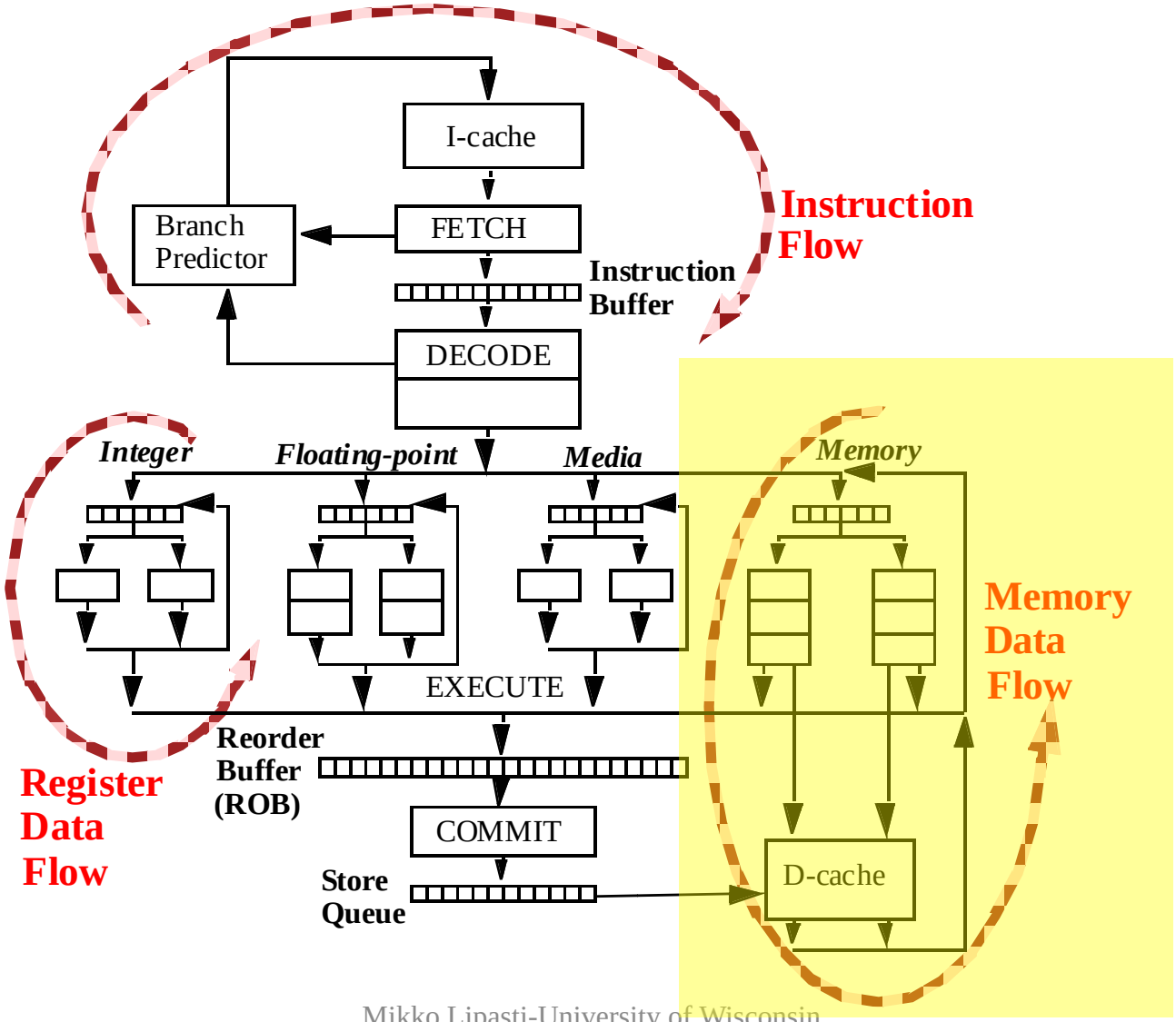
Memory Data Flow

ECE/CS 752 Fall 2017

Prof. Mikko H. Lipasti
University of Wisconsin-Madison

High-IPC Processor

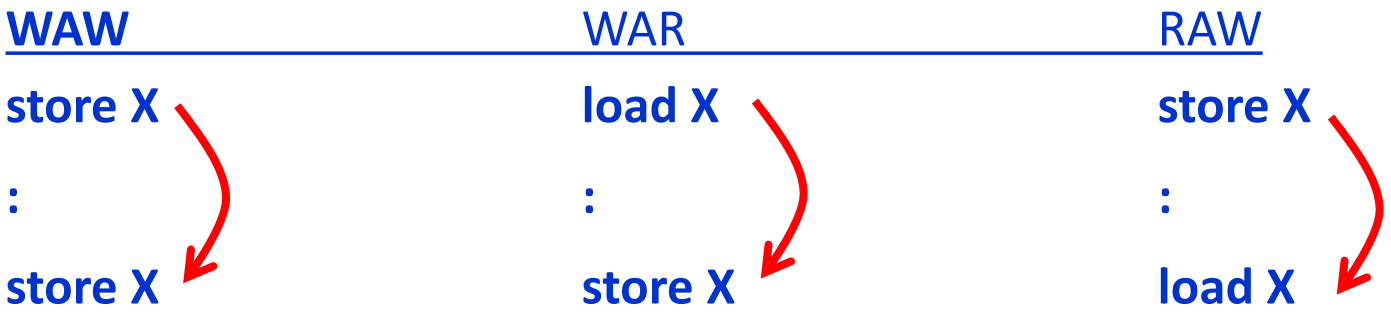


Memory Data Flow

- Memory Data Flow Challenges
  - Memory Data Dependences
  - Load Bypassing
  - Load Forwarding
  - Speculative Disambiguation
  - The Memory Bottleneck
- Cache Hits and Cache Misses
- Replacement Policies
- Prefetching

Memory Data Dependences

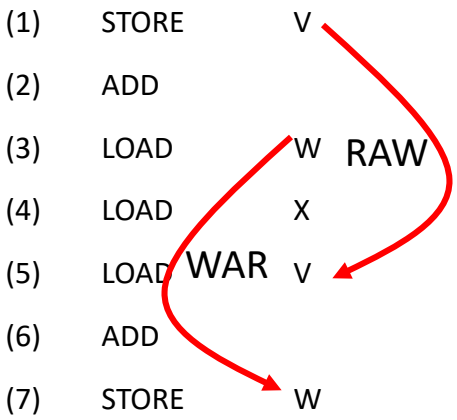
- **Besides branches, long memory latencies are one of the biggest performance challenges today.**
- **To preserve sequential (in-order) state in the data caches and external memory (so that recovery from exceptions is possible) stores are performed in order. This takes care of antidependences and output 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:

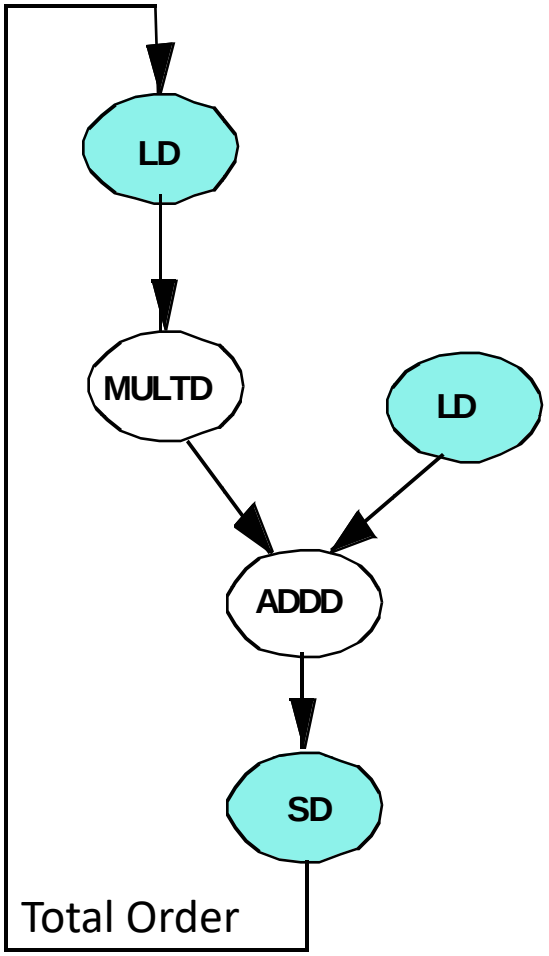


The DAXPY Example

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
```



Out-of-order Load Issue

Required for high performance

Hardware must monitor prior stores

- No alias: loads free to issue
- Alias: load must honor RAW

Complications

- Large comparators (64-bit addresses)
- Relative order of interleaved stores and loads
  - (must forward from most recent prior store)
- Speculative vs. non-speculative load issue

Optimizing Load/Store Disambiguation

Non-speculative load/store disambiguation

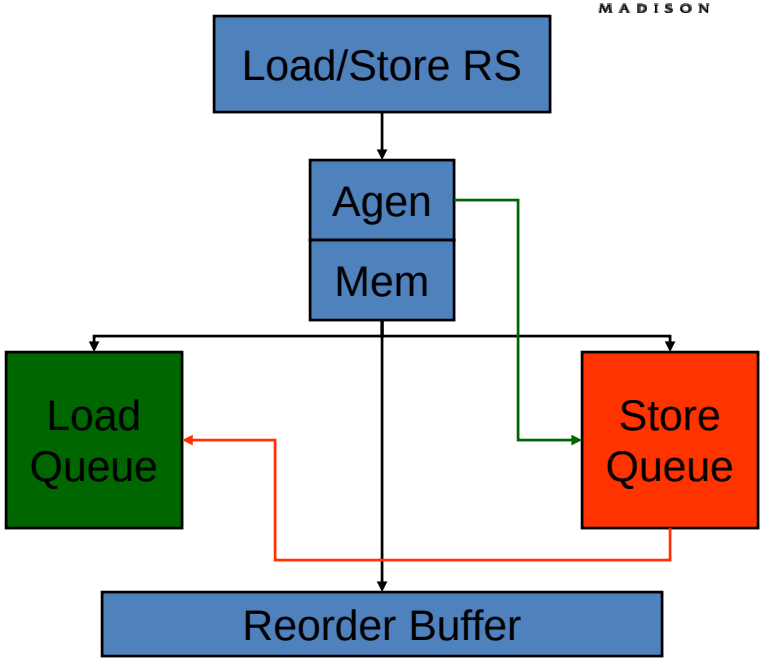
1. Loads wait for addresses of all prior stores
2. Full address comparison
3. Bypass if no match, forward if match

Step (1) can unnecessarily limit performance:

