Advanced Memory Operations

Daniel Sanchez
Computer Science and Artificial Intelligence Laboratory
M.I.T.
Reminder: Direct-Mapped Cache

Tag | Index | Block Offset
---|---|---

\[ t \]
\[ k \]
\[ b \]

\[ V \]| Tag
---|---

\[ 2^k \] lines

HIT

Data Block

Data Word or Byte
Write Performance

Tag | Index | Block Offset

V Tag

Data

Data Word or Byte

2^k lines

HIT

WE
Write Performance

How does write timing compare to read timing?
Write Performance

How does write timing compare to read timing?

Completely serial!

October 11, 2023

MIT 6.5900 Fall 2023
Reducing Write Hit Time

Problem: Writes take two cycles in memory stage, one cycle for tag check plus one cycle for data write if hit
Reducing Write Hit Time

Problem: Writes take two cycles in memory stage, one cycle for tag check plus one cycle for data write if hit

View: Treat as data dependence on micro-architectural value ‘hit/miss’
Reducing Write Hit Time

Problem: Writes take two cycles in memory stage, one cycle for tag check plus one cycle for data write if hit

View: Treat as data dependence on micro-architectural value ‘hit/miss’

Solutions:
Reducing Write Hit Time

Problem: Writes take two cycles in memory stage, one cycle for tag check plus one cycle for data write if hit

View: Treat as data dependence on micro-architectural value ‘hit/miss’

Solutions:
• Stall – delivering data as fast as possible:
  – Circuit-level techniques (CAM tags) may allow one-cycle writes
Reducing Write Hit Time

Problem: Writes take two cycles in memory stage, one cycle for tag check plus one cycle for data write if hit

View: Treat as data dependence on micro-architectural value ‘hit/miss’

Solutions:

• Stall – delivering data as fast as possible:
  – Circuit-level techniques (CAM tags) may allow one-cycle writes

• Speculate predicting hit with greedy data update:
  – Design data RAM that can perform read and write in one cycle
  – Restore old value after tag miss (abort)
Reducing Write Hit Time

Problem: Writes take two cycles in memory stage, one cycle for tag check plus one cycle for data write if hit.

View: Treat as data dependence on micro-architectural value ‘hit/miss’

Solutions:
- Stall – delivering data as fast as possible:
  - Circuit-level techniques (CAM tags) may allow one-cycle writes.
- Speculate predicting hit with greedy data update:
  - Design data RAM that can perform read and write in one cycle.
  - Restore old value after tag miss (abort).
- Speculate predicting miss with lazy data update:
  - Hold write data for store in single buffer ahead of cache.
  - Write cache data during next idle data access cycle (commit).
**Pipelined/Delayed Write Timing**

Problem: Need to commit lazily saved write data

<table>
<thead>
<tr>
<th></th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD₀</td>
<td></td>
</tr>
<tr>
<td>ST₁</td>
<td></td>
</tr>
<tr>
<td>ST₂</td>
<td>Tag</td>
</tr>
<tr>
<td>LD₃</td>
<td>Data</td>
</tr>
<tr>
<td>ST₄</td>
<td></td>
</tr>
<tr>
<td>LD₅</td>
<td></td>
</tr>
<tr>
<td>ST₆ miss</td>
<td>Buffer</td>
</tr>
</tbody>
</table>

- LD: Load
- ST: Store
- Tag: Tagged column
- Data: Data column
- Buffer: Buffer column
Pipelined/Delayed Write Timing

Problem: Need to commit lazily saved write data

Solution: Write data during idle data cycle of next store’s tag check

<table>
<thead>
<tr>
<th></th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD₀</td>
<td></td>
</tr>
<tr>
<td>ST₁</td>
<td>Tag</td>
</tr>
<tr>
<td>ST₂</td>
<td></td>
</tr>
<tr>
<td>LD₃</td>
<td>Data</td>
</tr>
<tr>
<td>ST₄</td>
<td></td>
</tr>
<tr>
<td>LD₅</td>
<td>Buffer</td>
</tr>
<tr>
<td>ST₆ miss</td>
<td></td>
</tr>
</tbody>
</table>
Pipelined/Delayed Write Timing

Problem: Need to commit lazily saved write data

Solution: Write data during idle data cycle of next store’s tag check
### Pipelined/Delayed Write Timing

**Problem:** Need to commit lazily saved write data

**Solution:** Write data during idle data cycle of next store’s tag check

<table>
<thead>
<tr>
<th>Time</th>
<th>Tag</th>
<th>Data</th>
<th>Buffer</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **LD** 0
- **ST** 1
- **ST** 2
- **LD** 3
- **ST** 4
- **LD** 5
- **ST** 6 miss

In the diagram, `LD_0` is highlighted to indicate the timing issue during which the lazily saved write data needs to be committed.
Pipelined/Delayed Write Timing

Problem: Need to commit lazily saved write data

Solution: Write data during idle data cycle of next store’s tag check
Pipelined/Delayed Write Timing

Problem: Need to commit lazily saved write data

Solution: Write data during idle data cycle of next store’s tag check

![Diagram showing the timing of load and store operations.]

