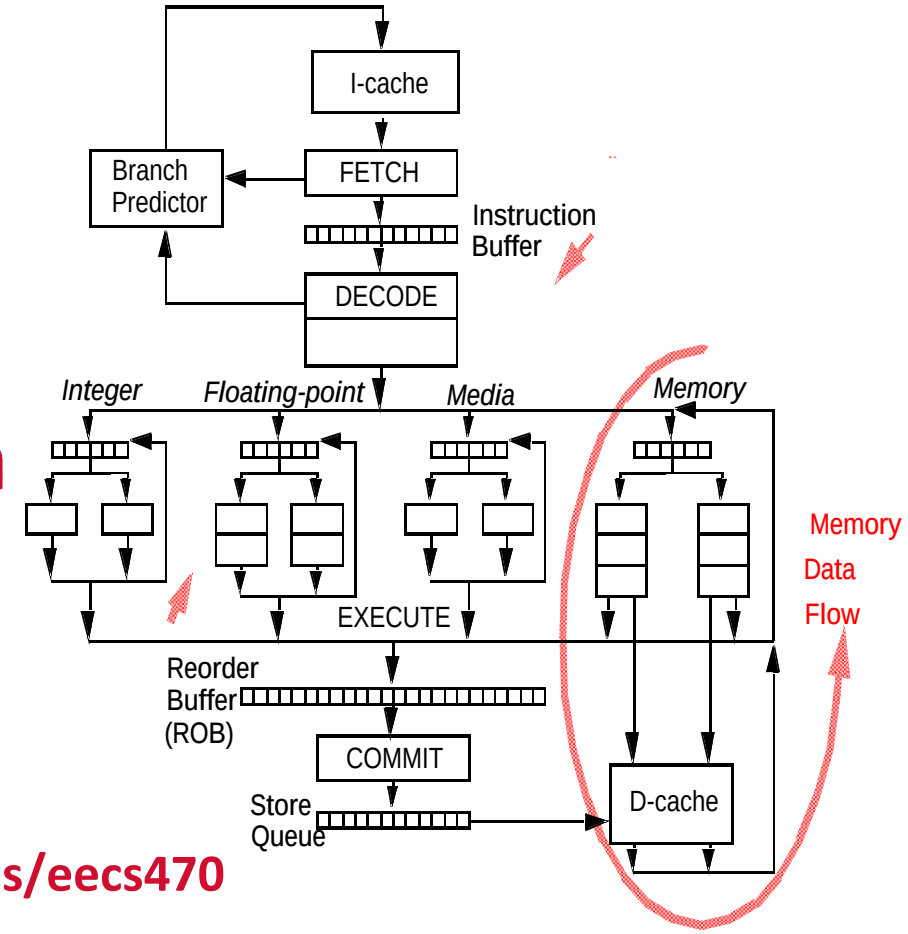
EECS 470

Lecture 12

Memory Speculation



Fall 2007

Prof. Thomas Wenisch

http://www.eecs.umich.edu/courses/eecs470

Slides developed in part by Profs. Austin, Brehob, Falsafi, Hill, Hoe, Lipasti, Martin, Roth, Shen, Smith, Sohi, Tyson, and Vijaykumar of Carnegie Mellon University, Purdue University, University of Michigan, Univerity of Pennsylvania, and University of Wisconsin.

Announcements

HW # 4 (deadline pushed to 11/2)

- Will be posted by Wednesday

Milestone 1 (due 10/29)

- Create handin/milestone1 in your svn repository
- `make syn` and `make test` should do something
- I will review these after 10/29 to provide feedback

Readings

For Wednesday:

- *H&P C.1-C.2,5.1-5.2*

Exam Statistics

	Q1	Q2	Q3	Q4	Q5	Q6	Tot.
Out of	16	16	18	15	15	20	100
Mean	13.8	14.6	12.3	10.5	12.3	14.4	78
Median	14	16	14	11	14	16	80.5
Std. Dev	1.8	2.6	4.6	2.8	3.4	5.3	12

Course policies (from syllabus)

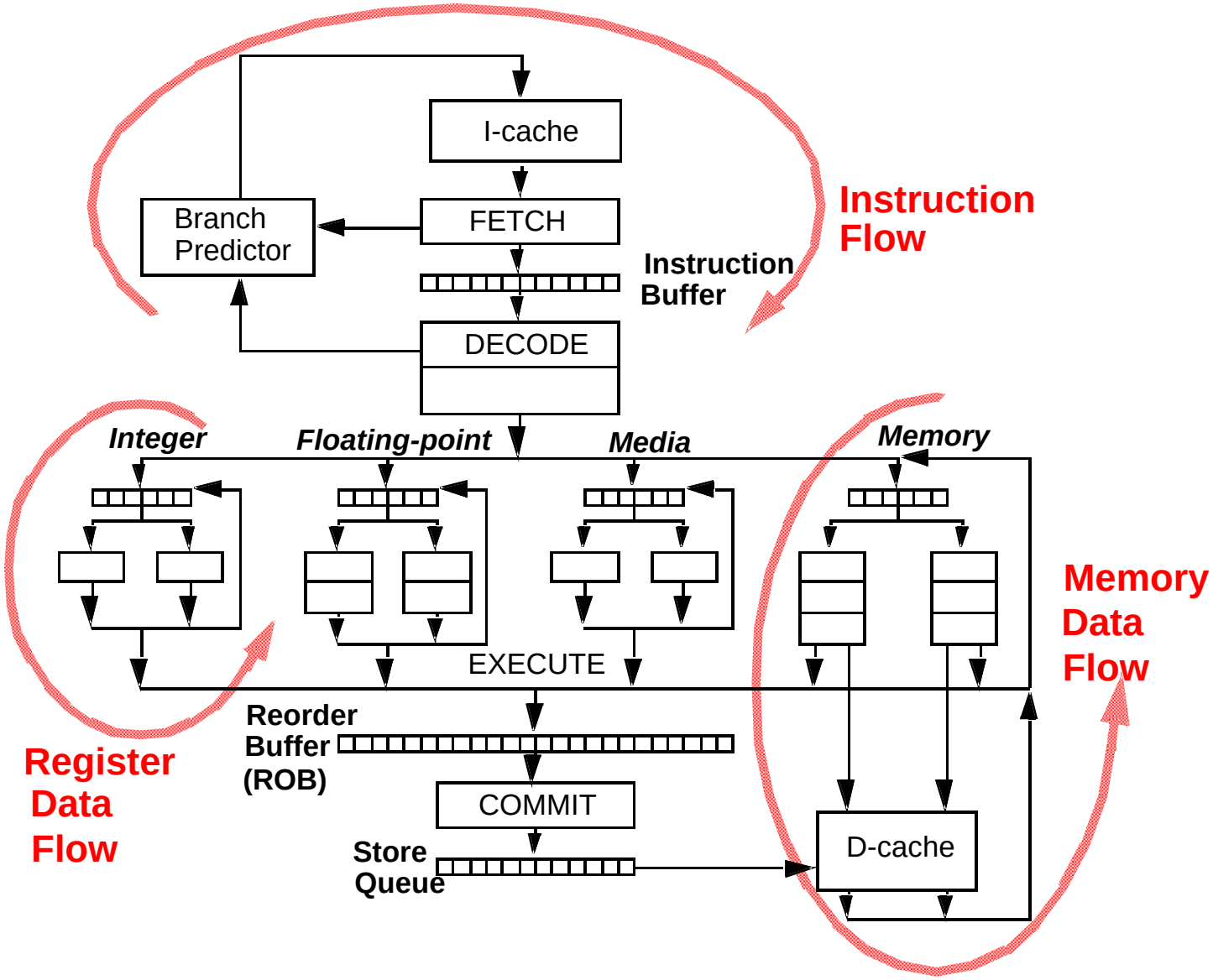
- You must pass both projects and exams to pass course
- Rough guide for passing (C): ~2 std.dev. under median

