

Portland State University
ECE 587/687

Memory Ordering

Handling Memory Operations

- Review pipeline for out of order, superscalar processors
- To maximize ILP, loads/stores may execute out of order
- Memory operations require special handling
 - ◆ Compare: Register dependences identified at decode time
 - Allows early renaming to remove false dependences
 - Maximizes ILP
 - ◆ Memory dependences cannot be determined before execution since memory addresses need to be computed first
 - ◆ False dependences exist in memory execution stream
 - Multiple stores to the same bytes
 - Frequent due to stack pushes and pops

Handling Memory Operations (Cont.)

- Functions that a processor has to do for memory operations:
 - ◆ Enforcing memory dependences among loads and stores
 - ◆ Stores/Writes to memory need to be ordered and non-speculative
 - ◆ Stores issue to the data cache after retirement
- Loads and stores ordering is enforced using load and store buffers while allowing out of order execution
- Review Fig 11 of Smith & Sohi's paper

Store Buffer

- Store addresses are buffered in a queue
 - ◆ Entries allocated in program order at rename
- Store addresses remain buffered until:
 - ◆ Store data is available
 - ◆ Store instruction is retired in the reorder buffer
- New load addresses are checked with the waiting store addresses
 - ◆ If there is a match:
 - The load waits OR
 - Store data is bypassed to the matching load
 - ◆ If there is no match:
 - Load can execute speculatively
 - Load waits till all prior store addresses are computed

Store Buffer (Cont.)

- To match loads with store addresses, the store buffer is typically organized as a fully-associative queue
 - ◆ An address comparator in each buffer entry compares each store address to the address of a load issued to the data cache
 - ◆ Multiple stores to same address may be present
 - ◆ Load-store address match is qualified with “age” information to match a load to the last preceding store in program order
 - Age is typically checked by attaching the number of the preceding store entry to each load at rename
 - Effectively provides a form of memory renaming

Load Buffer

- Loads can speculatively issue to the data cache out-of-order
- But load issue may stall
 - ◆ Unavailable resources/ports in the data cache
 - ◆ Unknown store addresses
 - ◆ Delayed store data execution
 - ◆ Memory mapped I/O
 - ◆ Lock operations
 - ◆ Misaligned loads

Load Buffer (Cont.)

- Load buffer is provided for stalled loads to wait
- Load entries are allocated in program order at rename
- A stalled load simply waits in its buffer entry until stall condition is removed
 - ◆ Scheduling logic checks for awakened loads and re-issues them to the data cache, typically in program order

Load Buffer (Cont.)

- Load buffers are sometimes used for load-store dependence speculation
 - ◆ When a load issues ahead of a preceding store, it is impossible to perform address match
 - ◆ Option 1: Stall the load until all prior store addresses are computed
 - Significant performance impact in machines with deep, wide pipelines
 - ◆ Option 2: Speculate that the load does not depend on the previous unknown stores
 - Needs misprediction detection and recovery mechanism

Load Buffer (Cont.)

- One detection mechanism is to make the load buffer a fully associative queue
 - ◆ Store addresses are checked against all previously issued load addresses
 - ◆ Only younger loads need to be checked

Memory Consistency

- Load buffer snoops other processors stores to maintain memory consistency on some processors
- Stores from other threads on the same processor need also to be snooped in the load buffer to maintain memory consistency
- Memory consistency models define ordering requirements of loads and stores to different addresses and from multiple processors
 - ◆ More details in ECE 588/688
- Examples for memory consistency models
 - ◆ Sequential Consistency (SC)
 - ◆ Total Store Order (TSO)

Memory Dependence Prediction

- Memory Order Violation: A load is executed before a prior store, reads the wrong value
- False Dependence: Loads wait unnecessarily for stores to different addresses
- Goals of Memory Dependence Prediction:
 - ◆ Predict the load instructions that would cause a memory-order violation
 - ◆ Delay execution of these loads only as long as necessary to avoid violations

Memory Dependence Prediction

- Memory misprediction recovery is expensive
 - ◆ Requires a pipeline flush if a store address matches a younger issued load
 - ◆ Compare to branch mispredictions
- Memory dependence prediction minimizes such mispredictions
 - ◆ Issue younger loads if predictor predicts “no dependence”
 - ◆ Stall younger loads if predictor predicts “dependence”
 - ◆ With a good predictor, this approach minimizes unnecessary stalls (false dependences) as well as pipeline flushes from mispredictions

Simple Alternatives to Memory Dependence Prediction

- No Speculation
 - ◆ Issue for any load waits till prior stores have issued
- Naïve Speculation
 - ◆ Always issue and execute loads when their register dependences are satisfied, regardless of memory dependences
- Perfect Memory dependence prediction
 - ◆ Does not cause memory order violations
 - ◆ Avoids all false dependences
- Simulation Results in Table 3.1, Fig. 3.1

Store Sets

- Based on the assumption that future dependences can be predicted from past behavior
- Each load has a store set consisting of all stores upon which it has ever depended
 - ◆ Store is identified by its PC
- When program starts, all loads have empty store sets
- When a memory order violation happens, store is added to load's store set
- Example 5.1 in paper
- Results in Fig 5.1 and Fig 5.2
- Discuss 2-bit saturating counter (Fig 5.3, 5.4)

Store Set Implementation

- Figure 6.1 shows basic components
- Store Set Identifier Table (SSIT): PC-indexed, maintains store sets
- Last Fetched Store Table (LFST) maintains dynamic inst. count about most recently fetched store for each store set
- Limitations:
 - ◆ Store PCs exist in one store set at a time
 - ◆ Two loads depending on the same store can share a store set
 - ◆ All stores in a store set are executed in order

Implementation Details

- Recently fetched loads
 - ◆ Access SSIT based on their PC, get their SSID
 - ◆ If SSID is valid, LFST is accessed to get most recent store in the load's store set
- Recently fetched stores
 - ◆ Access SSIT based on their PC
 - ◆ If SSID is valid, then store belongs to a valid store set
 - Access LFST to get most recently fetched store information in its store set
 - Update LFST inserting its own dynamic inst. count since it is now the last fetched store in that store set
 - After store is issued, it invalidates the LFST entry if it refers to itself to ensure loads & stores are only dependent on stores that haven't been issued

Store Set Assignment

- Destructive interference happens because stores can belong to only one store set
 - ◆ Store set merging avoids the problem
- When a store-load pair causes a memory order violation:
 - ◆ If neither has been assigned a store set, a store set is allocated and assigned to both instructions
 - ◆ If load has been assigned a store set but the store hasn't, the store is assigned the load's store set
 - ◆ If store has been assigned a store set but the load hasn't, the load is assigned the store's load set
 - ◆ If both have store sets, one of them is declared the winner, and the instruction belonging to the loser's store set is assigned the winner's store set
- Results for store set merging in Fig 6.2

Reading Assignment

- D. Kroft, “Lockup-Free Instruction Fetch/Prefetch Cache Organization,” ISCA 1981 (Review)
- M.D. Hill and A.J. Smith, “Evaluating Associativity in CPU Caches,” IEEE Transactions on Computers, 1989 (Skim)