- LD₀
- ST₁
- ST₂
- LD₃
- ST₄
- LD₅
- ST₆ miss

The diagram illustrates the timing of load (LD) and store (ST) operations. LD₀ and LD₃ are shown in the data column, indicating that write data can be committed lazily during the idle data cycle of the next store's tag check.
Pipelined/Delayed Write Timing

Problem: Need to commit lazily saved write data

Solution: Write data during idle data cycle of next store’s tag check
Pipelined/Delayed Write Timing

Problem: Need to commit lazily saved write data

Solution: Write data during idle data cycle of next store’s tag check
Pipelined/Delayed Write Timing

Problem: Need to commit lazily saved write data

Solution: Write data during idle data cycle of next store’s tag check

<table>
<thead>
<tr>
<th>Tag</th>
<th>Data</th>
<th>Buffer</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD₀</td>
<td>LD₀</td>
<td>ST₁</td>
</tr>
<tr>
<td>ST₁</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ST₂</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LD₃</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ST₄</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LD₅</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ST₆ miss</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Pipelined/Delayed Write Timing

Problem: Need to commit lazily saved write data

Solution: Write data during idle data cycle of next store’s tag check
Pipelined/Delayed Write Timing

Problem: Need to commit lazily saved write data

Solution: Write data during idle data cycle of next store’s tag check

<table>
<thead>
<tr>
<th>Time</th>
<th>Tag</th>
<th>Data</th>
<th>Buffer</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LD₀</td>
<td>LD₀</td>
<td>ST₁</td>
</tr>
<tr>
<td></td>
<td>ST₁</td>
<td>ST₁</td>
<td>ST₂</td>
</tr>
<tr>
<td></td>
<td>LD₃</td>
<td>LD₃</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ST₄</td>
<td>ST₄</td>
<td></td>
</tr>
<tr>
<td></td>
<td>LD₅</td>
<td>LD₅</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ST₆</td>
<td>ST₆ miss</td>
<td></td>
</tr>
</tbody>
</table>
Pipelined/Delayed Write Timing

Problem: Need to commit lazily saved write data

Solution: Write data during idle data cycle of next store’s tag check
Pipelined/Delayed Write Timing

Problem: Need to commit lazily saved write data

Solution: Write data during idle data cycle of next store’s tag check

<table>
<thead>
<tr>
<th>Time</th>
<th>Tag</th>
<th>Data</th>
<th>Buffer</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LD₀</td>
<td>ST₁</td>
<td>ST₂</td>
</tr>
<tr>
<td></td>
<td>LD₀</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ST₁</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>LD₃</td>
<td>ST₁</td>
<td>ST₂</td>
</tr>
<tr>
<td></td>
<td>LD₅</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

LD₀ → ST₁ → ST₂

ld₀

Solution:

Write data during idle data cycle of next store’s tag check.
Problem: Need to commit lazily saved write data

Solution: Write data during idle data cycle of next store’s tag check
Pipelined/Delayed Write Timing

Problem: Need to commit lazily saved write data

Solution: Write data during idle data cycle of next store’s tag check

<table>
<thead>
<tr>
<th>Time</th>
<th>Tag</th>
<th>Data</th>
<th>Buffer</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LD₀</td>
<td>ST₁</td>
<td>ST₂</td>
</tr>
<tr>
<td></td>
<td>ST₁</td>
<td>LD₁</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ST₂</td>
<td>LD₂</td>
<td></td>
</tr>
<tr>
<td></td>
<td>LD₃</td>
<td>ST₁</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ST₄</td>
<td>ST₁</td>
<td></td>
</tr>
<tr>
<td></td>
<td>LD₅</td>
<td>ST₁</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ST₆ miss</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Problem: Need to commit lazily saved write data