With assignments figured in, overall course stats:

- Mean: 83%, Median: 84%, Std. Dev: 10%

60% of your grade is still undetermined

Memory Dataflow Techniques



Out of Order Memory Operations

All insns are easy in out-of-order…

- Register inputs only
- Register renaming captures all dependences
- Tags tell you exactly when you can execute

… except loads

- Register and memory inputs (older stores)
- Register renaming does not tell you all dependences
- How do loads find older in-flight stores to same address (if any)?

Dynamic Reordering of Memory Operations

Storing to memory irrevocably in-order state
Hence, hold stores untill retire (ROB head)

No memory WAW or WAR

Allow OoO Loads that don't have RAW memory-dependence

What is hard about managing memory-dependence?

- memory address are much wider than reg names
- memory dependencies are not static
  - a load (or store) instruction's address can change
  - addresses need to be calculated and translated first
- memory instructions take longer to execute

Uniprocessor Load and Store Semantics

Given $Store_i(a, v) << Load_j(a)$ *("<<" means precedes)*

Load(*a*) must return *v* if there does not exist another $Store_k$ such that

$$Store_i(a, v) << Store_k(a, v') << Load_j(a)$$

This can be guaranteed by observing data dependence

- RAW Store(*a*, *v*) followed by Load(*a*)
- WAW Store(*a*, *v'*) followed by Store(*a*, *v*)
- WAR Load(*a*) followed by Store(*a*, *v'*)

For a uniprocessor, do we need to worry about loads and stores to different addresses? What about SMPs?

Some trivial ways to handle Loads

Allow only one load or store in OoO core

- Stall other operations at dispatch – very slow
- No need for LSQ

Load may only issue when LSQ head

- Stall other operations at dispatch
- Loads always get value from cache, only 1 outstanding load

More aggressive options:

- Load to store forwarding
- *Speculative* load-to-store forwarding – requires rewind mechanism

Several hardware realizations

- Unified LSQ (easier to understand, but nasty hardware)
- Separate LQ and SQ (more complicated, but elegant)

HW #1: Unified Load/Store Queue

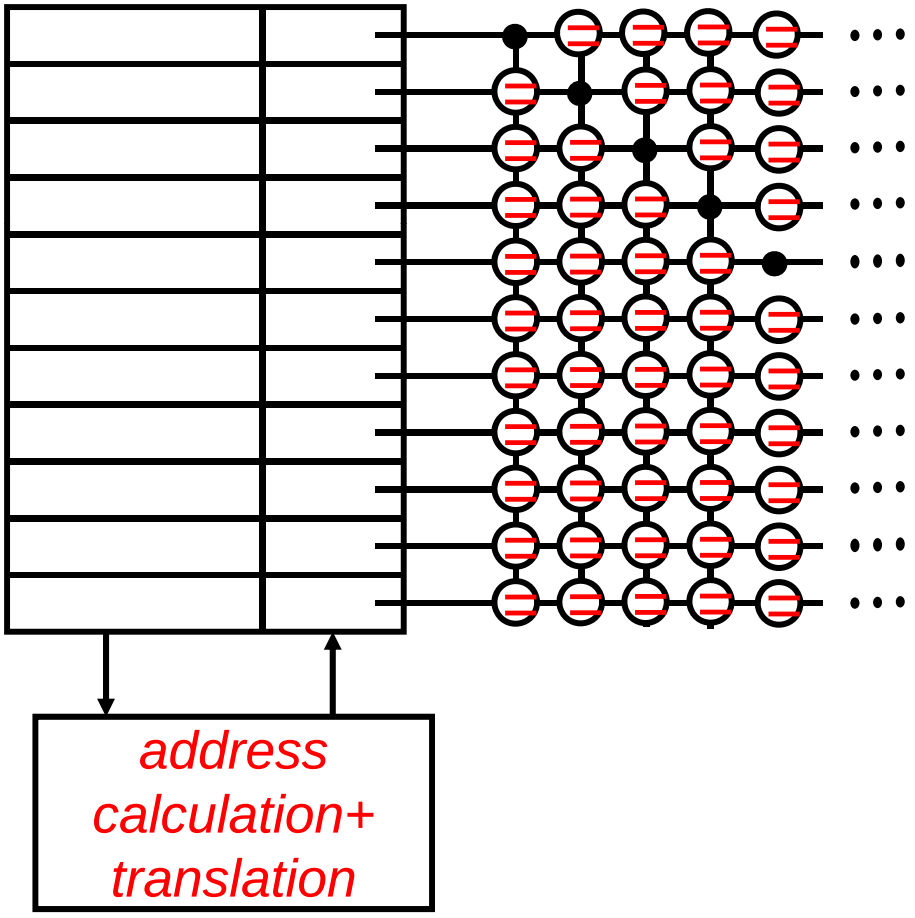
Operates as a circular FIFO

- allocate on dispatch
- de-allocate on retirement

Calc address in register dataflow order

A NxN comparator matrix detects memory address dependence (also considers relative age of entries)

