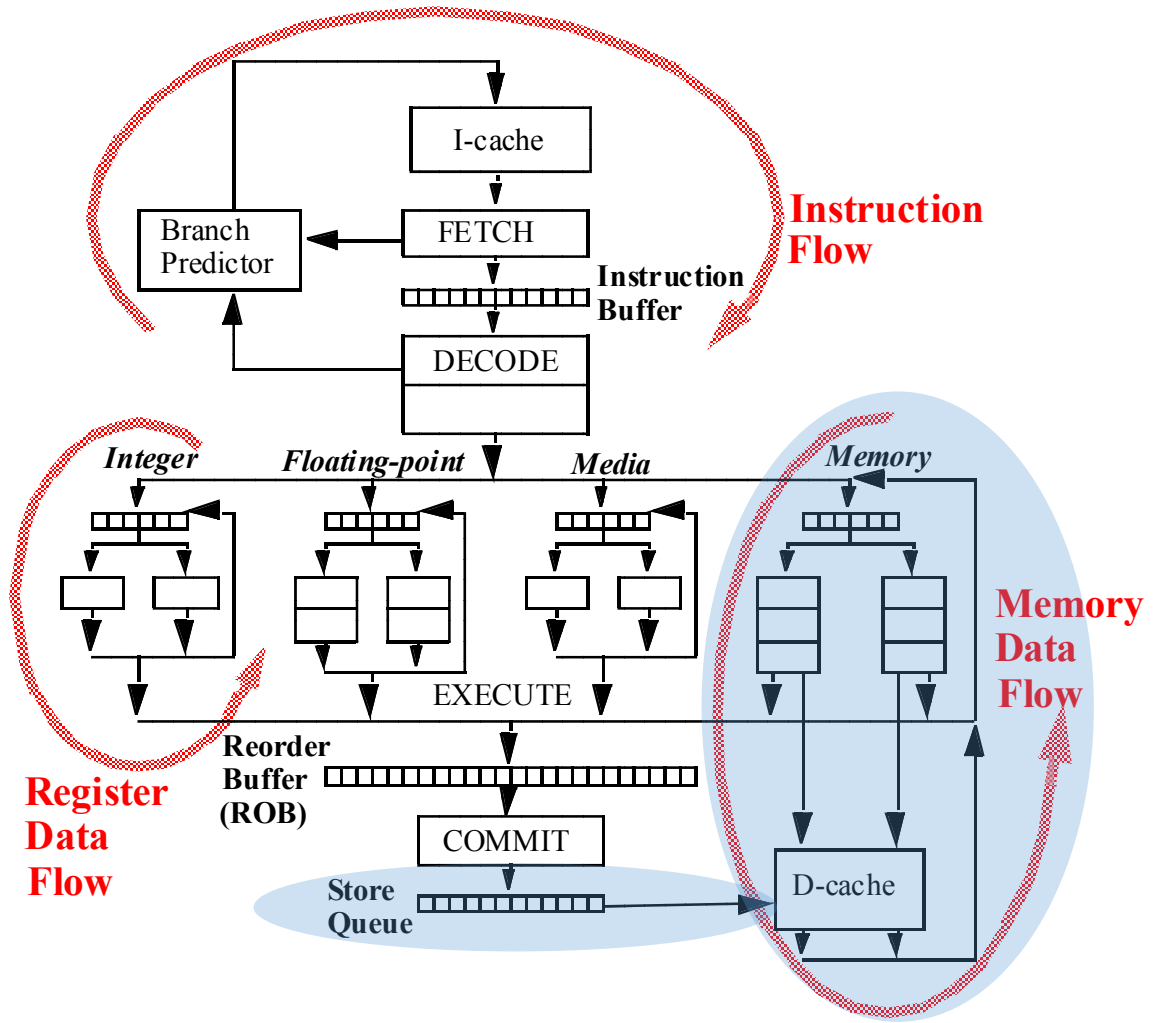


Memory Accesses in Out-of-Order Execution

Nima Honarmand

Big Picture

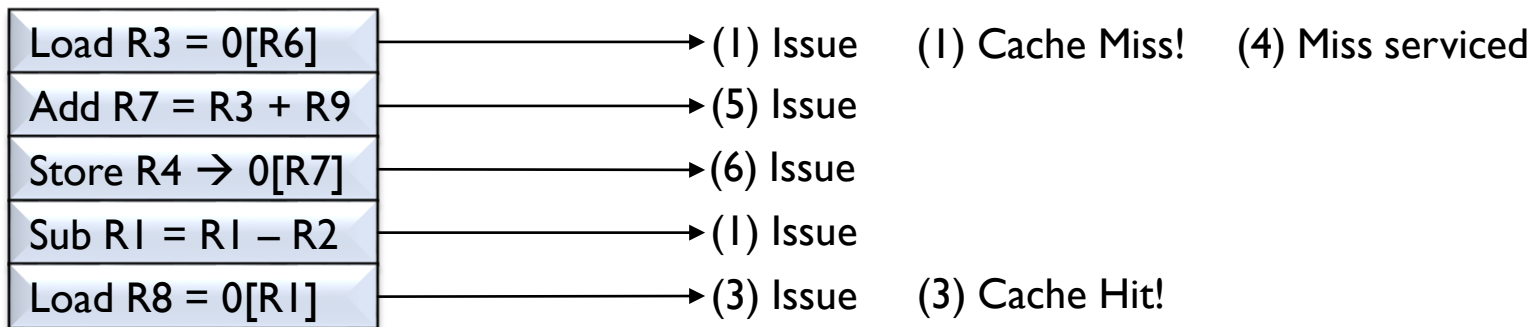


OoO and Memory Instructions

- Memory instructions benefit from out-of-order execution just like other ones
- Especially important to execute loads as soon as address is known
 - Loads are at the top of dependence chains
- To enable precise state recovery, stores are sent to D\$ after retirement
 - Sufficient to prevent wrong-branch-path stores
- Loads can be issued out-of-order w.r.t. other loads and stores if no dependence

OoO and Memory Instructions

- Memory insts have same 3 types of dependences as register-based insts
 - RAW (true), WAR and WAW (false)
- However, memory-based dependences are dynamic
 - Often not identifiable by looking at the instructions
 - Depend on program state (can change as the program executes)
 - Unlike register-based dependences



But there was a later load...

- $[R1] \neq [R7]$ -> Load and Store are **independent** -> **Correct** execution
- $[R1] == [R7]$ -> Load and Store are **dependent** -> **Incorrect** execution

Basic Concepts

- **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
 - Requires computing effective addresses of both memory references
- We say a memory op **is performed** when it is done in D\$
 - Loads perform in **Execute (X)** stage
 - Stores perform in **Retire (R)** stage

Scheme 1: In-Order Load/Stores

- Performs all loads/stores in-order with respect to each other
 - However, they can execute out of order with respect to other types of instructions
- Pessimistically, assuming dependence between all mem. ops

Load/Store Queue (LSQ)

- Operates as a circular FIFO
- Loads and store instructions are stored in program order
 - allocate on dispatch
 - de-allocate on retirement
- For each instruction, contains:
 - “Type”: Instruction type (S or L)
 - “Addr”: Memory addr
 - Addr is generated in dataflow order and copied to LSQ
 - “Val”: Data for stores
 - Val is generated in dataflow order and copied to LSQ
- You can think of LSQ as the RS for memory ops
 - i.e., each entry also contains tags and other RS stuff

Scheme 1: In-Order Load/Stores

- Only the instruction at the LSQ head can perform, if ready
 - If load, it can perform whenever ready
 - If store, it can perform if it is also at ROB head and ready
- Stores are held for all previous instructions
 - Since they perform in R stage
- Loads are only held for stores
- Easy to implement but killing most of OoO benefits
→ significant performance hit