Solution: Write data during idle data cycle of next store’s tag check

Pipelined/Delayed Write Timing

<table>
<thead>
<tr>
<th>Time</th>
<th>Tag</th>
<th>Data</th>
<th>Buffer</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LD₀</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ST₁</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ST₂</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>LD₃</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>LD₀</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ST₁</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>LD₃</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ST₁</td>
<td>ST₂</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>ST₂</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>LD₅</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ST₆</td>
<td>miss</td>
<td></td>
</tr>
</tbody>
</table>

LD₀, ST₁, ST₂, LD₃

Problem:

Solution:
Pipelined/Delayed Write Timing

Problem: Need to commit lazily saved write data

Solution: Write data during idle data cycle of next store’s tag check

<table>
<thead>
<tr>
<th>Time</th>
<th>Tag</th>
<th>Data</th>
<th>Buffer</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LD₀</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ST₁</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ST₂</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>LD₃</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>LD₀</td>
<td></td>
<td>ST₁</td>
</tr>
<tr>
<td></td>
<td>ST₁</td>
<td></td>
<td>ST₂</td>
</tr>
<tr>
<td></td>
<td>LD₃</td>
<td></td>
<td>ST₂</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ST₁</td>
<td>ST₂</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ST₂</td>
<td></td>
</tr>
</tbody>
</table>

LD₀: Load 0, ST₁: Store 1, LD₃: Load 3, ST₄: Store 4, ST₆: Store 6 miss
Pipelined/Delayed Write Timing

Problem: Need to commit lazily saved write data

Solution: Write data during idle data cycle of next store’s tag check

<table>
<thead>
<tr>
<th>Time</th>
<th>Tag</th>
<th>Data</th>
<th>Buffer</th>
<th>Buffer</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LD₀</td>
<td>ST₁</td>
<td>ST₂</td>
<td>ST₄</td>
</tr>
<tr>
<td></td>
<td>LD₀</td>
<td>ST₁</td>
<td>LD₃</td>
<td>ST₂</td>
</tr>
<tr>
<td></td>
<td>LD₅</td>
<td>ST₁</td>
<td>ST₂</td>
<td>ST₂</td>
</tr>
<tr>
<td></td>
<td>ST₆ miss</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **LD₀**: Load 0
- **ST₁**: Store 1
- **ST₂**: Store 2
- **LD₃**: Load 3
- **ST₄**: Store 4
- **ST₆**: Store 6

Problem: Need to commit lazily saved write data

Solution: Write data during idle data cycle of next store’s tag check
### Pipelined/Delayed Write Timing

**Problem:** Need to commit lazily saved write data

**Solution:** Write data during idle data cycle of next store’s tag check

<table>
<thead>
<tr>
<th>Time</th>
<th>Tag</th>
<th>Data</th>
<th>Buffer</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LD₀</td>
<td>ST₁</td>
<td>ST₁</td>
</tr>
<tr>
<td></td>
<td>ST₁</td>
<td>ST₂</td>
<td>ST₂</td>
</tr>
<tr>
<td></td>
<td>LD₃</td>
<td></td>
<td>ST₂</td>
</tr>
<tr>
<td></td>
<td>ST₄</td>
<td></td>
<td>ST₄</td>
</tr>
<tr>
<td></td>
<td>LD₅</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ST₆ miss</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Pipelined/Delayed Write Timing

Problem: Need to commit lazily saved write data

Solution: Write data during idle data cycle of next store’s tag check
Pipelined/Delayed Write Timing

Problem: Need to commit lazily saved write data

Solution: Write data during idle data cycle of next store’s tag check

<table>
<thead>
<tr>
<th>Time</th>
<th>Tag</th>
<th>Data</th>
<th>Buffer</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LD₀</td>
<td>ST₁</td>
<td>ST₁</td>
</tr>
<tr>
<td></td>
<td>ST₁</td>
<td>LD₃</td>
<td>ST₁</td>
</tr>
<tr>
<td></td>
<td>ST₂</td>
<td>ST₂</td>
<td>ST₂</td>
</tr>
<tr>
<td></td>
<td>LD₃</td>
<td>ST₂</td>
<td>ST₂</td>
</tr>
<tr>
<td></td>
<td>ST₄</td>
<td>ST₄</td>
<td>ST₄</td>
</tr>
<tr>
<td></td>
<td>LD₅</td>
<td>ST₄</td>
<td>ST₄</td>
</tr>
<tr>
<td></td>
<td>LD₅</td>
<td>ST₄</td>
<td>ST₄</td>
</tr>
</tbody>
</table>