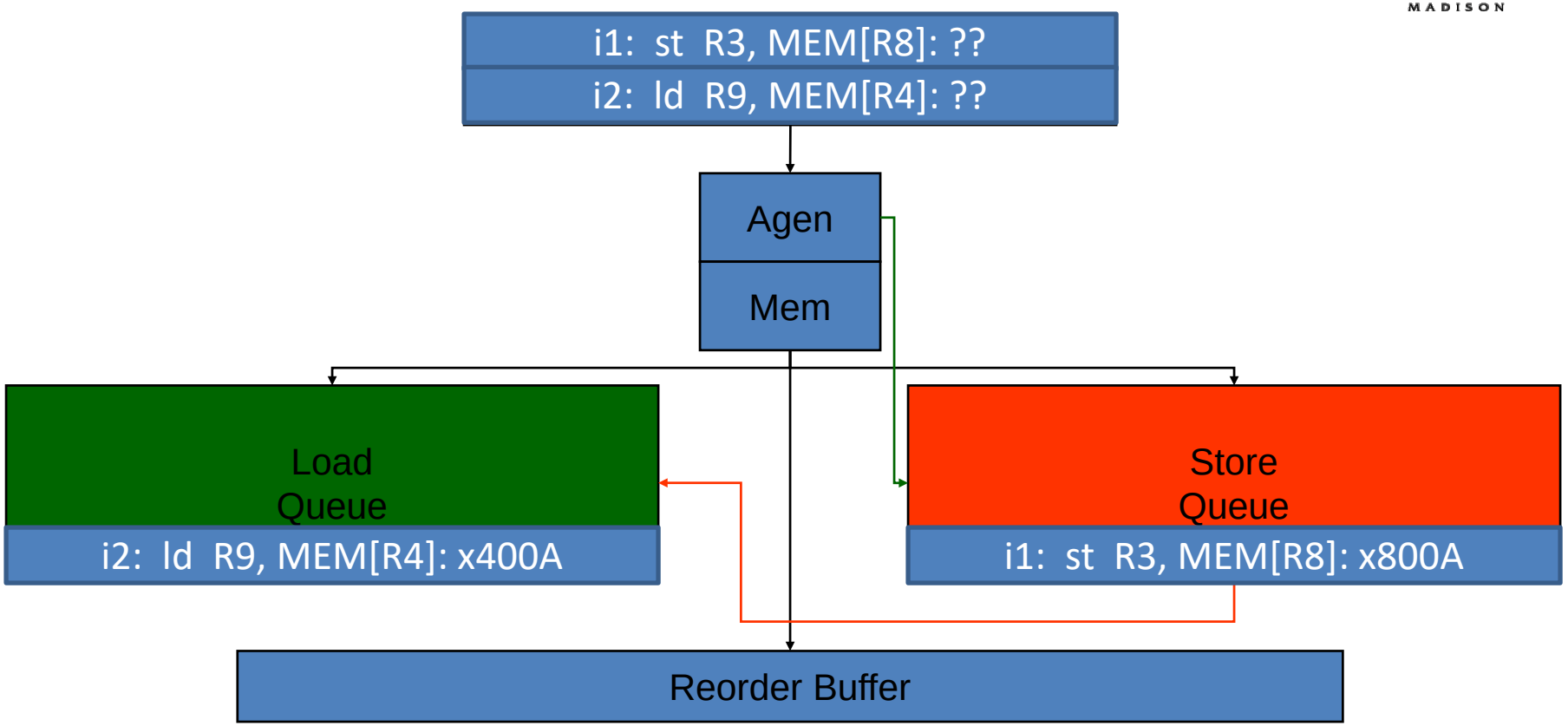
load r5,MEM[r3] ← cache miss
store r7, MEM[r5] ← RAW for agen, stalled
…
load r8, MEM[r9] ← independent load stalled

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

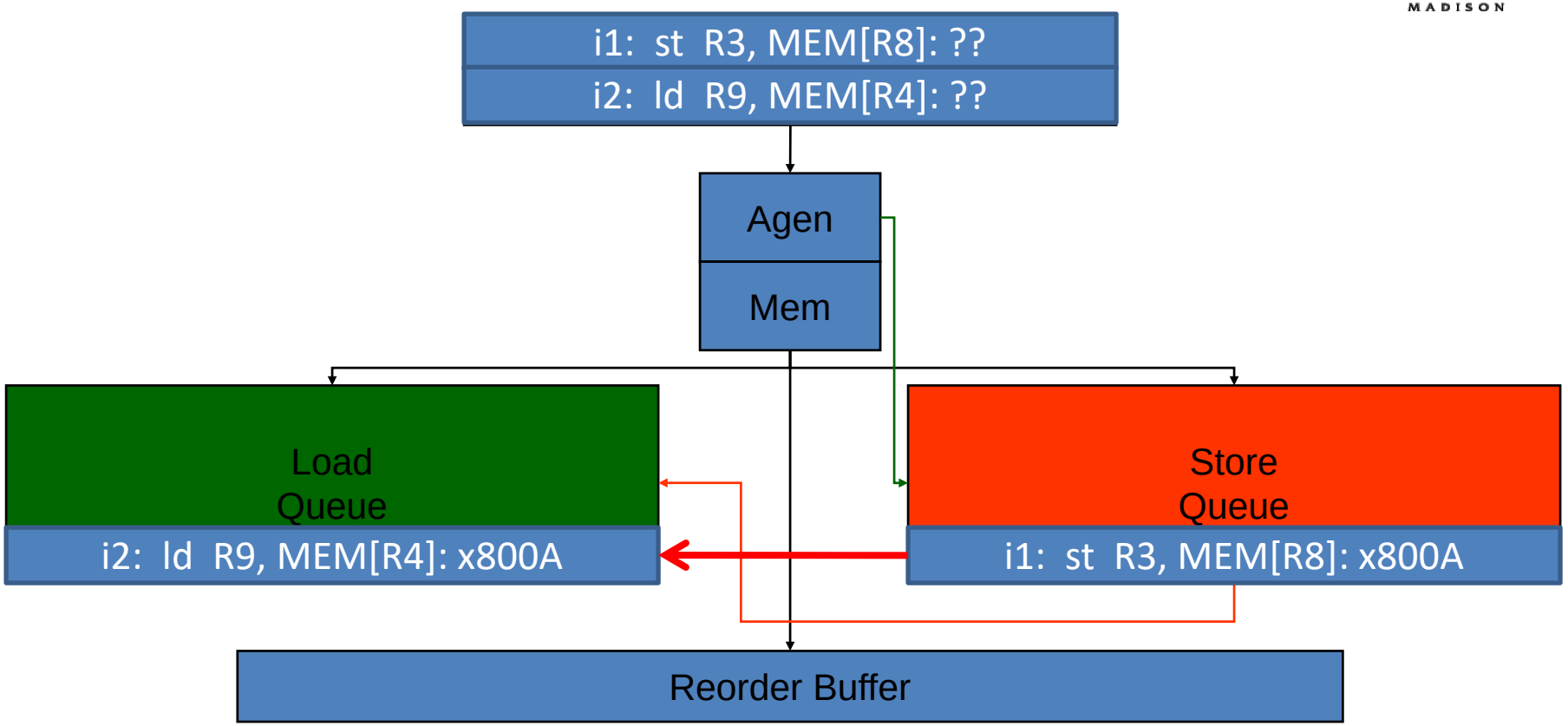


Speculative Disambiguation: Load Bypass



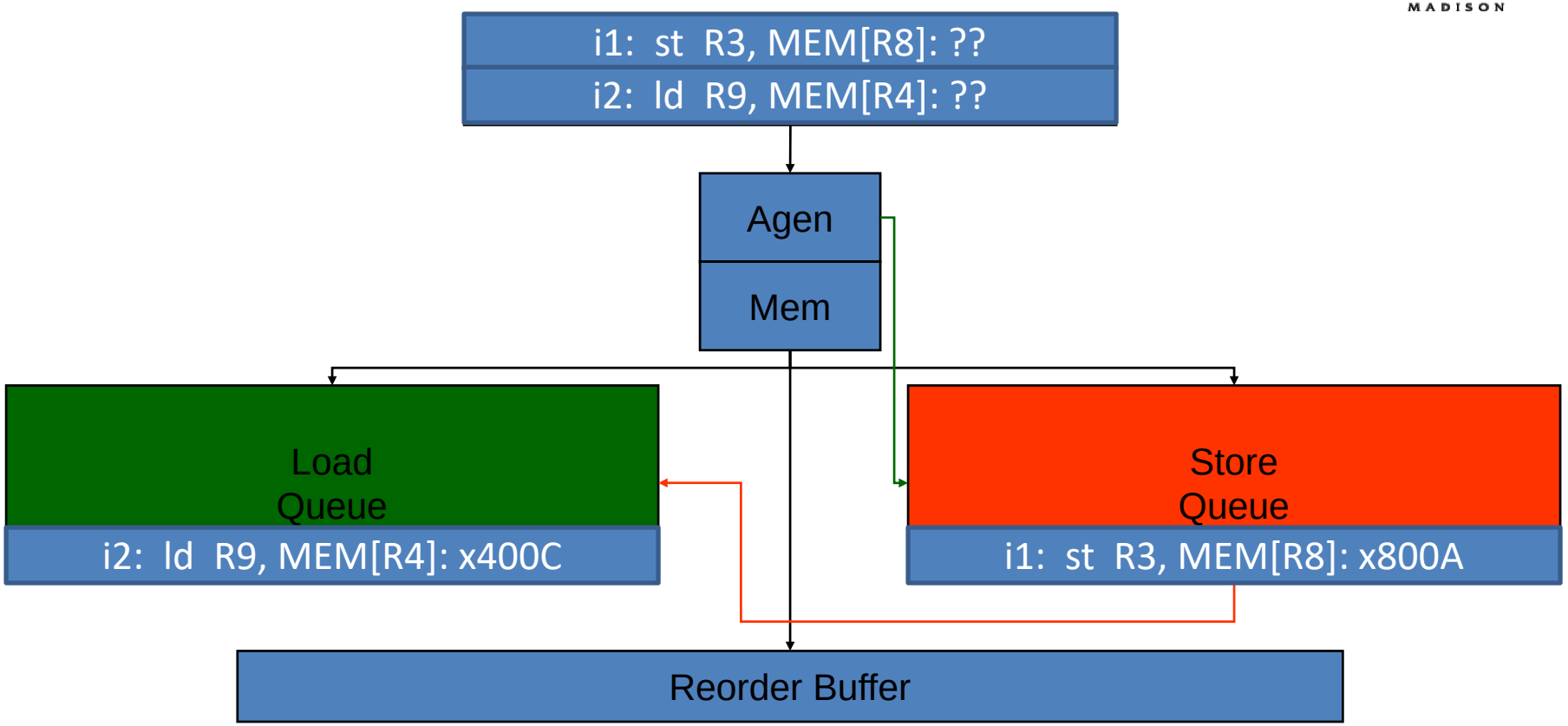
- i1 and i2 issue in program order
- i2 checks store queue (no match)

Speculative Disambiguation: Load Forward



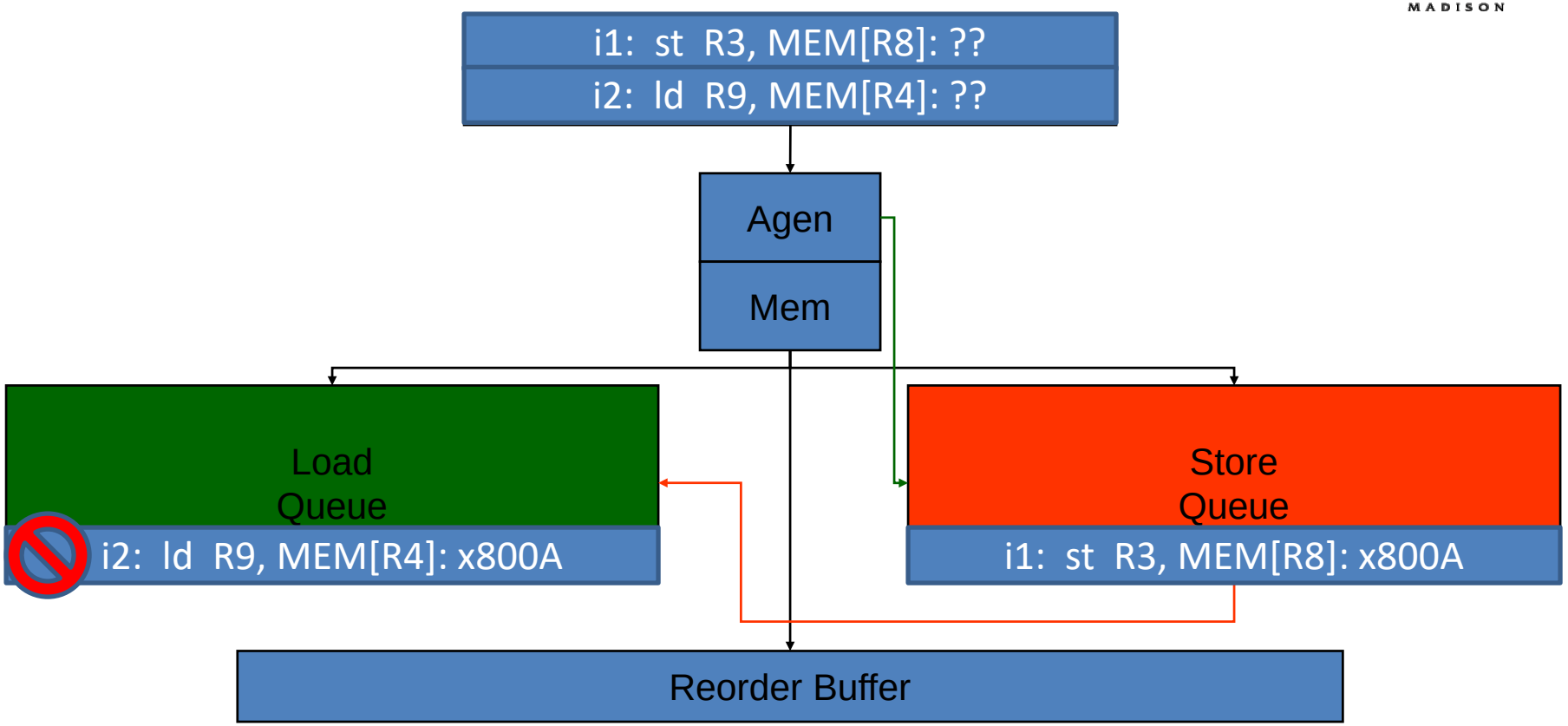
- i1 and i2 issue in program order
- i2 checks store queue (match=>forward)

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

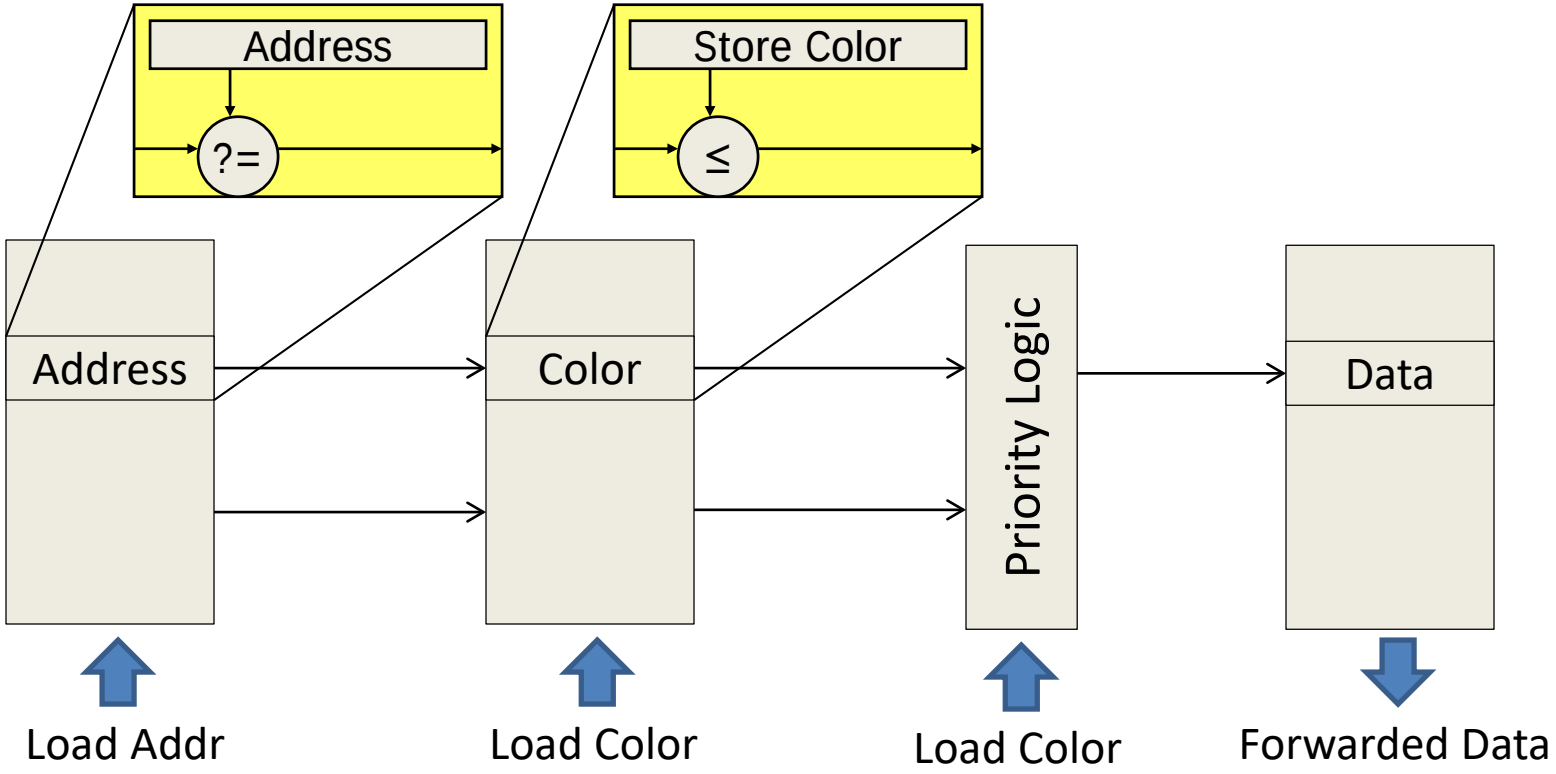
Use of Prediction

- 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, ISCA 1997]
  - Predict load/store pair
  - Directly copy store data register to load target register
  - Reduce data transfer latency to absolute minimum

Load/Store Disambiguation Discussion

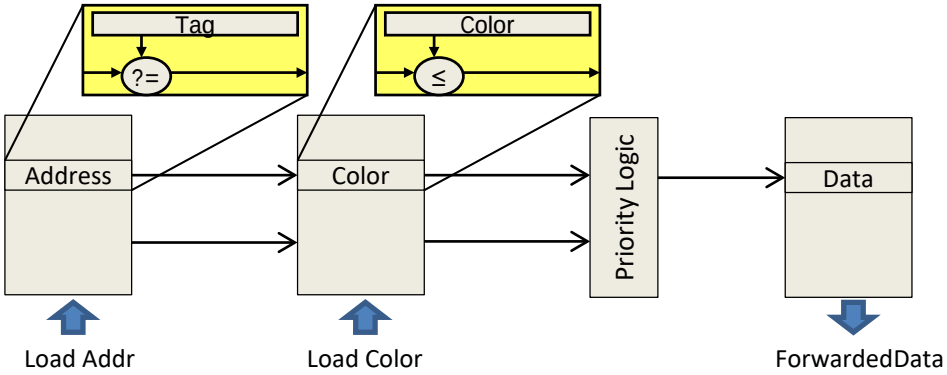
- RISC ISA:
  - Many registers, most variables allocated to registers
  - Aliases are rare
  - Most important to not delay loads (bypass)
  - Alias predictor may/may not be necessary
- CISC ISA:
  - Few registers, many operands from memory
  - Aliases much more common, forwarding necessary
  - Incorrect 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)

Store Queue Implementation



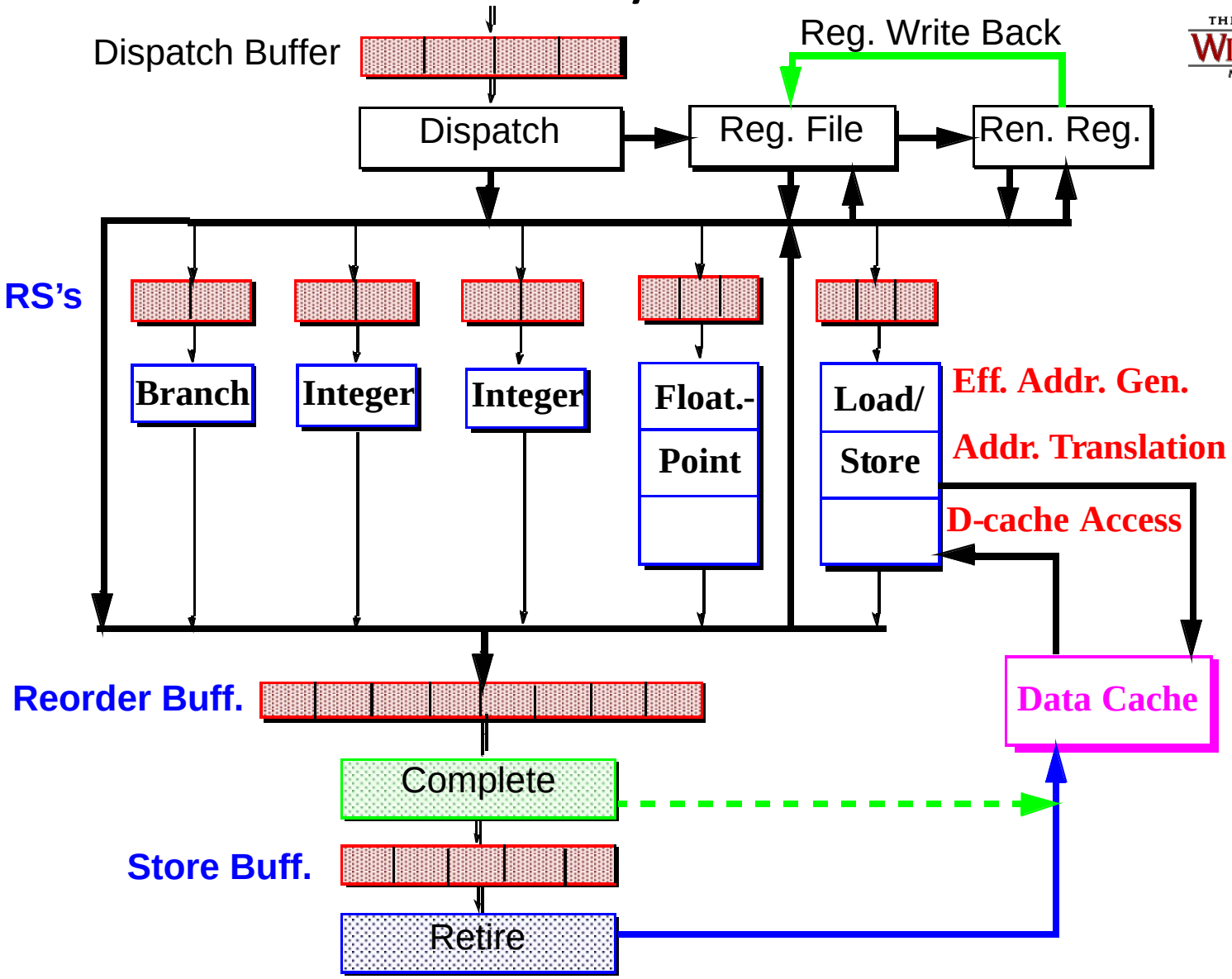
- Store color assigned at dispatch, increases monotonically
- Load inherits color from preceding store, only forwards if store is older
- Priority logic must find nearest matching store

Store Queue Complications



