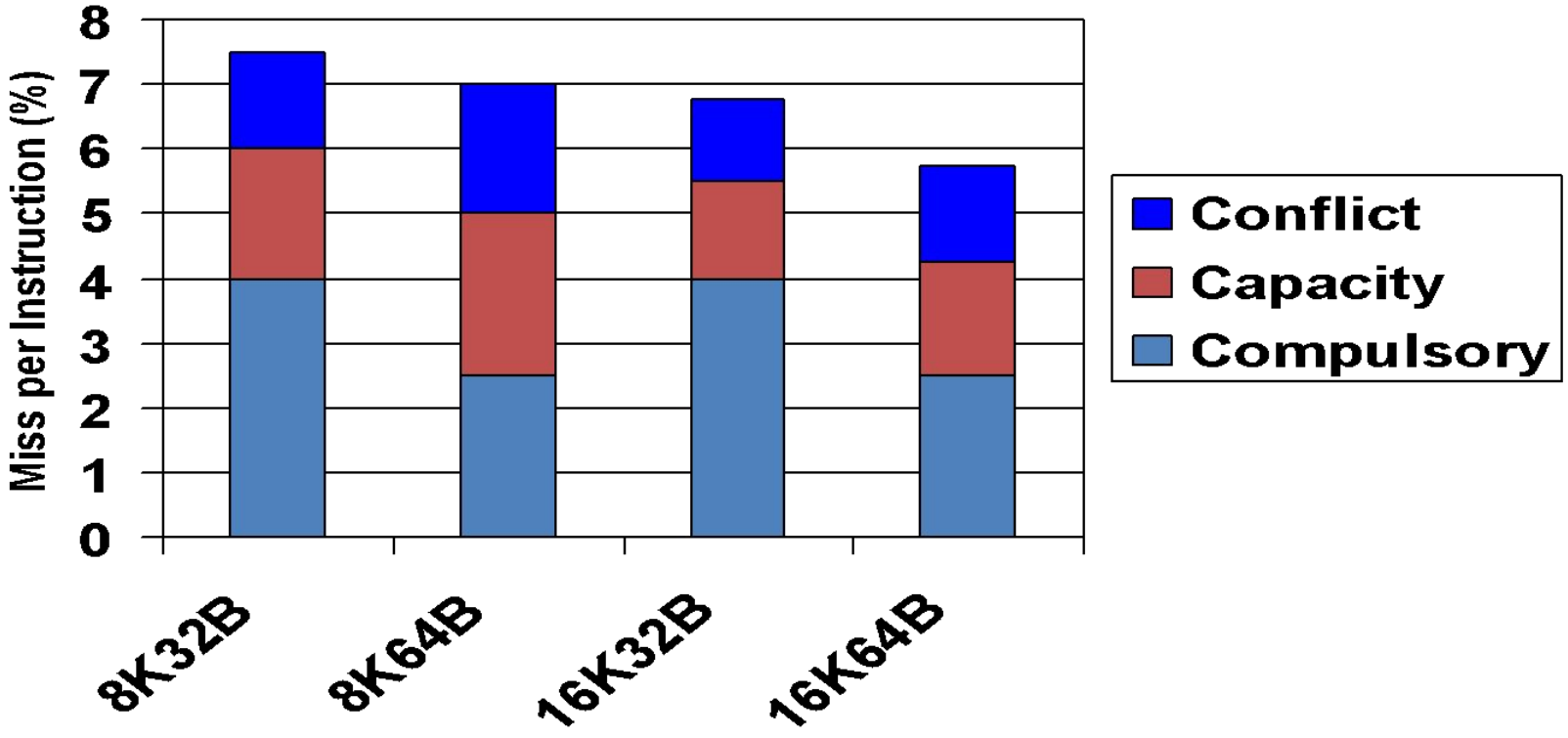
- Subtle tradeoffs between cache organization parameters
  - Large blocks reduce compulsory misses but increase miss penalty
    - #compulsory ~= (working set) / (block size)
    - #transfers = (block size)/(bus width)
  - Large blocks increase conflict misses
    - #blocks = (cache size) / (block size)
  - 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

Summary

- **Memory Data Flow**
  - Memory Data Dependences
  - Load Bypassing
  - Load Forwarding
  - Speculative Disambiguation
  - The Memory Bottleneck
- Cache Hits and Cache Misses
- Further coverage of memory hierarchy later in the semester