- LD₀: Load 0
- ST₁: Store 1
- ST₂: Store 2
- LD₃: Load 3
- ST₄: Store 4
- LD₅: Load 5
- ST₆: Store 6 (miss)
## Pipelined/Delayed Write Timing

### Problem
Need to commit lazily saved write data

### Solution
Write data during idle data cycle of next store’s tag check

<table>
<thead>
<tr>
<th>Tag</th>
<th>LD₀</th>
<th>ST₁</th>
<th>ST₂</th>
<th>LD₃</th>
<th>ST₄</th>
<th>LD₅</th>
<th>ST₆</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data</td>
<td>LD₀</td>
<td>ST₁</td>
<td>LD₃</td>
<td>ST₂</td>
<td>LD₅</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Buffer</td>
<td></td>
<td>ST₁</td>
<td>ST₂</td>
<td>ST₂</td>
<td>ST₄</td>
<td>ST₄</td>
<td></td>
</tr>
</tbody>
</table>

- LD₀, ST₁, ST₂, LD₃, ST₄, LD₅: Operations
- ST₆ miss: Store tag check miss

**Diagram:**
- Arrows indicate the flow of data operations.
- Colored boxes represent the timing of operations over time.

**Remark:**
- Write data during idle data cycle of next store’s tag check.
Problem: Need to commit lazily saved write data

Solution: Write data during idle data cycle of next store’s tag check

```
<table>
<thead>
<tr>
<th>Time</th>
<th>Tag</th>
<th>Data</th>
<th>Buffer</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LD₀</td>
<td>ST₁</td>
<td>ST₁</td>
</tr>
<tr>
<td></td>
<td>ST₁</td>
<td>LD₃</td>
<td>ST₂</td>
</tr>
<tr>
<td></td>
<td>ST₂</td>
<td>LD₅</td>
<td>ST₄</td>
</tr>
<tr>
<td></td>
<td>LD₅</td>
<td>ST₆</td>
<td>ST₆</td>
</tr>
<tr>
<td></td>
<td>ST₆mü</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Problem: Need to commit lazily saved write data

Solution: Write data during idle data cycle of next store’s tag check

```
<table>
<thead>
<tr>
<th>Time</th>
<th>Tag</th>
<th>Data</th>
<th>Buffer</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LD₀</td>
<td>ST₁</td>
<td>ST₁</td>
</tr>
<tr>
<td></td>
<td>ST₁</td>
<td>LD₃</td>
<td>ST₂</td>
</tr>
<tr>
<td></td>
<td>ST₂</td>
<td>LD₅</td>
<td>ST₄</td>
</tr>
<tr>
<td></td>
<td>LD₅</td>
<td>ST₆</td>
<td>ST₆</td>
</tr>
<tr>
<td></td>
<td>ST₆mü</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
### Pipelined/Delayed Write Timing

**Problem:** Need to commit lazily saved write data

**Solution:** Write data during idle data cycle of next store’s tag check

<table>
<thead>
<tr>
<th>Time</th>
<th>Buffer</th>
<th>Data</th>
<th>Tag</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ST₁</td>
<td>ST₁</td>
<td>LD₀</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>ST₂</td>
</tr>
<tr>
<td></td>
<td>ST₂</td>
<td>ST₂</td>
<td>LD₃</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>ST₂</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>LD₅</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>ST₆</td>
</tr>
<tr>
<td></td>
<td>ST₆ miss</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

LD₀ points to the first load operation. ST₁, ST₂, ST₄, and ST₆ are store operations, with ST₆ miss indicating a cache miss.

LD₃ points to the second load operation, and so on.

The timing diagram shows the sequence of operations with arrows indicating the flow of data and tag checks through the pipeline stages.
Pipelining Cache Writes

What if instruction needs data in delayed write buffer?
Pipelining Cache Writes

*What if instruction needs data in delayed write buffer?*  Bypass
Pipelining Cache Writes

What if instruction needs data in delayed write buffer? Bypass

What is condition for bypass?
Pipelining Cache Writes

What if instruction needs data in delayed write buffer? Bypass

What is condition for bypass? Tag and index match “delayed write addr”
Pipelining Cache Writes

What if instruction needs data in delayed write buffer?  Bypass

What is condition for bypass?  Tag and index match “delayed write addr”

Address and Store Data From CPU

![Diagram of cache and store operations]

- Tags
- Index
- Delayed Write Addr.
- Delayed Write Data
- Data
- Hit?
- Load Data to CPU
- Load/Store
- S L