- If entries are positional, priority logic looks like carry chain (slow)
- If entries are not positional, priority logic is quite complex
  - See [Buyuktosunogly, El-Moursy, Albonesi, 2002 IEEE ASIC/SOC Conference]
- Partial store/load overlap may prevent bypassing (not all bytes present)
  - Must stall load instead
- Store color has finite range, clever logic trick:
  - For 2^n store queue entries, use (n+1) bits for color, e.g. 16 SQ entries requires 5 bits
  - If leading bit of oldest store is zero, use unsigned comparisons
  - If leading bit of oldest store is one, use signed comparisons

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

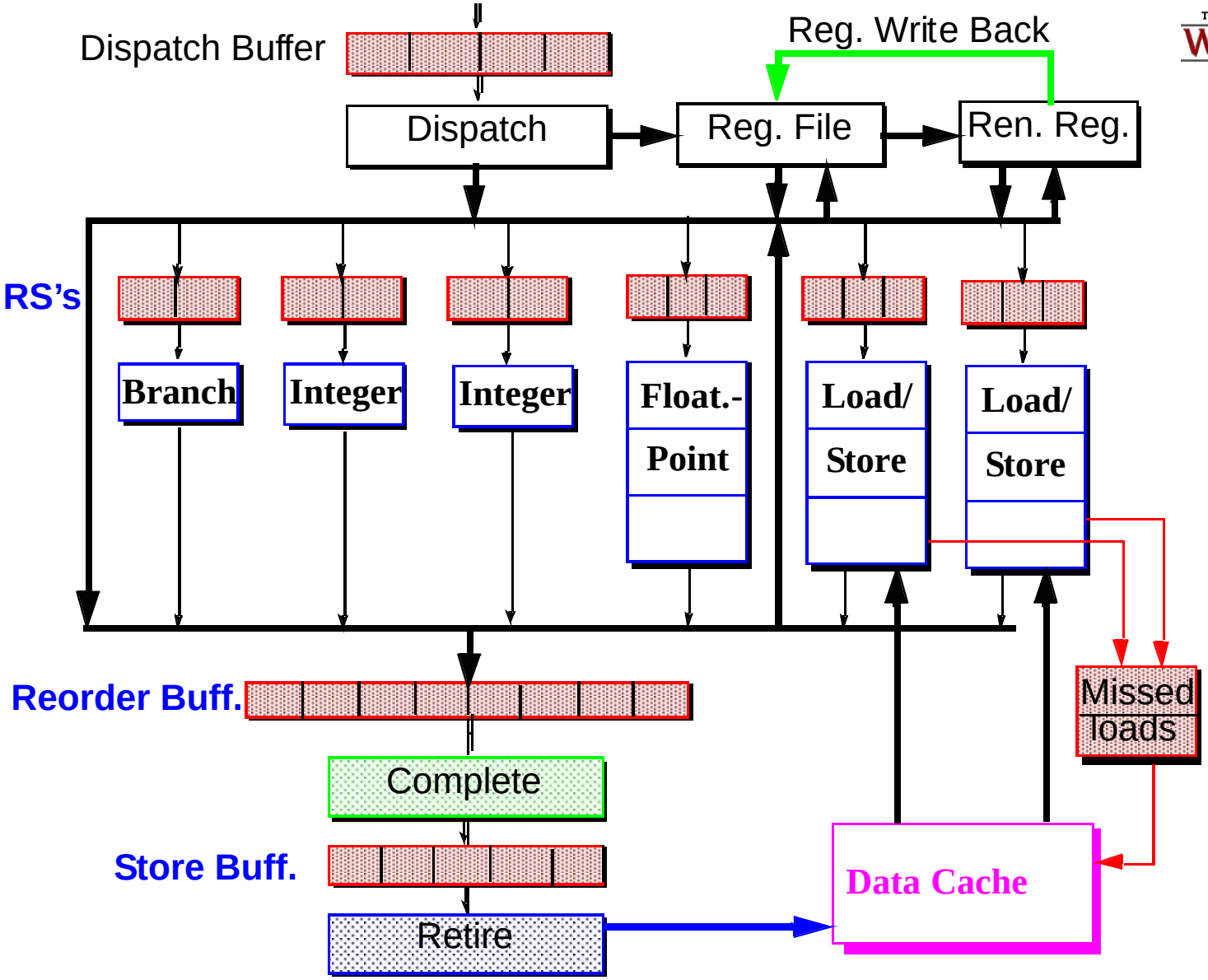
For Stores: D-cache Access (Write)

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



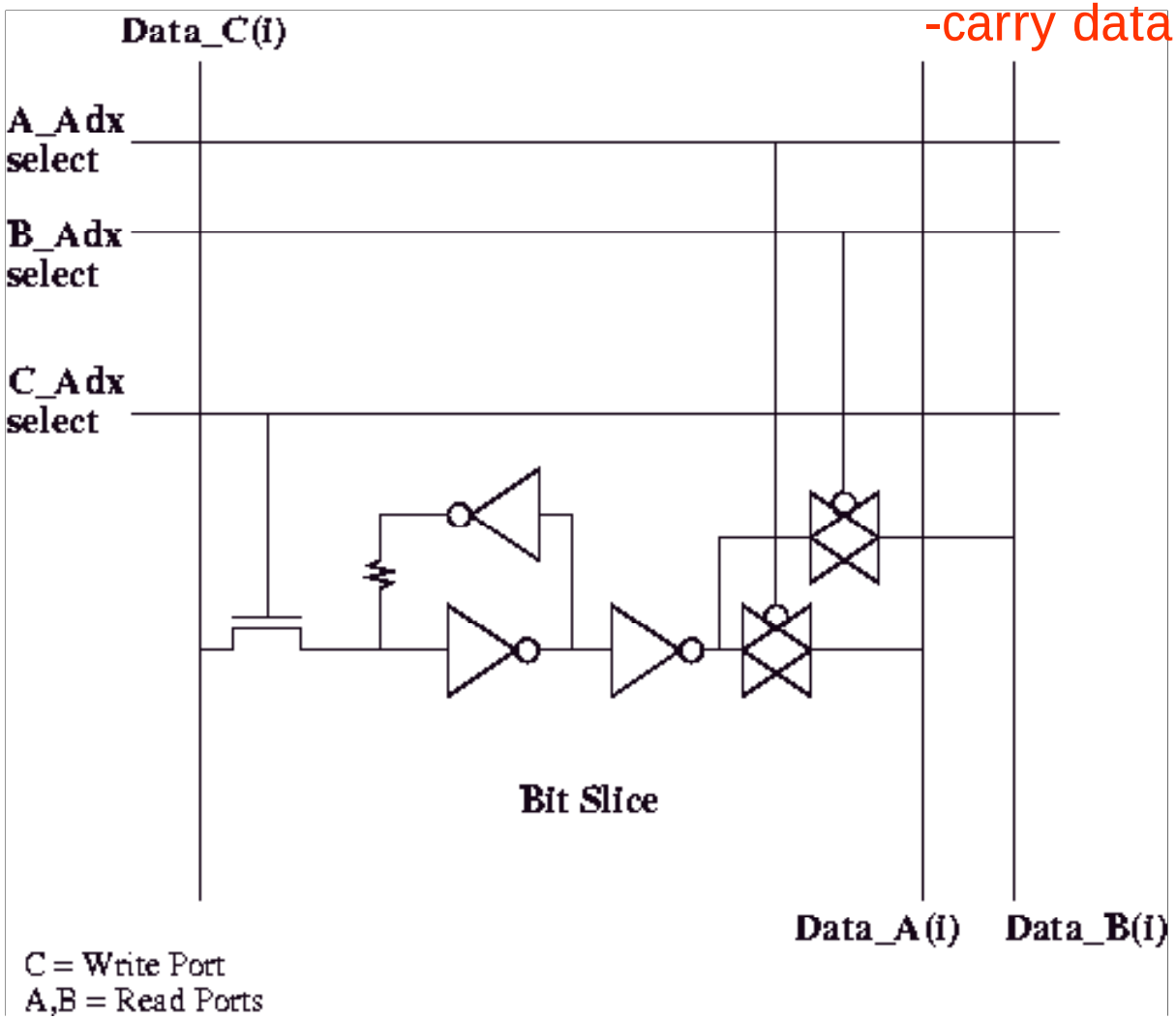
Superscalar Caches

- Increasing issue width => wider caches
- Parallel cache accesses are harder than parallel functional units
- Fundamental difference:
  - Caches have **state**, functional units don't
  - Operation thru one port affects future operations thru others
- Several approaches used
  - True multi-porting
  - Multiple cache copies
  - Virtual multi-porting
  - Multi-banking (interleaving)

True Multiporting of SRAM

"Bit" Lines
-carry data in/out

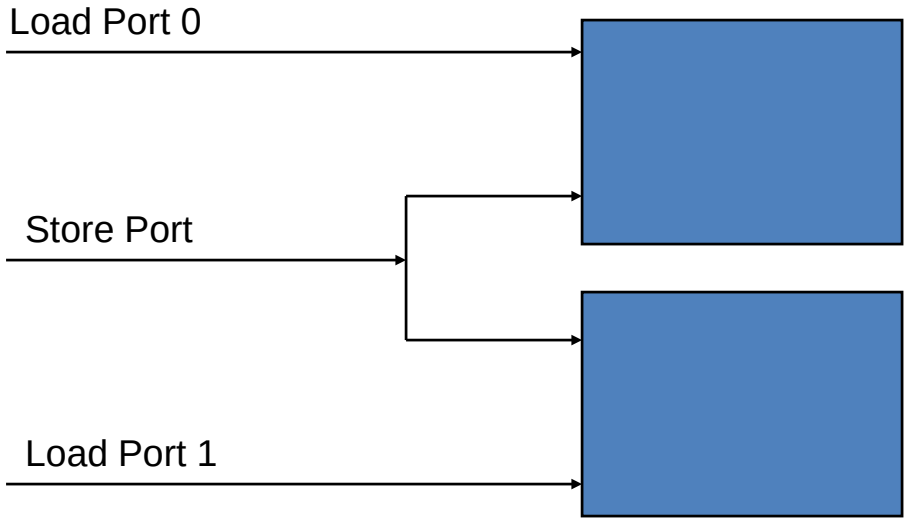
"Word" Lines
-select a row



True Multiporting of SRAM

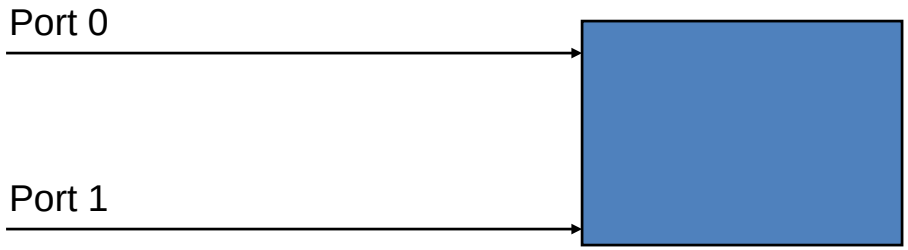
- Would be ideal
- Increases cache area
  - Array becomes wire-dominated
- Slower access
  - Wire delay across larger area
  - Cross-coupling capacitance between wires