Scheme 1 Pipeline

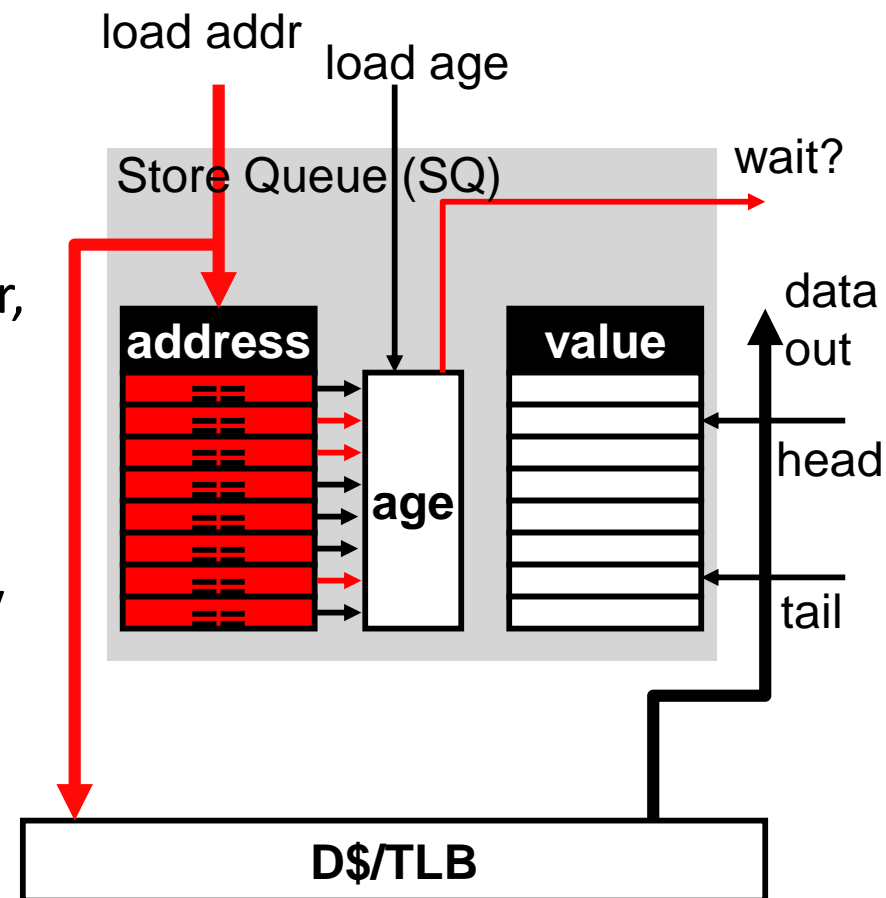
- Stores
 - **Dispatch (D)**
 - Allocate entry at LSQ tail
 - **Execute (X)**
 - Calculate and write address and data into corresponding LSQ slot
 - **Retire (R)**
 - Write address/data from LSQ head to D\$, free LSQ head
- Loads
 - **Dispatch (D)**
 - Allocate entry at LSQ tail
 - **Addr Gen (G)**
 - Calculate and write address into corresponding LSQ slot
 - **Execute (X)**
 - Send load to D\$ if at the head of LSQ
 - **Retire (R)**
 - Free LSQ head

Scheme 2: Load Bypassing

- Loads can be allowed to bypass older stores (if no aliasing)
 - Requires checking addresses of older stores
 - Addresses of older stores must be known in order to check
- To implement, use separate load queue (LQ) and store queue (SQ)
 - Think of separate RS for loads and stores
- Need to know the relative order of instructions in the queues
 - “Age”: new field added to both queues
 - A simple counter incremented during the in-order dispatch (for now)