Tag and index match "delayed write addr"
Write Policy Choices

• Cache hit:
  – **Write-through**: write both cache & memory
    • generally higher traffic but simplifies multi-processor design
  – **Write-back**: write cache only
    (memory is written only when the entry is evicted)
    • a dirty bit per block can further reduce the traffic

• Cache miss:
  – **No-write-allocate**: only write to main memory
  – **Write-allocate** (aka fetch on write): fetch into cache

• Common combinations:
  – write-through and no-write-allocate
  – write-back with write-allocate
Reducing Read Miss Penalty

**Problem:** Write buffer may hold updated value of location needed by a read miss – RAW data hazard

Evicted dirty lines for writeback cache
OR
All writes in writethrough cache
Reducing Read Miss Penalty

**Problem:** Write buffer may hold updated value of location needed by a read miss – RAW data hazard

**Stall:** On a read miss, wait for the write buffer to go empty
Reducing Read Miss Penalty

Problem: Write buffer may hold updated value of location needed by a read miss – RAW data hazard

Stall: On a read miss, wait for the write buffer to go empty

Bypass: Check write buffer addresses against read miss addresses, if no match, allow read miss to go ahead of writes, else, return value in write buffer
We’ve handled the register dependencies, but what about memory operations?
Speculative Loads / Stores

- Problem: Just like register updates, stores should not permanently change the architectural memory state until after the instruction is committed.

- Choice: Data update policy: greedy or lazy?
Speculative Loads / Stores

• Problem: Just like register updates, stores should not permanently change the architectural memory state until after the instruction is committed

• Choice: Data update policy: greedy or lazy?
  Lazy: Add a speculative store buffer, a structure to lazily hold speculative store data.
Speculative Loads / Stores

- Problem: Just like register updates, stores should not permanently change the architectural memory state until after the instruction is committed.

- Choice: Data update policy: greedy or lazy?
  Lazy: Add a speculative store buffer, a structure to lazily hold speculative store data.

- Choice: Handling of store-to-load data hazards: stall, bypass, speculate...?
Speculative Loads / Stores

- Problem: Just like register updates, stores should not permanently change the architectural memory state until after the instruction is committed.

- Choice: Data update policy: greedy or lazy?
  Lazy: Add a speculative store buffer, a structure to lazily hold speculative store data.

- Choice: Handling of store-to-load data hazards: stall, bypass, speculate...?
  Bypass: ...
Store Buffer Responsibilities
Store Buffer Responsibilities

- **Lazy store of data:** Buffer new data values for stores
Store Buffer Responsibilities

- **Lazy store of data**: Buffer new data values for stores

- **Commit/abort**: The data from the oldest instructions must either be committed to memory or forgotten
Store Buffer Responsibilities

• **Lazy store of data:** Buffer new data values for stores

• **Commit/abort:** The data from the oldest instructions must either be committed to memory or forgotten

*Commits are generally done in order – why?*
Store Buffer Responsibilities

- **Lazy store of data:** Buffer new data values for stores

- **Commit/abort:** The data from the oldest instructions must either be committed to memory or forgotten

Commits are generally done in order – why?

WAW Hazards
Store Buffer Responsibilities

- **Lazy store of data:** Buffer new data values for stores

- **Commit/abort:** The data from the oldest instructions must either be committed to memory or forgotten

- **Bypass:** Data from older instructions must be provided (or forwarded) to younger instructions before the older instruction is committed

"Commits are generally done in order – why?"

WAW Hazards
Store Buffer – Lazy data management

Speculative Store Buffer

<table>
<thead>
<tr>
<th>Inum</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>VS</td>
<td>VT</td>
<td>D1</td>
</tr>
<tr>
<td>VS</td>
<td>VT</td>
<td>D2</td>
</tr>
<tr>
<td>VS</td>
<td>VT</td>
<td>D3</td>
</tr>
<tr>
<td>VS</td>
<td>VT</td>
<td>D4</td>
</tr>
<tr>
<td>VS</td>
<td>VT</td>
<td>D5</td>
</tr>
</tbody>
</table>

Store Address

L1 Data Cache

Tags

Data

Store Commit Path

Load Data

- On store execute:
Store Buffer – Lazy data management

• On store execute:
  - mark valid and speculative; save tag, data, and instruction number
Store Buffer – Lazy data management