- Read stability suffers if latch drives bitline
  - Need isolating access logic (2T per bitline)

Multiple Cache Copies



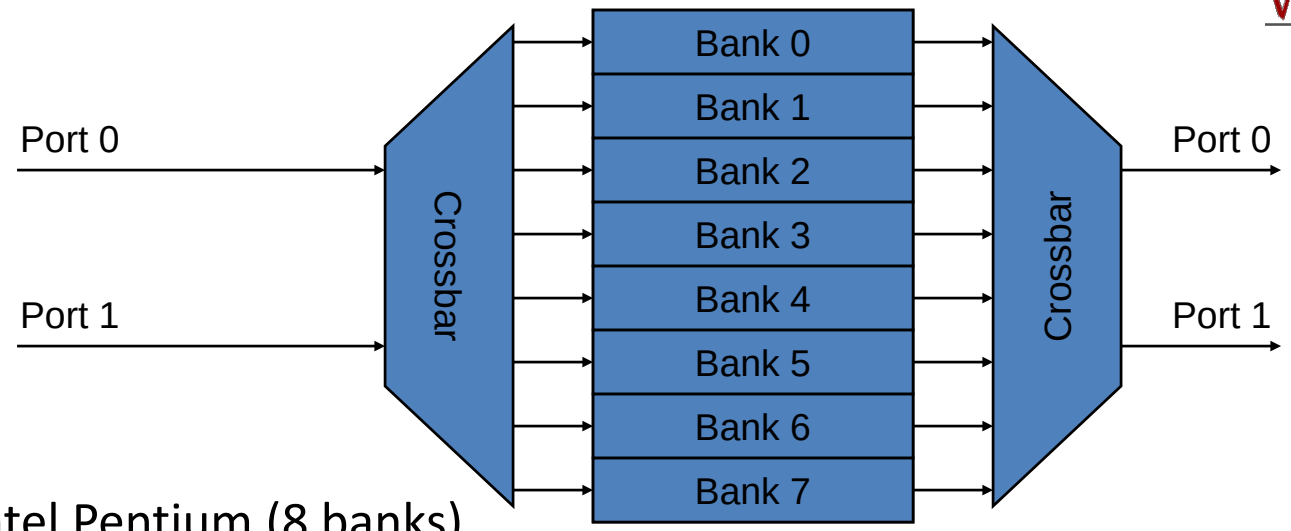
- Used in DEC Alpha 21164, IBM Power4
- Independent load paths
- Single shared store path
  - May be exclusive with loads, or internally dual-ported
- Bottleneck, not practically scalable beyond 2 paths
- Provides some fault-tolerance
  - Parity protection per copy
  - Parity error: restore from known-good copy
  - Avoids more complex ECC (no RMW for subword writes), still provides SEC

Virtual Multiporting



- Used in IBM Power2 and DEC 21264
  - Wave pipelining: pipeline wires WITHOUT latches
- Time-share a single port
- Not scalable beyond 2 ports
- Requires very careful array design to guarantee balanced paths
  - Second access cannot catch up with first access
- Short path constraint limits maximum clock period
  - Does not support CPU power states

Multi-banking or Interleaving



- Used in Intel Pentium (8 banks)
- Need routing network
- Must deal with bank conflicts
  - Bank conflicts not known till address generated
  - Difficult in non-data-capture machine with speculative scheduling
    - Replay – looks just like a cache miss
  - Sensitive to bank interleave: fine-grained vs. coarse-grained

Memory Data Flow

- Memory Data Flow Challenges
  - Memory Data Dependences
  - Load Bypassing
  - Load Forwarding
  - Speculative Disambiguation
  - The Memory Bottleneck
- Cache Hits and Cache Misses
- Replacement Policies
- Prefetching

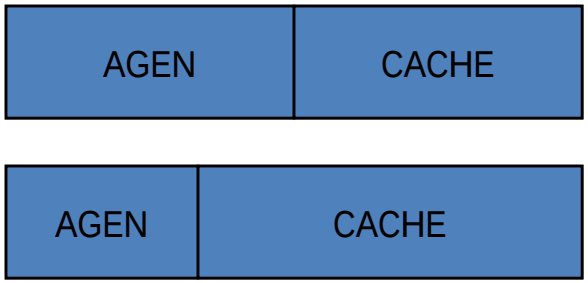
Caches and Performance

- 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