- store ops are held until retirement
- load ops are issued when no dependency exists & all older store addresses known

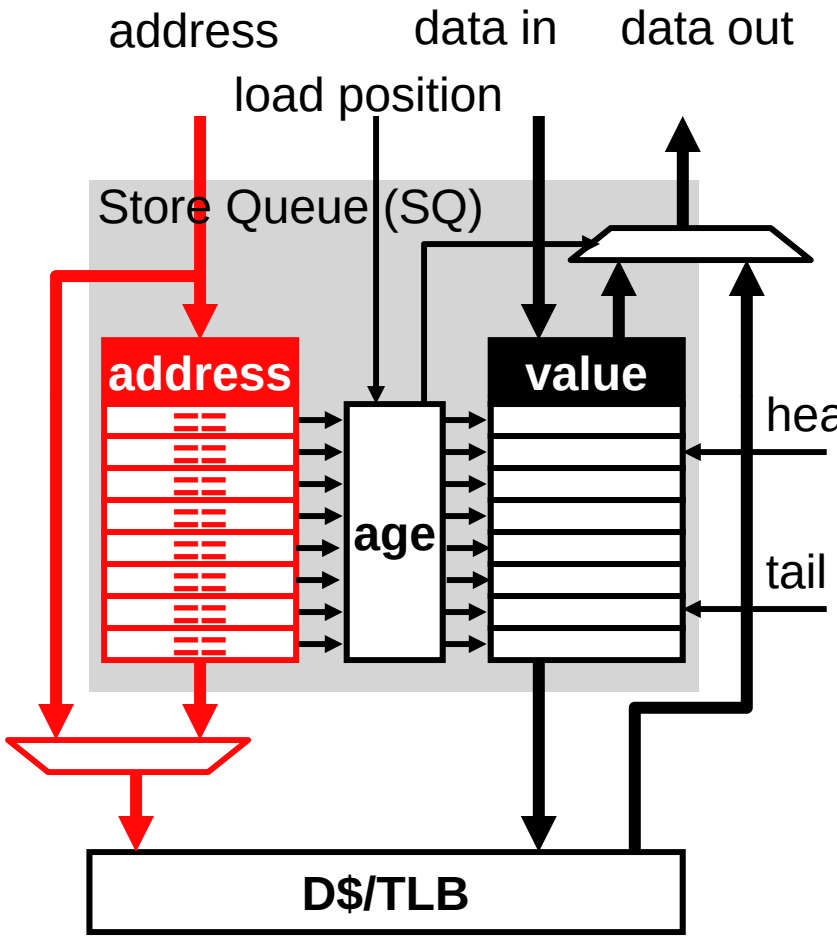


HW#2: Split Queues

D$/TLB + structures to handle in-flight loads/stores

- Performs four functions
- **In-order store retirement**
  - Writes stores to D$ in order
  - Basic, implemented by store queue (SQ)
- **Store-load forwarding**
  - Allows loads to read values from older un-retired stores
  - Also basic, also implemented by store queue (SQ)
- **Memory ordering violation detection**
  - Checks load speculation (more later)
  - Advanced, implemented by load queue (LQ)
- **Memory ordering violation avoidance**
  - Advanced, implemented by dependence predictors

Simple Data Memory FU: D$/TLB + SQ



Just like any other FU

- 2 register inputs (addr, data in)
- 1 register output (data out)
- 1 non-register input (load pos)?

Store queue (SQ)

- In-flight store address/value
- In program order (like ROB)
- Addresses associatively searchable
- Size heuristic: 15-20% of ROB

But what does it do?

Data Memory FU "Pipeline"

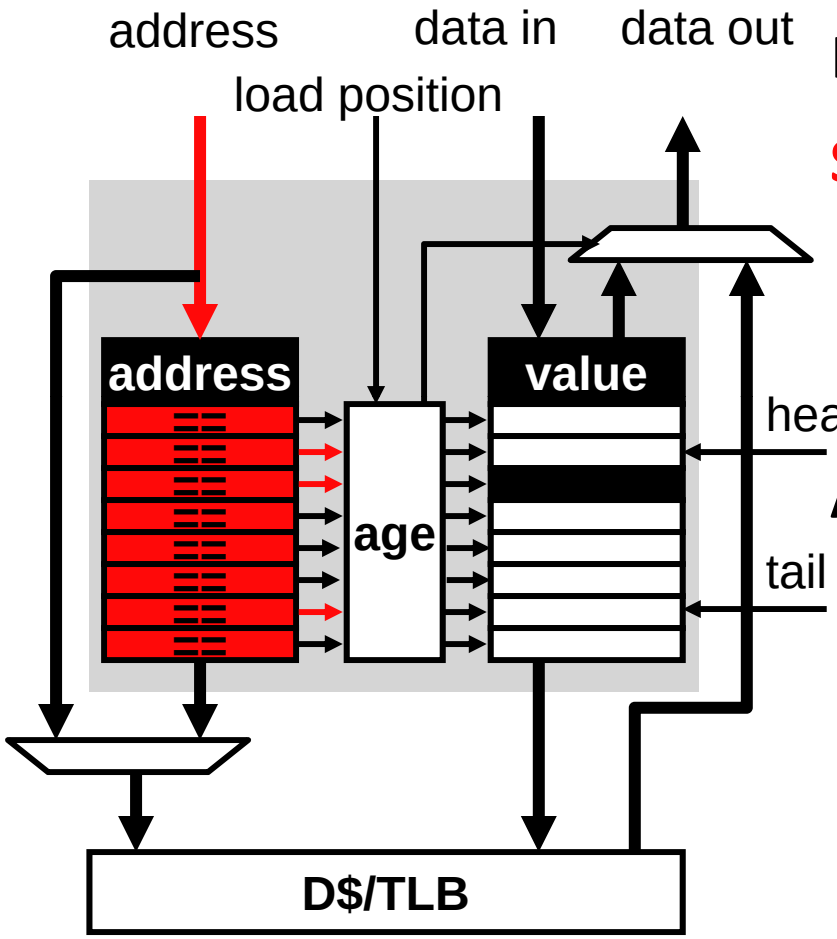
Stores

- **Dispatch (D)**
  - Allocate entry at SQ tail
- **Execute (X)**
  - Write address and data into corresponding SQ slot
- **Retire (R)**
  - Write address/data from SQ head to D$, free SQ head