- On store execute:
  - mark valid and speculative; save tag, data, and instruction number
- On store commit:

```
<table>
<thead>
<tr>
<th>Inum</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
L1 Data Cache
```

```
Tags
```

```
Data
```

```
Store Address
```

```
Speculative Store Buffer
```

```
Load Data
```

```
Store Commit Path
```

October 11, 2023
Store Buffer – Lazy data management

- On store execute:
  - mark valid and speculative; save tag, data, and instruction number
- On store commit:
  - clear speculative bit and eventually move data to cache
Store Buffer – Lazy data management

- On store execute:
  - mark valid and speculative; save tag, data, and instruction number
- On store commit:
  - clear speculative bit and eventually move data to cache
- On store abort:
Store Buffer – Lazy data management

- On store execute:
  - mark valid and speculative; save tag, data, and instruction number
- On store commit:
  - clear speculative bit and eventually move data to cache
- On store abort:
  - clear valid bit
### Store Buffer - Bypassing

#### Load Address

<table>
<thead>
<tr>
<th>Inum</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>VS</td>
<td>Inum</td>
<td>Tag</td>
</tr>
<tr>
<td>VS</td>
<td>Inum</td>
<td>Tag</td>
</tr>
<tr>
<td>VS</td>
<td>Inum</td>
<td>Tag</td>
</tr>
<tr>
<td>VS</td>
<td>Inum</td>
<td>Tag</td>
</tr>
<tr>
<td>VS</td>
<td>Inum</td>
<td>Tag</td>
</tr>
<tr>
<td>VS</td>
<td>Inum</td>
<td>Tag</td>
</tr>
<tr>
<td>VS</td>
<td>Inum</td>
<td>Tag</td>
</tr>
</tbody>
</table>
Store Buffer - Bypassing

<table>
<thead>
<tr>
<th>VS</th>
<th>Inum</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>VS</td>
<td>Inum</td>
<td>Tag</td>
<td>Data</td>
</tr>
<tr>
<td>VS</td>
<td>Inum</td>
<td>Tag</td>
<td>Data</td>
</tr>
<tr>
<td>VS</td>
<td>Inum</td>
<td>Tag</td>
<td>Data</td>
</tr>
<tr>
<td>VS</td>
<td>Inum</td>
<td>Tag</td>
<td>Data</td>
</tr>
<tr>
<td>VS</td>
<td>Inum</td>
<td>Tag</td>
<td>Data</td>
</tr>
<tr>
<td>VS</td>
<td>Inum</td>
<td>Tag</td>
<td>Data</td>
</tr>
</tbody>
</table>
Store Buffer - Bypassing

What fields must be examined for bypassing?

<table>
<thead>
<tr>
<th>V</th>
<th>S</th>
<th>Inum</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>V</td>
<td>S</td>
<td>Inum</td>
<td>Tag</td>
<td>Data</td>
</tr>
<tr>
<td>V</td>
<td>S</td>
<td>Inum</td>
<td>Tag</td>
<td>Data</td>
</tr>
<tr>
<td>V</td>
<td>S</td>
<td>Inum</td>
<td>Tag</td>
<td>Data</td>
</tr>
<tr>
<td>V</td>
<td>S</td>
<td>Inum</td>
<td>Tag</td>
<td>Data</td>
</tr>
<tr>
<td>V</td>
<td>S</td>
<td>Inum</td>
<td>Tag</td>
<td>Data</td>
</tr>
</tbody>
</table>
Store Buffer - Bypassing

What fields must be examined for bypassing?

Valid, Inum, and tag
Store Buffer - Bypassing

What fields must be examined for bypassing?

- Valid, Inum, and tag

- If data in both store buffer and cache, which should we use?
Store Buffer - Bypassing

What fields must be examined for bypassing?

Valid, Inum, and tag

- If data in both store buffer and cache, which should we use?
  Speculative store buffer - if store older than load
Store Buffer - Bypassing

What fields must be examined for bypassing?

Valid, Inum, and tag

- If data in both store buffer and cache, which should we use?
  Speculative store buffer - if store older than load

- If same address in store buffer twice, which should we use?
Store Buffer - Bypassing

What fields must be examined for bypassing?
Valid, Inum, and tag

- If data in both store buffer and cache, which should we use?
  Speculative store buffer - if store older than load

- If same address in store buffer twice, which should we use?
  Youngest store older than load

<table>
<thead>
<tr>
<th>Load Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>VS</td>
</tr>
<tr>
<td>VS</td>
</tr>
<tr>
<td>VS</td>
</tr>
<tr>
<td>VS</td>
</tr>
<tr>
<td>VS</td>
</tr>
<tr>
<td>VS</td>
</tr>
<tr>
<td>VS</td>
</tr>
</tbody>
</table>
Store Buffer - Bypassing

What fields must be examined for bypassing?