  - Typically 1-3 cycles for L1 cache
    - Intel/HP McKinley: 1 cycle
      - Heroic physical design of cache arrays, peripheral logic
      - Only register indirect addressing supported: load r1, (r2)
    - IBM Power4: 3 cycles
      - Address generation
      - Array access
      - Word select and align
      - Register file write (no bypass)

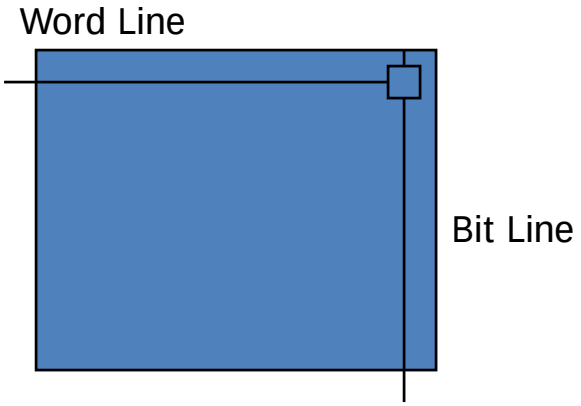
Cache Hit continued



- 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 Intel 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
  - DRC "flexibility"

Cache Misses and Performance

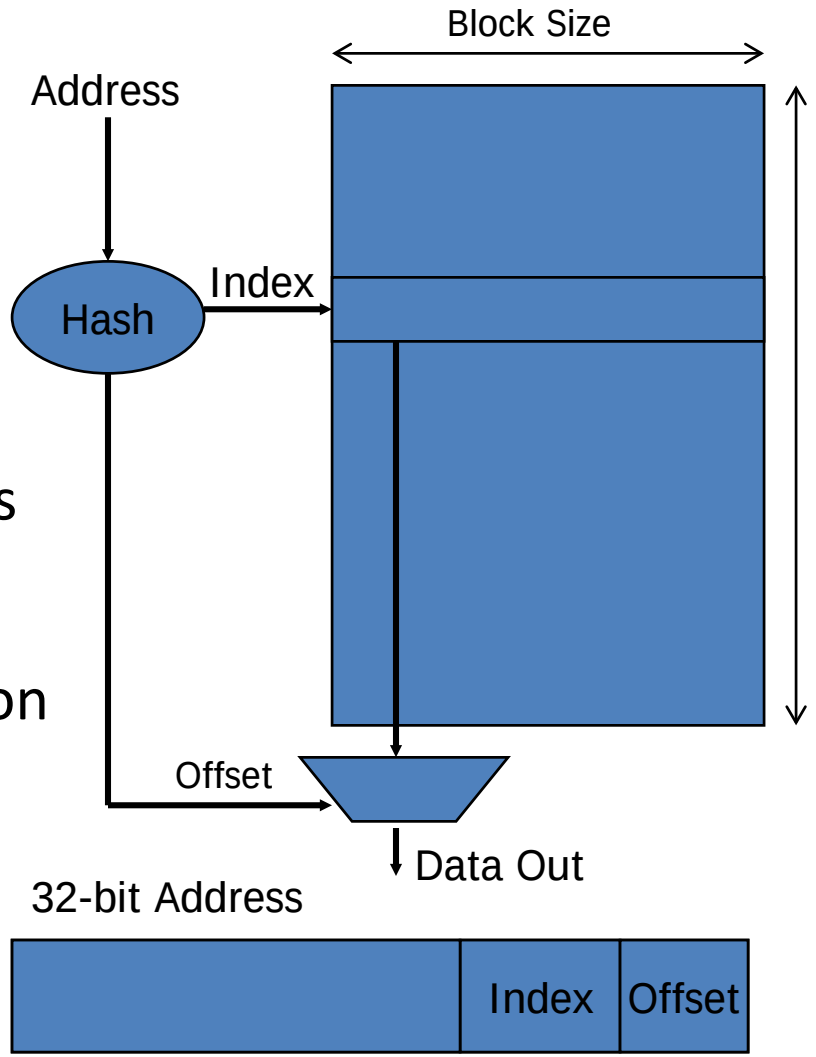
- Miss penalty
  - Detect miss: 1 or more cycles
  - Find victim (replace block): 1 or more cycles
    - Write back if dirty
  - Request block from next level: several cycles
    - May need to **find** line from one of many caches (coherence)
  - Transfer block from next level: several cycles
    - (block 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
    - Replacement policy

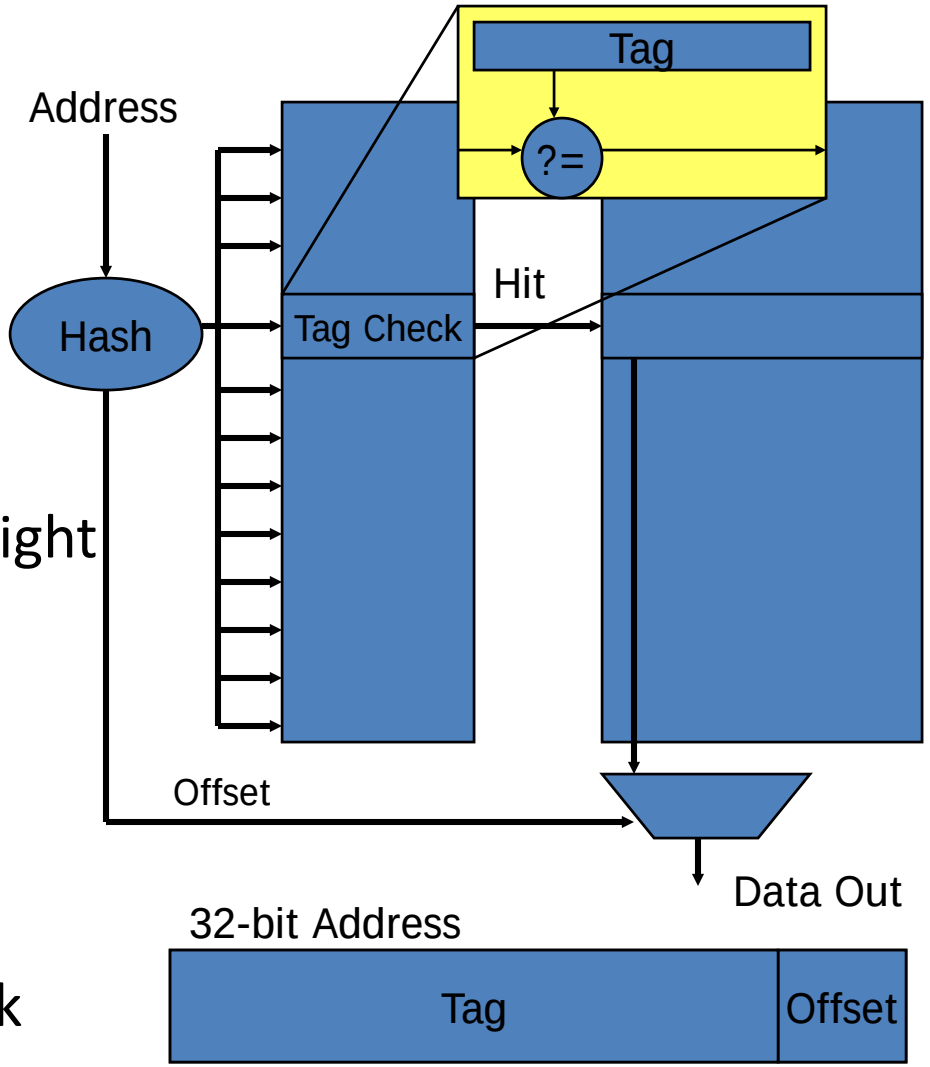
Review: Placement

- 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



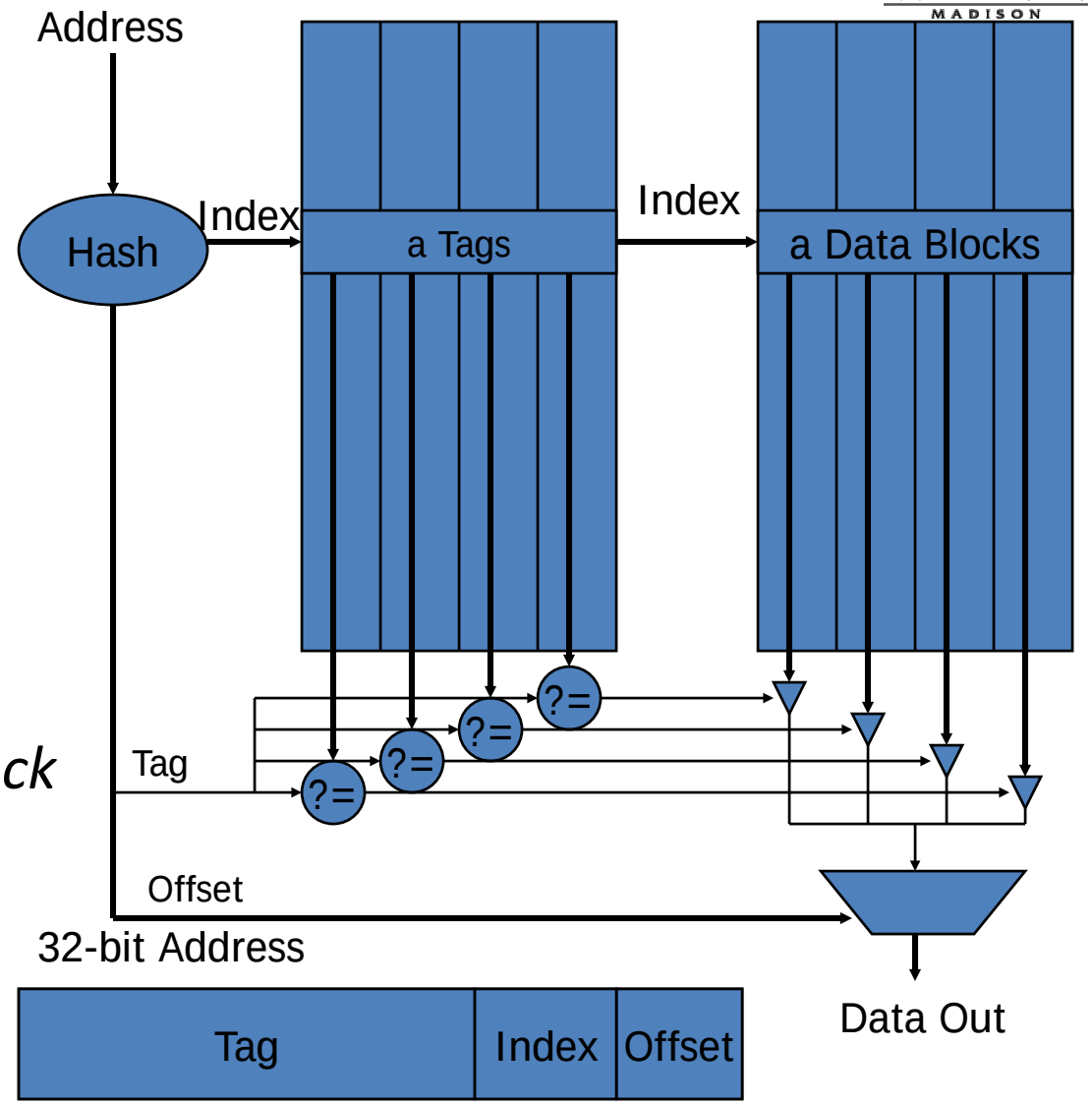
Review: Identification

- *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



Review: Placement

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



Memory Data Flow

- Memory Data Flow Challenges
  - Memory Data Dependences
  - Load Bypassing
  - Load Forwarding
  - Speculative Disambiguation
  - The Memory Bottleneck
- Cache Hits and Cache Misses
- Replacement Policies
- Prefetching

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?