Loads

- **Dispatch (D)**
  - Record current SQ tail as "load position"
- **Execute (X)**
  - Where the good stuff happens

"Out-of-Order" Load Execution



In parallel with D$ access

Send address to SQ

- Compare with all store addresses
- CAM: like FA$, or RS tag match
- Select all matching addresses

Age logic selects youngest store that is older than load

- Uses load position input
- Any? load **"forwards"** value from SQ
- None? Load gets value from D$

Conservative Load Scheduling

Why “” in “out-of-order”?

- + Load can execute out-of-order with respect to (wrt) other loads
- + Stores can execute out-of-order wrt other stores
- – **Loads must execute in-order wrt older stores**
  - ○ Load execution requires knowledge of all older store addresses
  - + Simple
  - – Restricts performance
- ❐ Used in P6

Opportunistic Memory Scheduling

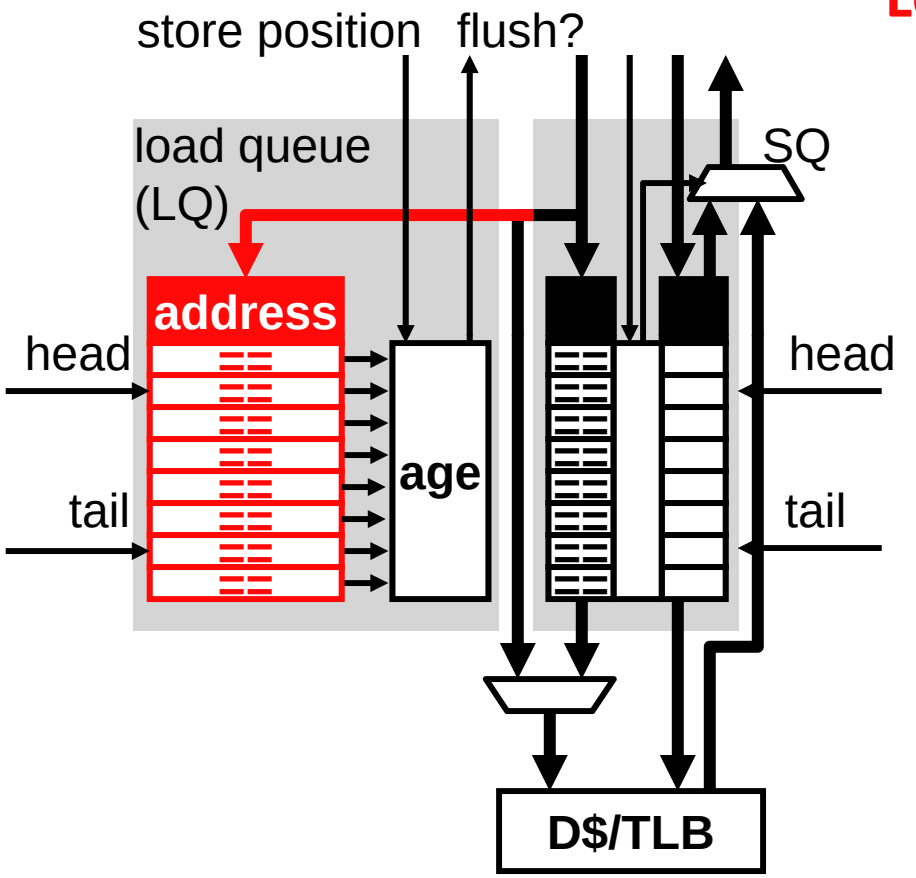
Observe: on average, < 10% of loads forward from SQ

- Even if older store address is unknown, chances are it won't match
- Let loads execute in presence of older **"ambiguous stores"**
  - Increases performance
- But what if ambiguous store *does* match?

Memory ordering violation: load executed too early

- Must detect…
- And fix (e.g., by flushing/refetching insns starting at load)

D$/TLB + SQ + LQ



Load queue (LQ)

- In-flight load addresses
- In program-order (like ROB,SQ)
- Associatively searchable
- Size heuristic: 20-30% of ROB

Advanced Memory "Pipeline" (LQ Only)