Valid, Inum, and tag

• If data in both store buffer and cache, which should we use?
  Speculative store buffer - if store older than load

• If same address in store buffer twice, which should we use?
  Youngest store older than load

• Calculating entry needed in the store buffer can be considered a dependence on the index needed to access the store buffer. So store buffer bypassing can be managed speculatively by building a simple predictor that guesses that the specific entry in the store buffer the load needs. So what happens if we guessed the wrong entry?
Store Buffer - Bypassing

What fields must be examined for bypassing?

Valid, Inum, and tag

- If data in both store buffer and cache, which should we use?
  Speculative store buffer - if store older than load

- If same address in store buffer twice, which should we use?
  Youngest store older than load

- Calculating entry needed in the store buffer can be considered a
dependence on the index needed to access the store buffer. So store
buffer bypassing can be managed speculatively by building a simple
predictor that guesses that the specific entry in the store buffer the
load needs. So what happens if we guessed the wrong entry?
  Declare a mis-speculation and abort.
Memory Dependencies

For registers, we used tags or physical register numbers to determine dependencies. What about memory operations?

\[
\text{sw } x1, (x2) \\
\text{lw } x3, (x4)
\]

*When is the load dependent on the store?*
Memory Dependencies

For registers, we used tags or physical register numbers to determine dependencies. What about memory operations?

\[
\text{sw } x1, (x2) \\
\text{lw } x3, (x4)
\]

*When is the load dependent on the store?*

*When* \( x2 == x4 \)*
Memory Dependencies

For registers, we used tags or physical register numbers to determine dependencies. What about memory operations?

\[
\text{sw x1, (x2)} \\
\text{lw x3, (x4)}
\]

When is the load dependent on the store?

When x2 == x4

Does our ROB know this at issue time?
Memory Dependencies

For registers, we used tags or physical register numbers to determine dependencies. What about memory operations?

\[
\text{sw } x1, (x2) \\
\text{lw } x3, (x4)
\]

*When is the load dependent on the store?*

When \( x2 == x4 \)

*Does our ROB know this at issue time?* No
In-Order Memory Queue

sw x1, (x2)
lw x3, (x4)

Stall naively:

• Execute all loads and stores in program order

=> Load and store cannot start execution until all previous loads and stores have completed execution

• Can still execute loads and stores speculatively, and out-of-order with respect to other instructions
Conservative O-o-O Load Execution

\[
\begin{align*}
\text{sw } & \ x1, \ (x2) \\
\text{lw } & \ x3, \ (x4)
\end{align*}
\]

Stall intelligently:

- Split execution of store instruction into two phases: address calculation and data write
- Can execute load before store, if addresses known and x4 \(\neq\) x2
- Each load address compared with addresses of all previous uncommitted stores (\textit{can use partial conservative check, e.g., bottom 12 bits of address})
- Don’t execute load if any previous store address not known

\textit{(MIPS R10K, 16 entry address queue)}
Address Speculation

\[
\begin{align*}
&\text{sw } x1, (x2) \\
&\text{lw } x3, (x4)
\end{align*}
\]
Address Speculation

```
sw x1, (x2)
lw x3, (x4)
```

1. Guess that x4 != x2, and execute load before store address known
Address Speculation

sw x1, (x2)
lw x3, (x4)

1. Guess that x4 != x2, and execute load before store address known
2. If x4 != x2 commit...
Address Speculation

\[
\text{sw x1, (x2)} \\
\text{lw x3, (x4)}
\]

1. Guess that \( x_4 \neq x_2 \), and execute load before store address known

2. If \( x_4 \neq x_2 \) commit...

3. But if \( x_4 = x_2 \), squash load and \textit{all} following instructions

   - To support squash we need to hold all completed but uncommitted load/store addresses/data in program order
Address Speculation

\[
\text{sw } x1, \ (x2) \\
\text{lw } x3, \ (x4)
\]

1. Guess that \( x4 \neq x2 \), and execute load before store address known

2. If \( x4 \neq x2 \) commit…

3. But if \( x4=x2 \), squash load and all following instructions

   - To support squash we need to hold all completed but uncommitted load/store addresses/data in program order