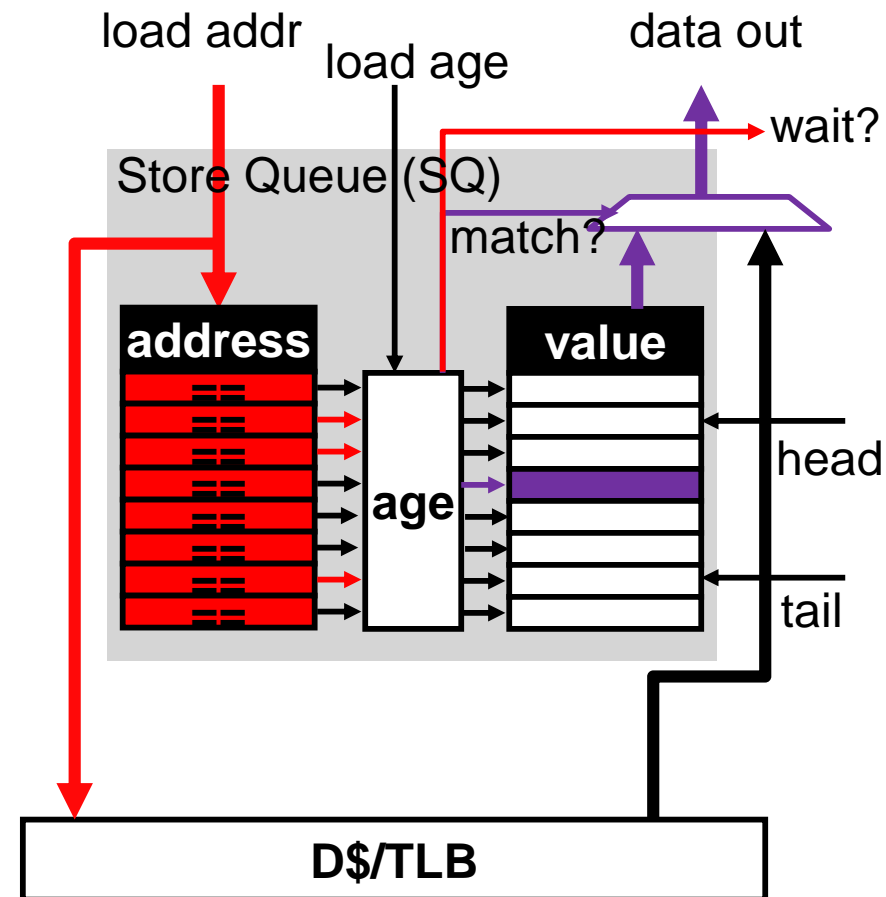
Scheme 2: Load Bypassing

- Loads: for the oldest ready load in LQ, check the addr of older stores in SQ
 - If any older stores with an *uncomputed or matching* addr, load cannot issue
 - Check SQ in parallel with accessing D\$
- Requires associative memory (CAM)
- Stores: can always execute when at ROB head



Scheme 3: Load Forwarding + Bypassing

- Loads: can be satisfied from the stores in the store queue on an address match
 - If the store data is available
- Avoids waiting until the store is sent to the cache
- Stores: can always execute when at ROB head



Schemes 2 & 3 Pipeline

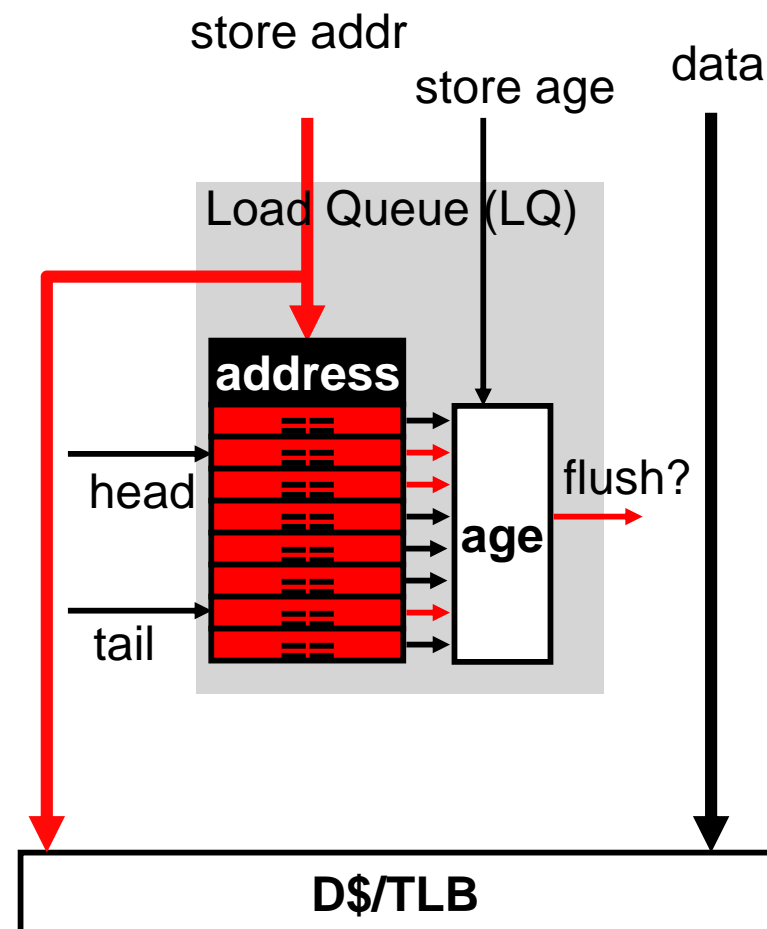
- Stores
 - **Dispatch (D)**
 - Allocate entry at SQ tail and record age
 - **Execute (X)**
 - Calculate and write address and data into corresponding SQ slot
 - **Retire (R)**
 - Write address/data from SQ head to D\$, free SQ head
- Loads
 - **Dispatch (D)**
 - Allocate entry at LQ tail and record age
 - **Addr Gen (G)**
 - Calculate and write address into corresponding LQ slot
 - **Execute (X)**
 - Send load to D\$ when D\$ available and check the SQ for aliasing stores
 - **Retire (R)**
 - Free LQ head