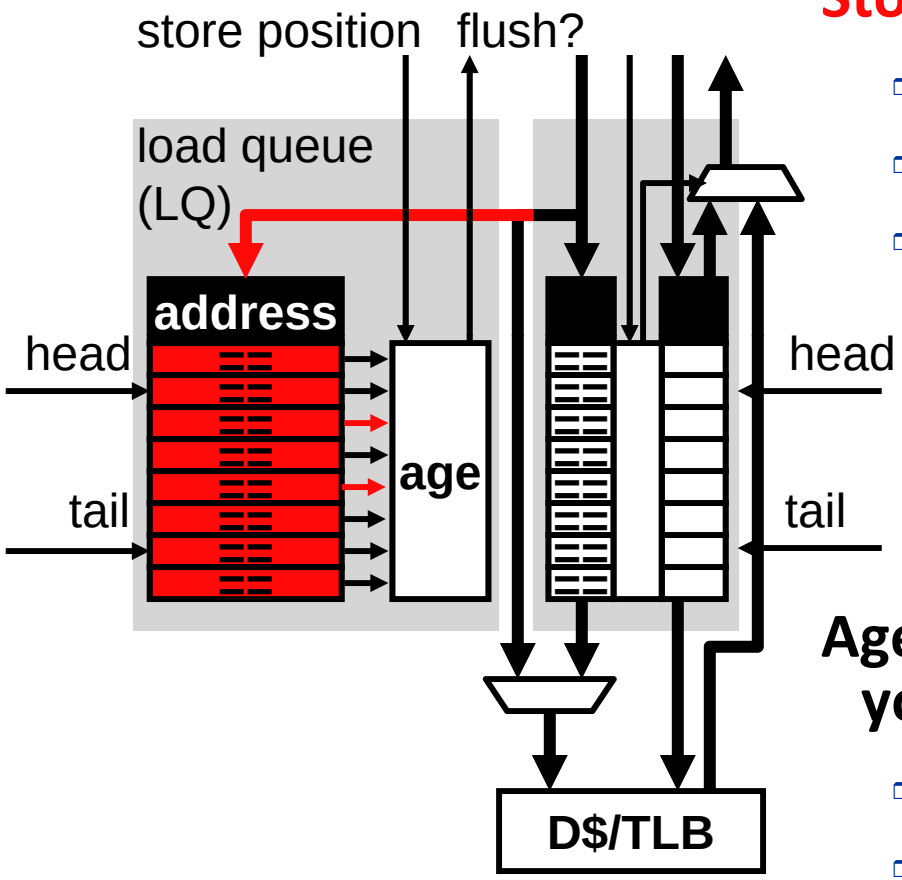
Loads

- **Dispatch (D)**
  - Allocate entry at LQ tail
- **Execute (X)**
  - Write address into corresponding LQ slot

Stores

- **Dispatch (D)**
  - Record current LQ tail as "store position"
- **Execute (X)**
  - Where the good stuff happens

Detecting Memory Ordering Violations



Store sends address to LQ

- Compare with all load addresses
- Selecting matching addresses
- Matching address?
  - Load executed before store
  - Violation
  - Fix!

Age logic selects oldest load that is younger than store

- Use store position
- Processor flushes and restarts

Intelligent Load Scheduling

Opportunistic scheduling better than conservative…

- Avoids many unnecessary delays

…but not significantly

- Introduces a few flushes, but each is much costlier than a delay

Observe: loads/stores that cause violations are "stable"

- Dependences are mostly program based, program doesn't change
- Scheduler is deterministic

Exploit: **intelligent load scheduling**

- Hybridize conservative and opportunistic
- Predict which loads, or load/store pairs will cause violations
- Use conservative scheduling for those, opportunistic for the rest

Memory Dependence Prediction

Store-blind prediction

- Predict load only, wait for all older stores to execute
- ± Simple, but a little too heavy handed
- Example: Alpha 21264

Store-load pair prediction

- Predict load/store pair, wait only for one store to execute
- ± More complex, but minimizes delay
- Example: Store-Sets
  - Load identifies the right dynamic store in two steps
  - Store-Set Table: load-PC → store-PC
  - Last Store Table: store-PC → SQ index of most recent instance
- Implemented in next Pentium? (guess)

Summary

Modern dynamic scheduling must support precise state

- A software sanity issue, not a performance issue

Strategy: Writeback → Complete (OoO) + Retire (iO)

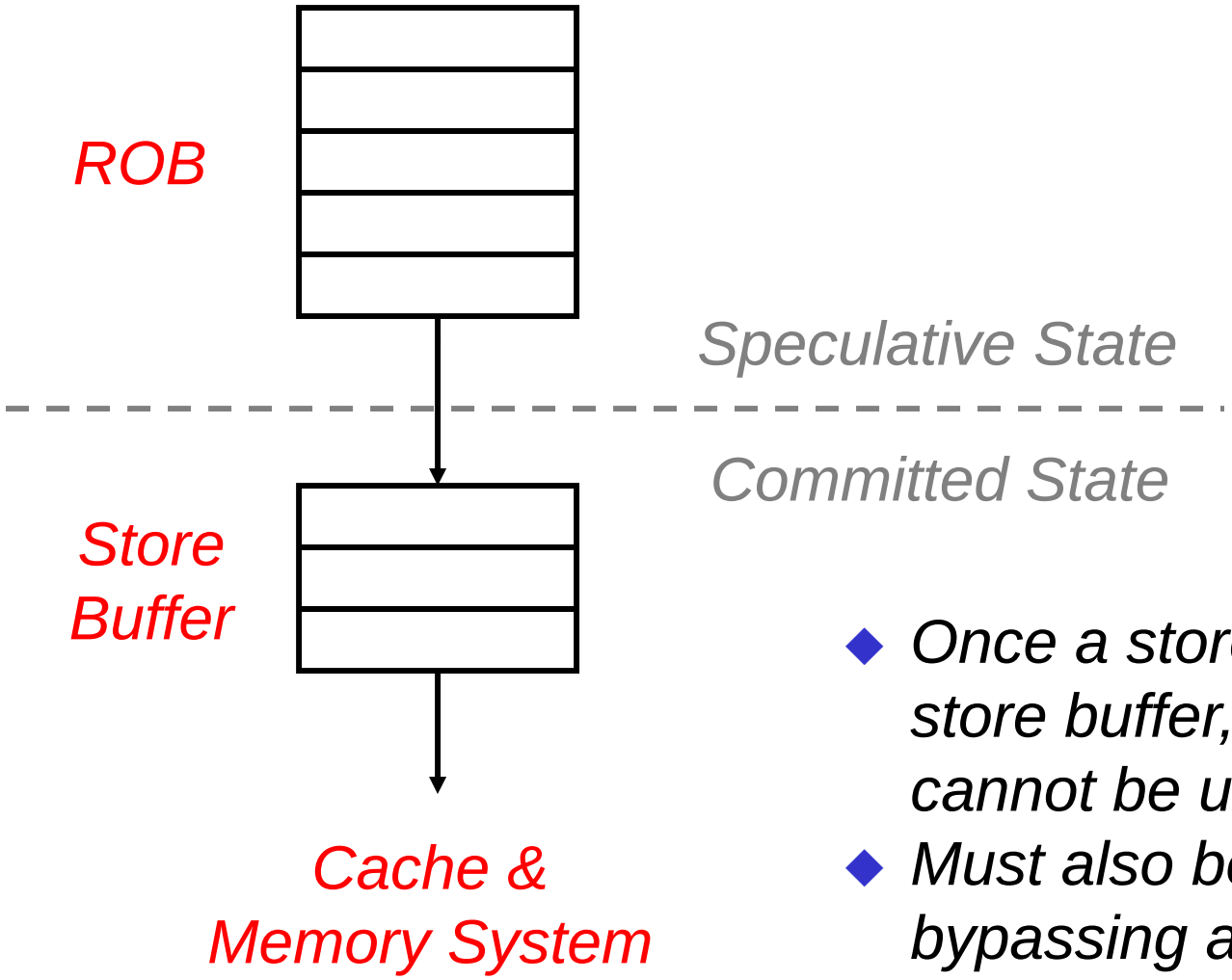
Two basic designs

- P6: Tomasulo + re-order buffer, copy based register renaming
  - Precise state is simple, but fast implementations are difficult
- R10K: implements true register renaming
  - Easier fast implementations, but precise state is more complex