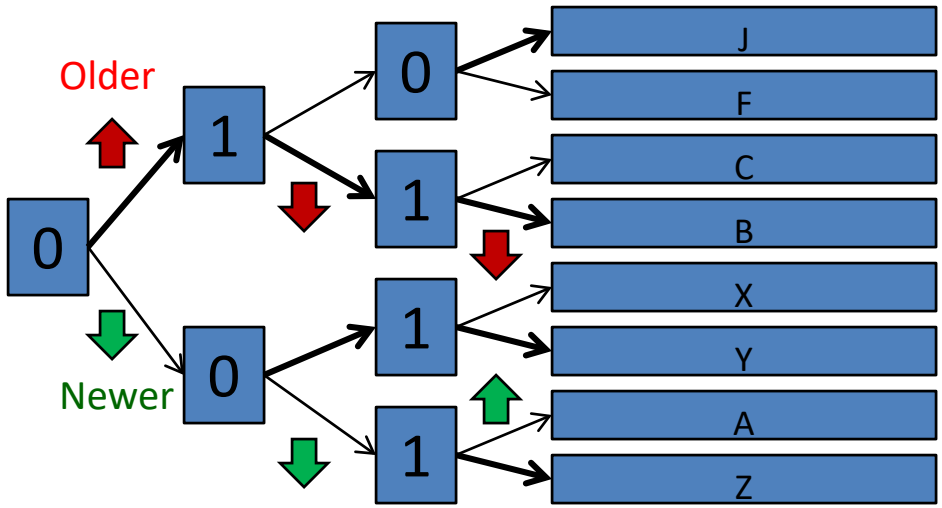
[Belady, IBM Systems Journal, 1966]

- 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
  - See [Jimenez MICRO '13]

Least-Recently Used

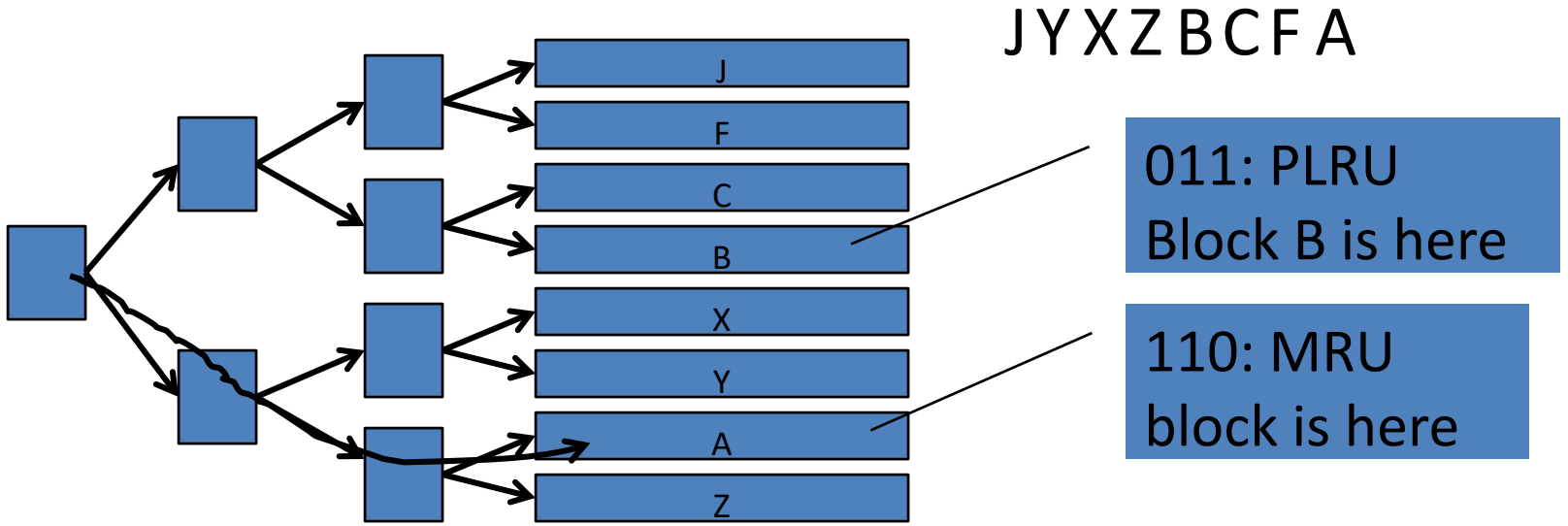
- For a=2, LRU is equivalent to NMRU
  - Single bit per set indicates LRU/MRU
  - Set/clear on each access
- For a>2, LRU is difficult/expensive
  - Timestamps? How many bits?
    - Must find min timestamp on each eviction
  - Sorted list? Re-sort on every access?
- List overhead: $log_2(a)$ bits /block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

Practical Pseudo-LRU

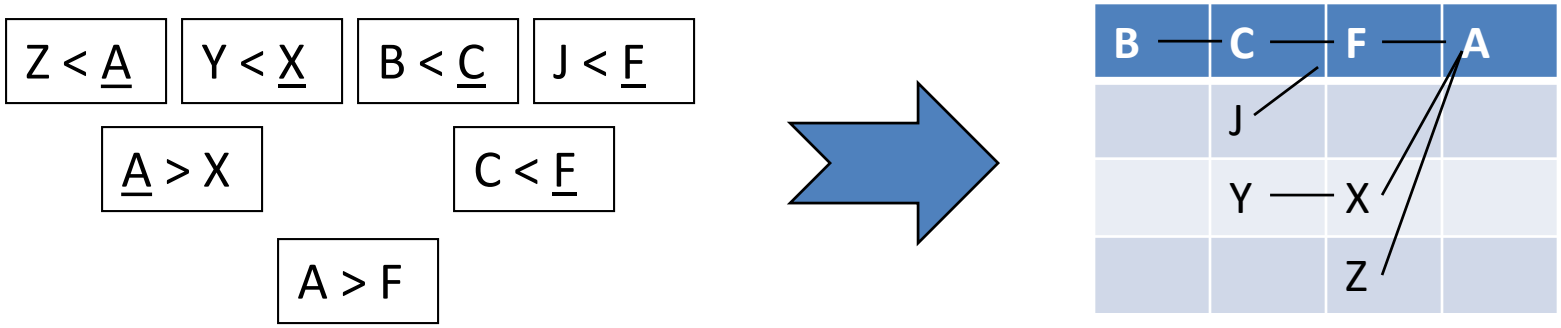


- Rather than true LRU, use binary tree
- Each node records which half is older/newer
- Update nodes on each reference
- Follow older pointers to find LRU victim

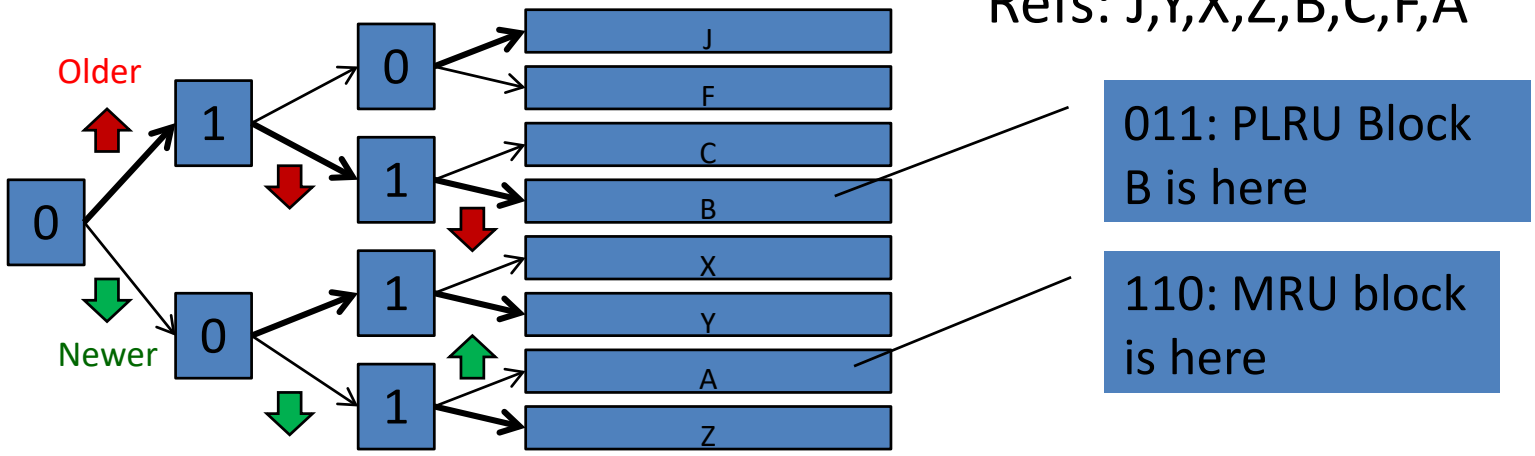
Practical Pseudo-LRU In Action



Partial Order Encoded in Tree:



Practical Pseudo-LRU



- 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
- Recently revisited [Jimenez MICRO-2013]

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

RRIP [Jaleel et al. ISCA 2010]

- Re-reference Interval Prediction
- Extends NRU to multiple bits
  - Start in the middle, promote on hit, demote over time
- Can predict *near-immediate*, *intermediate*, and *distant* re-reference
- Low overhead: 2 bits/block
- Static and dynamic variants (like LIP/DIP)
  - Set dueling

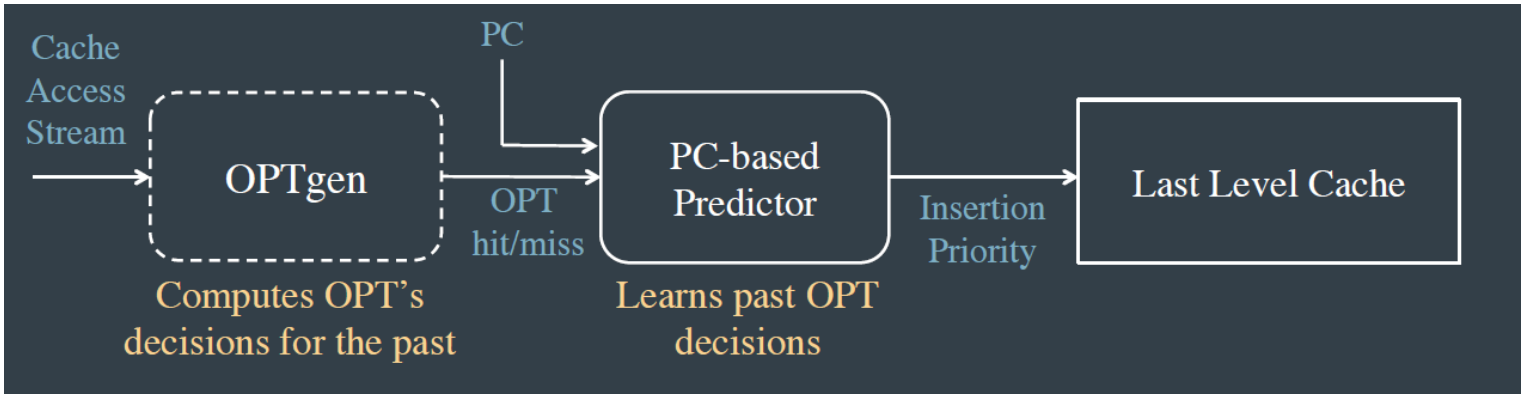
Least Frequently Used

- 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?