Scheme 4: Loads Execute When Ready

- Drawback of previous schemes:
 - Loads must wait for all older stores to compute their “Addr”
 - i.e., to “execute”
- Alternative: let the loads go ahead even if older stores exist with uncomputed “Addr”
 - Most aggressive scheme
- Greatest potential IPC: loads never stall
- A form of speculation: speculate that uncomputed stores are to other addresses
 - Relies on the fact that aliases are rare
 - **Potential for incorrect execution**
 - **Need to be able to “undo” bad loads (mis-speculations)**

Detecting Ordering Violations

- Case 1: Older store execs before younger load
 - No problem, HW from Scheme 3 takes care of this
- Case 2: Older store execs after younger load
 - Store scans all younger loads
 - Address match \rightarrow ordering violation
 - Requires associative search in LQ

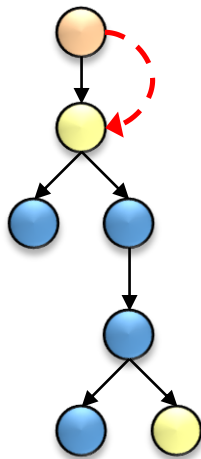


Scheme 4 Pipeline

- Stores
 - **Dispatch (D)**
 - Allocate entry at SQ tail and record age
 - **Execute (X)**
 - Calculate and write address and data into corresponding SQ slot
 - **Retire (R)**
 - Write address/data from SQ head to D\$, free SQ head
 - Check LQ for potential aliases, initiate “recovery” if necessary
- Loads
 - **Dispatch (D)**
 - Allocate entry at LQ tail and record age
 - **Addr Gen (G)**
 - Calculate and write address into corresponding LQ slot
 - **Execute (X)**
 - Send load to D\$ when D\$ available and check the SQ for aliasing stores
 - **Retire (R)**
 - Free LQ head

Dealing with Misspeculations

- Loads are not the only things which are wrong
 - Loads propagate wrong values to all their dependents
- These must somehow be re-executed



- Easiest: flush all instructions after (and including?) the misspeculated load, and just refetch
- Load uses forwarded value
- Correct value propagated when instructions re-execute

Flushing the pipeline has very high-overhead

Lowering Flush Overhead (1)

- ***Selective Re-execution***: re-execute only the dependent instructions
- Ideal case w.r.t. maintaining high IPC
 - No need to re-fetch/re-dispatch/re-rename/re-execute
- Very complicated
 - Need to hunt down only data-dependent instructions
 - Some bad instructions already executed (now in ROB)
 - Some bad instructions didn't execute yet (still in RS)
- Pentium 4 does something like this (called “replay”)

Lowering Flush Overhead (2)

- Observation: loads/stores that cause violations are “stable”
 - Dependences are mostly program based, program doesn’t change
- **Alias Prediction**: predict which load/store pairs are likely to alias
 - Use a hybrid scheme
 - Predict which loads, or load/store pairs will cause violations
 - Use Scheme 3 for those
 - Use Scheme 4 with pipeline flush for the rest