Out-of-order memory operations

- Store queue: conservative load scheduling (iO wrt older stores)
- Load queue: opportunistic load scheduling (OoO wrt older stores)
- Intelligent memory scheduling: hybrid

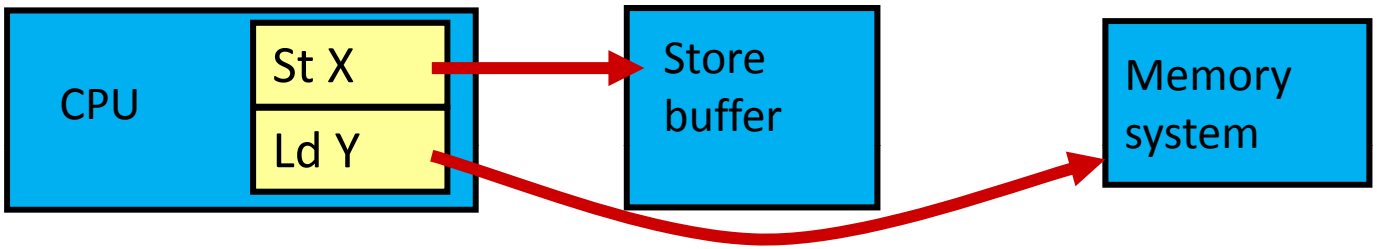
Store Buffer



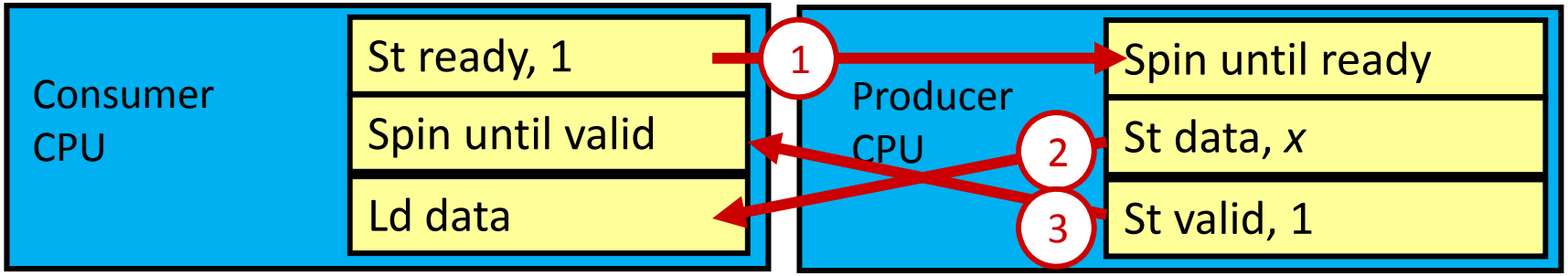
- *Once a store enters the store buffer, its effect cannot be undone*
- *Must also be check by load bypassing and forwarding*