Cache Replacement Championship

- CRC-1 Held at ISCA 2010
  - http://www.jilp.org/jwac-1
  - Several variants, improvements
  - Simulation infrastructure
    - Implementations for all entries
- CRC-2 held at ISCA 2017
  - http://crc2.ece.tamu.edu
  - Several categories, each with different winner
  - Overall winner: Hawkeye (but close)

Hawkeye Replacement



[Jain, Lin, CRC-2]

- Based on Belady's OPT algorithm
  - Observe from the past
  - Train predictor
  - Apply predictor to present

Replacement Recap

- 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
  - RRIP (re-reference interval prediction)
  - Least frequently used (LFU)
- Championship contests

Replacement 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.

Aamer Jaleel, Kevin B. Theobald, Simon C. Steely Jr. , Joel Emer, “High Performance Cache Replacement Using Re-Reference Interval Prediction “, In ISCA, 2010.

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.

Replacement References

W. Lin et al. “Predicting last-touch references under optimal replacement.” Technical Report CSE-TR-447-02, U. of Michigan, 2002.

H. Liu et al. “Cache Bursts: A New Approach for Eliminating Dead Blocks and Increasing Cache Efficiency.” In Micro-41, 2008.

G. Loh. “Extending the Effectiveness of 3D-Stacked DRAM Caches with an Adaptive Multi-Queue Policy”. In Micro, 2009.

C.-K. Luk et al. Pin: building customized program analysis tools with dynamic instrumentation. In PLDI, pages 190–200, 2005.

N. Megiddo and D. S. Modha, “ARC: A self-tuning, low overhead replacement cache,” in FAST, 2003.

E. J. O’Neil et al. “The LRU-K page replacement algorithm for database disk buffering,” in Proc. ACM SIGMOD Conf., pp. 297–306, 1993.

M. Qureshi, A. Jaleel, Y. Patt, S. Steely, J. Emer. “Adaptive Insertion Policies for High Performance Caching”. In ISCA-34, 2007.

K. Rajan and G. Ramaswamy. “Emulating Optimal Replacement with a Shepherd Cache”. In Micro-40, 2007.

J. T. Robinson and M. V. Devarakonda, “Data cache management using frequency-based replacement,” in SIGMETRICS Conf, 1990.

R. Sugumar and S. Abraham, “Efficient simulation of caches under optimal replacement with applications to miss characterization,” in SIGMETRICS, 1993.

Y. Xie, G. Loh. “PIPP: Promotion/Insertion Pseudo-Partitioning of Multi-Core Shared Caches.” In ISCA-36, 2009

Y. Zhou and J. F. Philbin, “The multi-queue replacement algorithm for second level buffer caches,” in USENIX Annual Tech. Conf, 2001.

Memory Data Flow

- Memory Data Flow Challenges
  - Memory Data Dependences
  - Load Bypassing
  - Load Forwarding
  - Speculative Disambiguation
  - The Memory Bottleneck
- Cache Hits and Cache Misses
- Replacement Policies
- Prefetching

Prefetching

- Even “demand fetching” prefetches other words in block
  - Spatial locality
- Prefetching is useless
  - Unless a prefetch costs less than demand miss
- Ideally, prefetches should
  - Always get data before it is referenced
  - Never get data not used
  - Never prematurely replace data
  - Never interfere with other cache activity

Software Prefetching

- For example:

```
do j= 1, cols
  do ii = 1 to rows by BLOCK
    prefetch (&(x[ii,j])+BLOCK)    # prefetch one block ahead
     do i = ii to ii + BLOCK-1
       sum = sum + x[i,j]
```

- How many blocks ahead should we prefetch?
  - Affects timeliness of prefetches
  - Must be scaled based on miss latency

Hardware Prefetching

- What to prefetch
  - One block spatially ahead
  - N blocks spatially ahead
  - Based on observed stride, track/prefetch multiple strides
- Training hardware prefetcher
  - On every reference (expensive)
  - On every miss (information loss)
  - Misses at what level of cache?
  - Prefetchers at every level of cache?
- Pressure for nonblocking miss support (MSHRs)

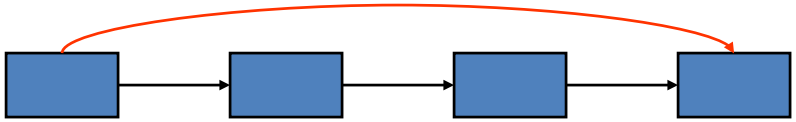
Prefetching for Pointer-based Data Structures

What to prefetch?

- Next level of tree: n+1, n+2, n+?
  - Entire tree? Or just one path
- Next node in linked list: n+1, n+2, n+?

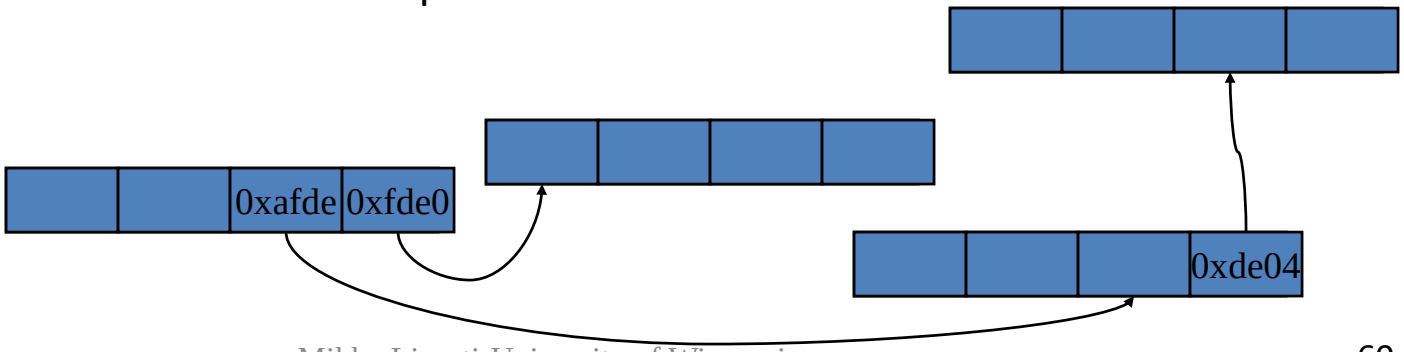
Jump-pointer prefetching [Roth, Sohi, ISCA 1999]

- Software places jump pointers in data structure



Content-driven data prefetching [Cooksey et al. ASPLOS 2002]

- Hardware scans blocks for pointers



Stream or Prefetch Buffers

- Prefetching causes capacity and conflict misses (pollution)
  - Can displace useful blocks
- Aimed at compulsory and capacity misses
- Prefetch into buffers, NOT into cache
  - On miss start filling stream buffer with successive lines
  - Check both cache and stream buffer
    - Hit in stream buffer => move line into cache (promote)
    - Miss in both => clear and refill stream buffer
- Performance
  - Very effective for I-caches, less for D-caches
  - Multiple buffers to capture multiple streams (better for D-caches)
- Can use with any prefetching scheme to avoid pollution

Case Study: Global History Buffer

[K. Nesbit, J. Smith, “Prefetching using a global history buffer”, HPCA 2004]

- Following slides © K. Nesbit, J. Smith

- Hardware prefetching scheme
- Monitors miss stream
- Learns correlations
- Issues prefetches for likely next address

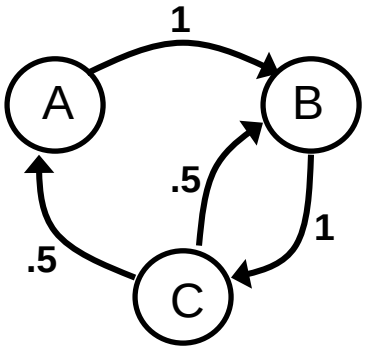
Markov Prefetching

- Markov prefetching forms address correlations
  - Joseph and Grunwald (ISCA '97)
- Uses global memory addresses as states in the Markov graph
- Correlation Table *approximates* Markov graph

Miss Address Stream

A B C A B C B C . . .

Markov Graph



Correlation Table

miss address	1st predict.	2nd predict.