Memory Ordering for Shared Memory Multiprocessors

Uniprocessors: Never wait to write memory

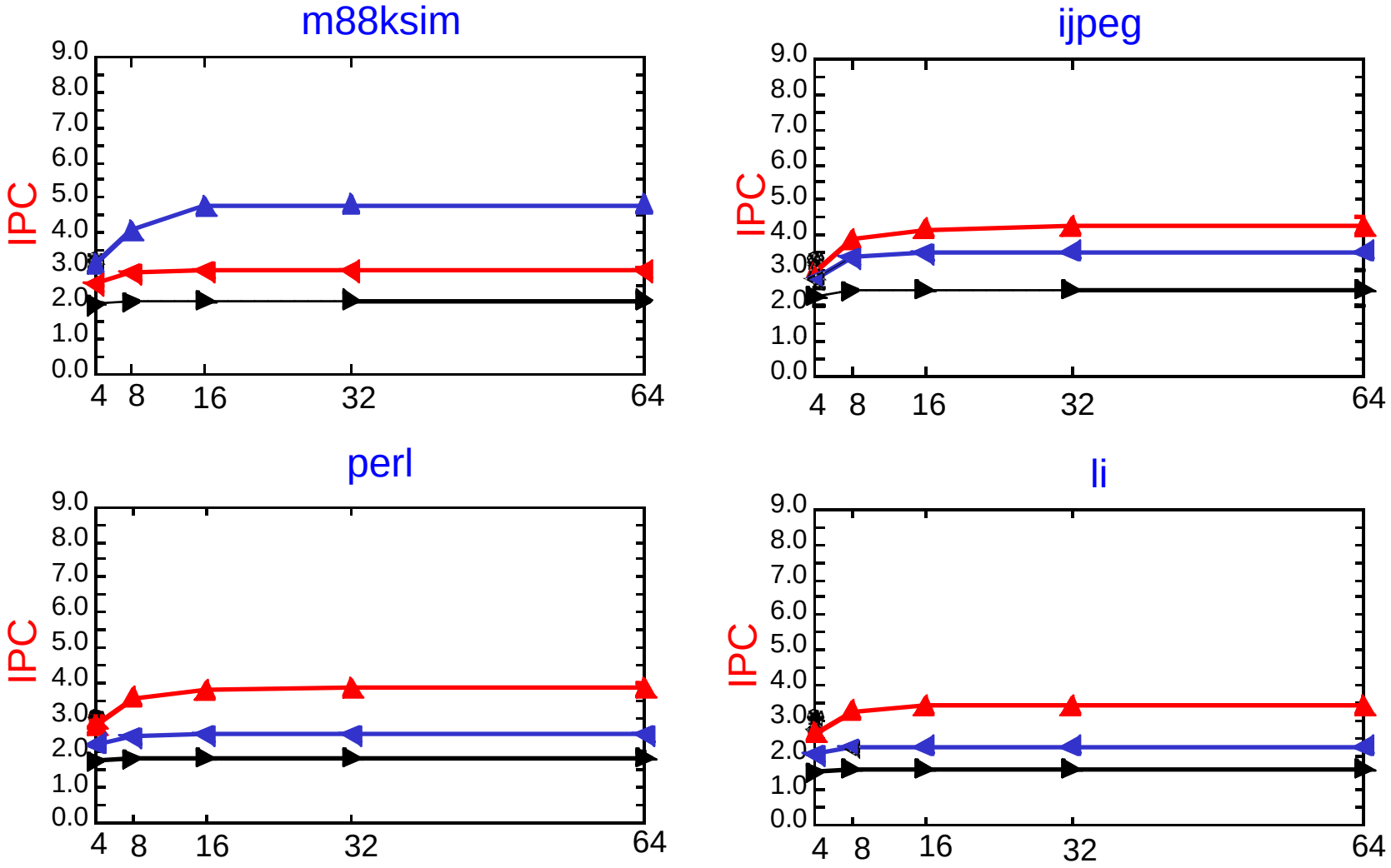


Multiprocessors: Must wait - order matters!



Much more on multiprocessor memory ordering later

Dataflow Limit on Superscalar Micros



Cycles before dispatch: ▲1 ◄2 ►3

Assume infinite functional units

Breaking Dataflow Dependence:

Prediction and Speculation

Branch prediction:

- Branch target history
- Branch direction history

Load value prediction:

- Value history for each static load

Register value prediction:

- Source or destination value history per static instruction operand

Assumes a very large transistor budget

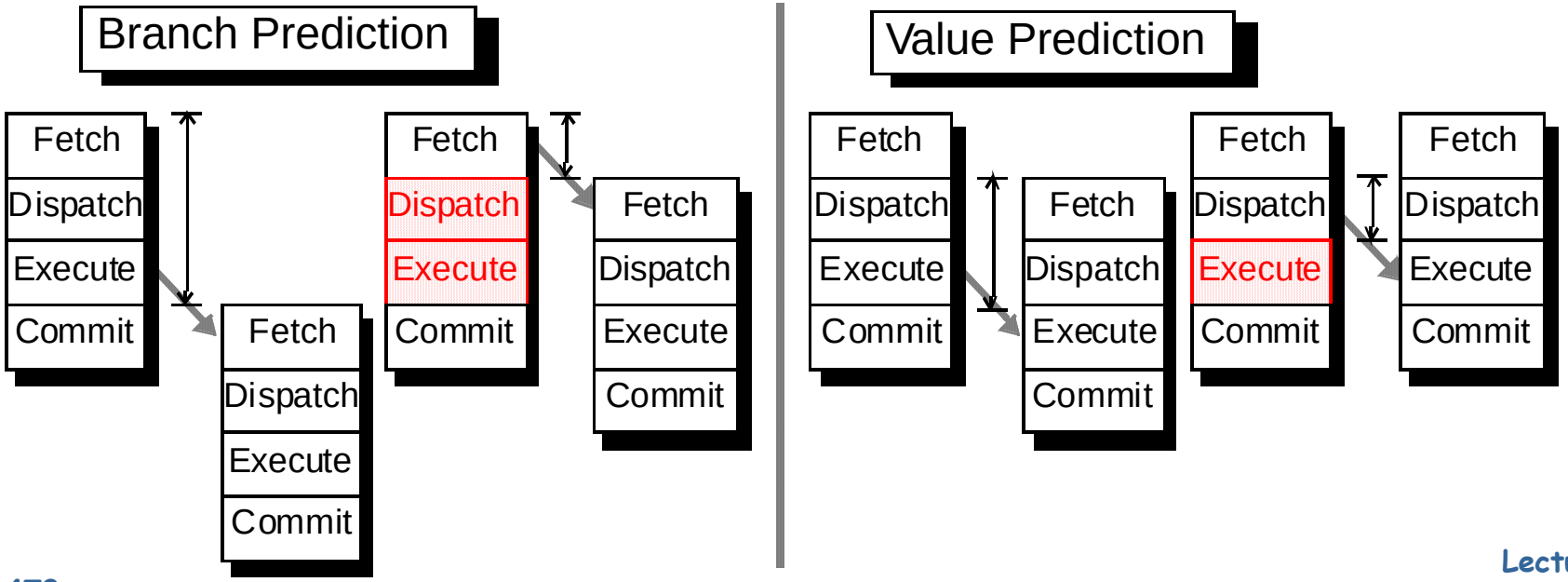
1. Large complicated prediction logic for accuracy

2. Spare resources to spend on speculated computation

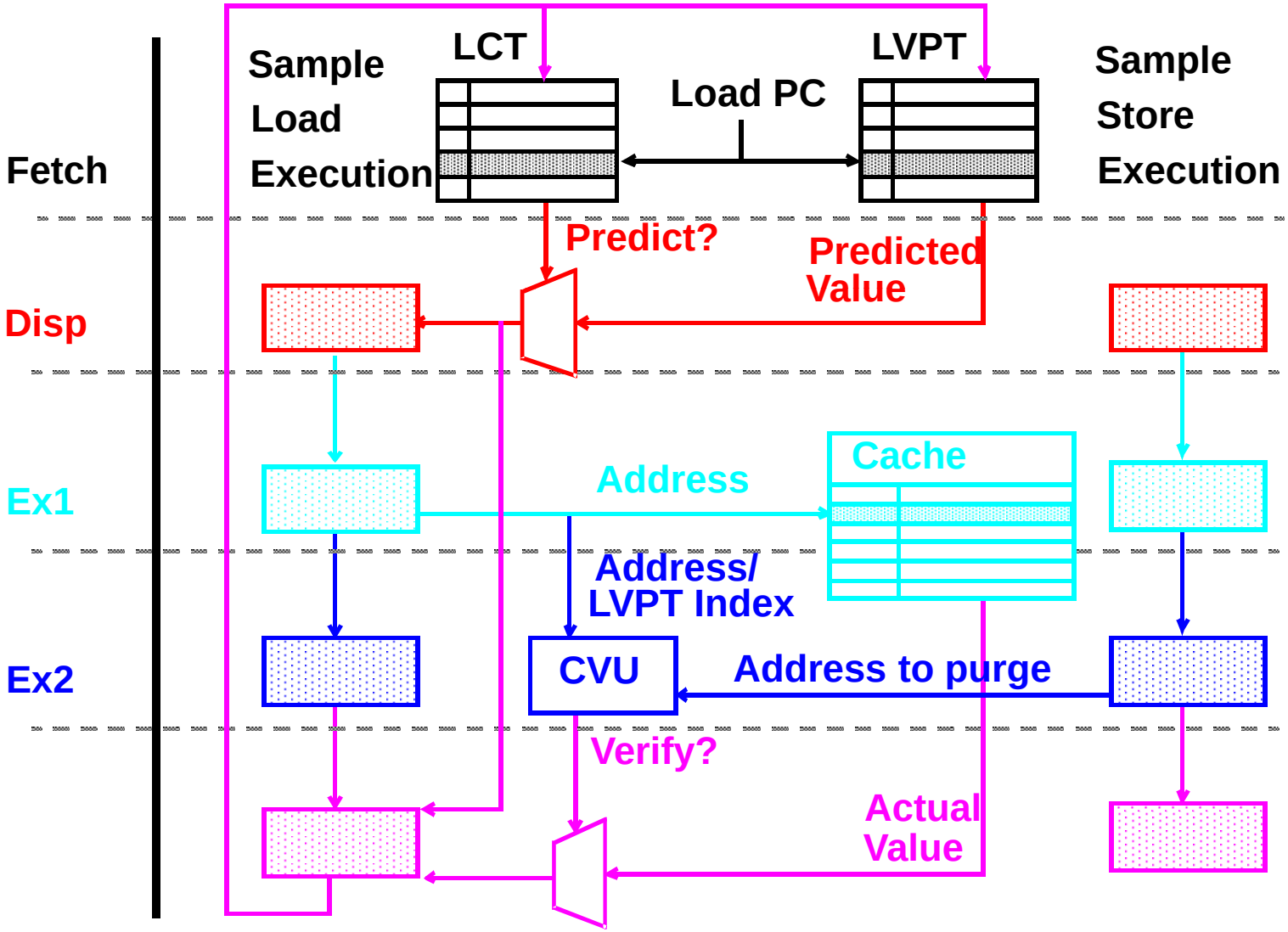
Dynamic Pipeline Contraction

Fold away pipeline stages via speculation:

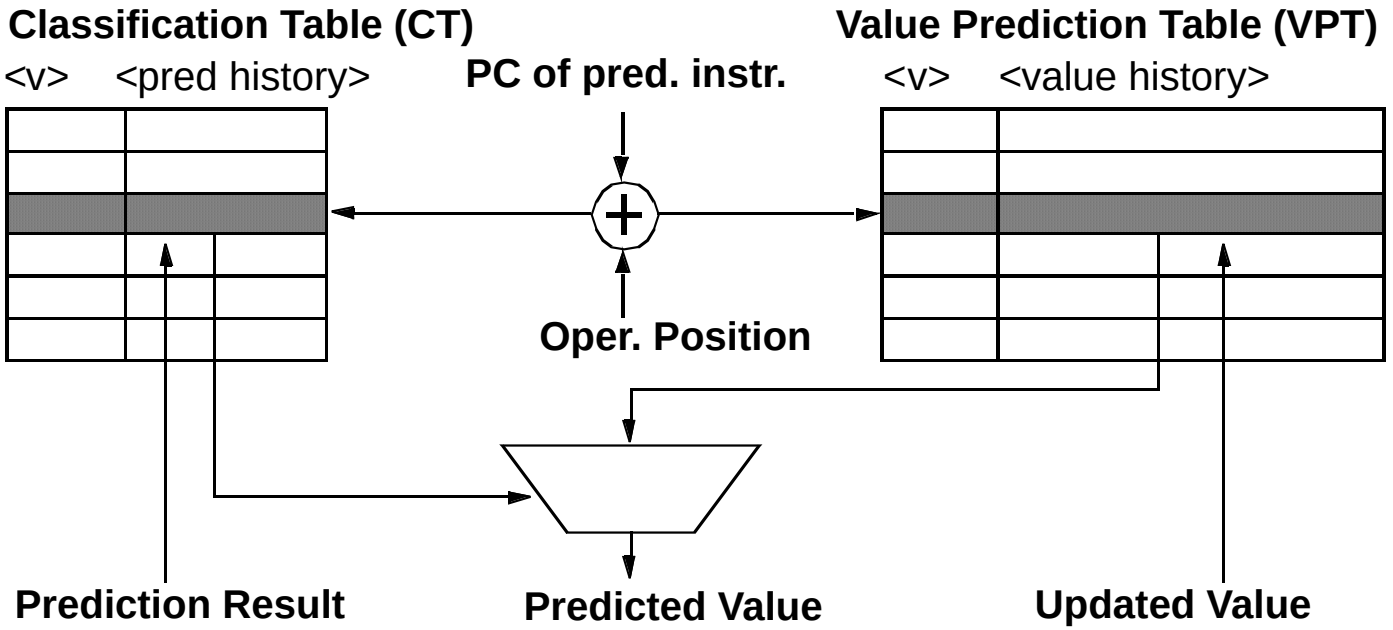
- Predict: obtain semantic outcome of instruction early
- Speculate: allow dependent instr. to execute in parallel
- Recover: Perform fix-up when mis-speculation occurs



Load Value Prediction and Classification



Source Operand Value Prediction



Similar to earlier work on value prediction, but predicts source operands:

- Decouples execution from dependence checking
- Don't care where value is coming from until validation

Confidence mechanism (CT) filters out wrong predictions