A	B	
B	C	
C	B	A

Correlation Prefetching

- Distance Prefetching forms *delta* correlations
  - Kandiraju and Sivasubramaniam (ISCA '02)
- Delta-based prefetching leads to much smaller table than "classical" Markov Prefetching
- Delta-based prefetching can remove compulsory misses

Markov Prefetching

Miss Address Stream

27 28 29 27 28 29 28 29

Distance Prefetching

Global Delta Stream

1 1 -2 1 1 -1 1

miss address →

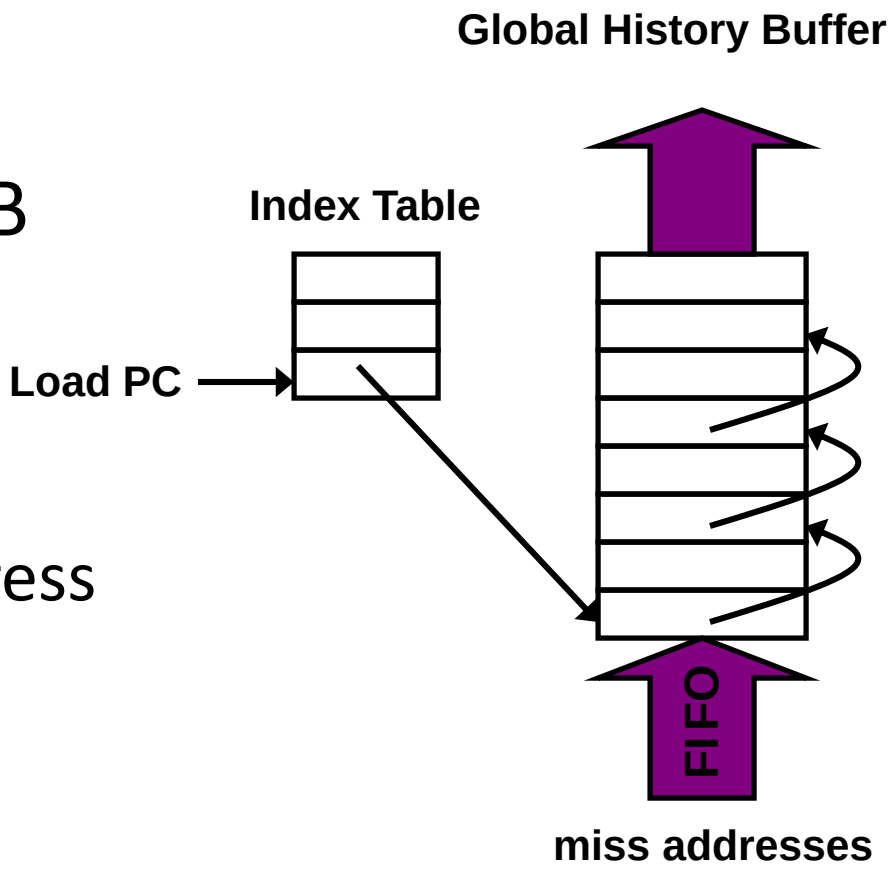
	1st predict.	2nd predict.
27	**28**	
28	**29**	
29	**28**	**29**

global delta →

	1st predict.	2nd predict.
-2	**1**	
-1	**1**	
1	**-1**	**-2**

Global History Buffer (GHB)

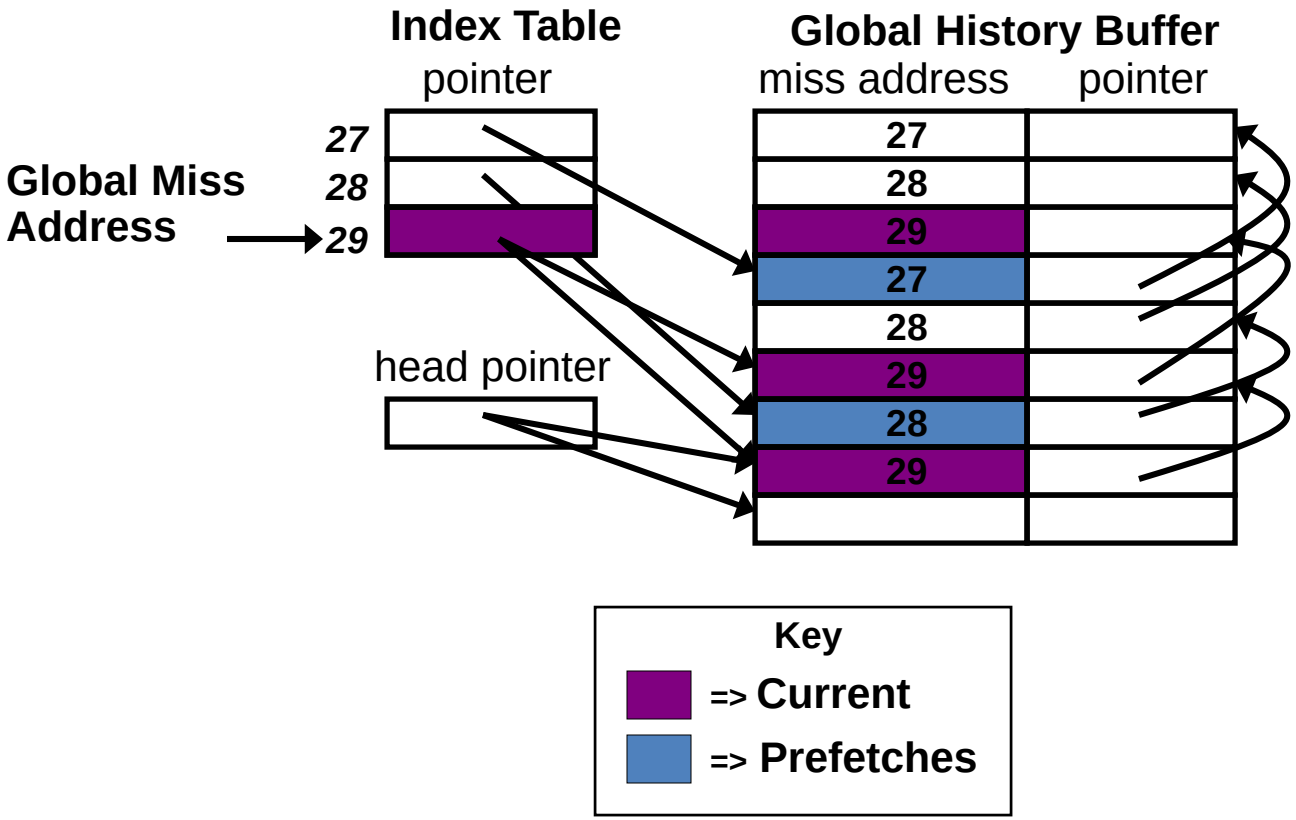
- Holds miss address history in FIFO order
- Linked lists within GHB connect related addresses
  - Same static load
  - Same global miss address
  - Same global delta
- Linked list walk is short compared with L2 miss latency



GHB - Example

Miss Address Stream

27 28 29 27 28 29 28 29



GHB – Deltas

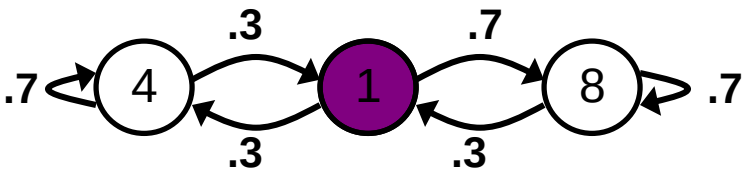
Miss Address Stream

27 28 36 44 45 49 53 54 62 70 71

Global Delta Stream

1 8 8 1 4 4 1 8 8 1

Markov Graph

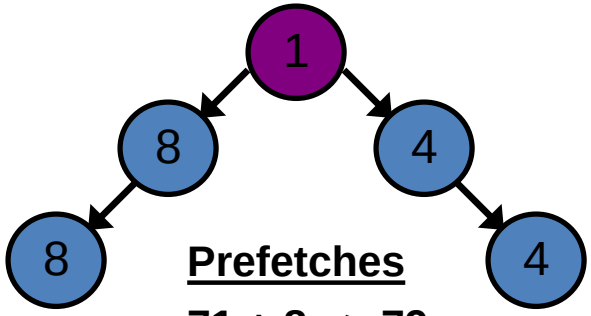


Key

=> Current

=> Prefetches

Depth



Prefetches

71 + 8 => 79

71 + 4 => 75

Prefetches

71 + 8 => 79

79 + 8 => 87

Prefetches

71 + 4 => 75

79 + 4 => 79

GHB – Hybrid Delta

- Width prefetching suffers from poor accuracy and short look-ahead
- Depth prefetching has good look-ahead, but may miss prefetch opportunities when a number of “next” addresses have similar probability
- The hybrid method combines depth and width

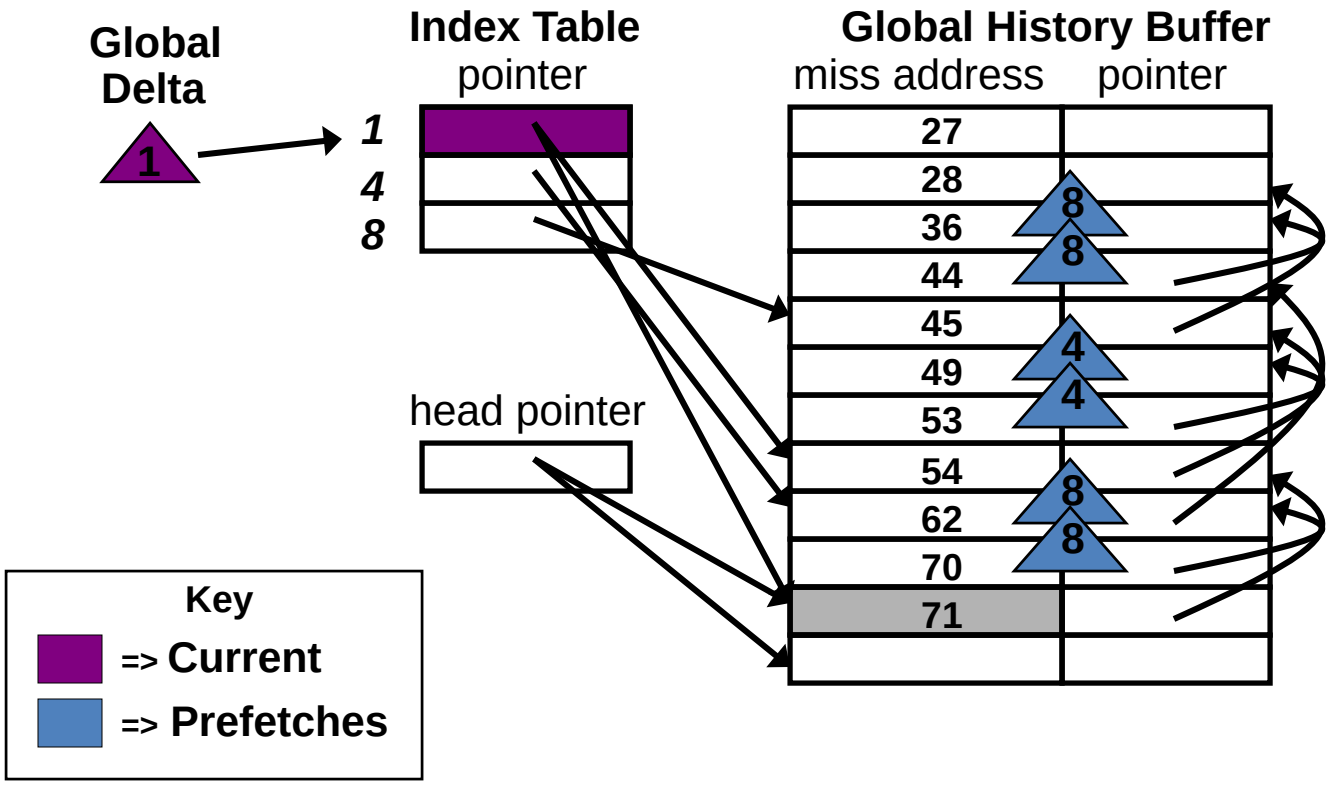
GHB - Hybrid Example

Miss Address Stream

27 28 36 44 45 49 53 54 62 70 **71**

Global Delta Stream

1 8 8 1 4 4 1 8 8 **1**



Prefetches

71 + 8 => 79

79 + 8 => 87

Prefetching Championships

- DPC-1 held at HPCA 2009
  - http://www.jilp.org/dpc
  - Winner: AMPM prefetching
    - Robust to out-of-order issue by capturing patterns instead of strides
- DPC-2 held at ISCA 2015
  - http://comparch-conf.gatech.edu/dpc2
  - Winner: Best-offset prefetcher
    - Based on ideas from Sandbox Prefetcher [Pugsley et al. HPCA 2014]
    - Considers prefetch timeliness
- Simulation infrastructure
  - Implementations for all entries

Prefetching Recap

- Prefetching anticipates future memory references
  - Software prefetching
  - Next-block, stride prefetching
  - Global history buffer prefetching
- Issues/challenges
  - Accuracy
  - Timeliness
  - Overhead (bandwidth)
  - Conflicts (displace useful data)

Summary

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