   How do we resolve the speculation, i.e., detect when we need to squash?
Address Speculation

\[
\begin{align*}
\text{sw } x1, (x2) \\
\text{lw } x3, (x4)
\end{align*}
\]

1. Guess that \( x4 \neq x2 \), and execute load before store address known

2. If \( x4 \neq x2 \) commit...

3. But if \( x4 = x2 \), squash load and \textit{all} following instructions
   - To support squash we need to hold all completed but uncommitted load/store addresses/data in program order

\textit{How do we resolve the speculation, i.e., detect when we need to squash?}

Watch for stores that arrive after load that needed its data
Speculative Load Buffer

**Speculation check:** Detect if a load has executed before an earlier store to the same address – missed RAW hazard

- On load execute:
Speculative Load Buffer

**Speculation check:** Detect if a load has executed before an earlier store to the same address – missed RAW hazard

- **On load execute:**
  - mark entry valid, and set instruction number and address tag.

---

Load Address

<table>
<thead>
<tr>
<th>V</th>
<th>Inum</th>
<th>Tag</th>
</tr>
</thead>
<tbody>
<tr>
<td>V</td>
<td>Inum</td>
<td>Tag</td>
</tr>
<tr>
<td>V</td>
<td>Inum</td>
<td>Tag</td>
</tr>
<tr>
<td>V</td>
<td>Inum</td>
<td>Tag</td>
</tr>
<tr>
<td>V</td>
<td>Inum</td>
<td>Tag</td>
</tr>
<tr>
<td>V</td>
<td>Inum</td>
<td>Tag</td>
</tr>
</tbody>
</table>
Speculative Load Buffer

Speculation check:
Detect if a load has executed before an earlier store to the same address – missed RAW hazard

- On load execute:
  - mark entry valid, and set instruction number and address tag.
- On load commit:

![Speculative Load Buffer Diagram]

<table>
<thead>
<tr>
<th>Load Address</th>
<th>V</th>
<th>Inum</th>
<th>Tag</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>V</td>
<td>Inum</td>
<td>Tag</td>
</tr>
<tr>
<td></td>
<td>V</td>
<td>Inum</td>
<td>Tag</td>
</tr>
<tr>
<td></td>
<td>V</td>
<td>Inum</td>
<td>Tag</td>
</tr>
<tr>
<td></td>
<td>V</td>
<td>Inum</td>
<td>Tag</td>
</tr>
<tr>
<td></td>
<td>V</td>
<td>Inum</td>
<td>Tag</td>
</tr>
</tbody>
</table>
Speculative Load Buffer

**Speculation check:**
Detect if a load has executed before an earlier store to the same address – missed RAW hazard

- On load execute:
  - mark entry valid, and set instruction number and address tag.
- On load commit:
  - clear valid bit
Speculative Load Buffer

**Speculation check:**
Detect if a load has executed before an earlier store to the same address – missed RAW hazard

- On load execute:
  - mark entry valid, and set instruction number and address tag.
- On load commit:
  - clear valid bit
- On load abort:
Speculative Load Buffer

**Speculation check:** Detect if a load has executed before an earlier store to the same address – missed RAW hazard

- **On load execute:**
  - mark entry valid, and set instruction number and address tag.
- **On load commit:**
  - clear valid bit
- **On load abort:**
  - clear valid bit
Speculative Load Buffer

- If data in load buffer with instruction younger than store:
Speculative Load Buffer

- If data in load buffer with instruction younger than store:
  - Speculative violation – abort!
Speculative Load Buffer

- If data in load buffer with instruction younger than store:
  - Speculative violation – abort!

=> Large penalty for inaccurate address speculation
Speculative Load Buffer

- If data in load buffer with instruction younger than store:
  - Speculative violation – abort!

=> Large penalty for inaccurate address speculation

Does tag match have to be perfect?
Speculative Load Buffer

- If data in load buffer with instruction younger than store:
  - Speculative violation – abort!

=> Large penalty for inaccurate address speculation

*Does tag match have to be perfect?* No!
Speculative Load Buffer

- If data in load buffer with instruction younger than store:
  - Speculative violation – abort!

=> Large penalty for inaccurate address speculation

*Does tag match have to be perfect?* No!

*How do we reduce the chance of mis-speculation?*
Speculative Load Buffer

- If data in load buffer with instruction younger than store:
  - Speculative violation – abort!

=> Large penalty for inaccurate address speculation

Does tag match have to be perfect? No!

How do we reduce the chance of mis-speculation? Predict
Memory Dependence Prediction
(Alpha 21264)

sw x1, (x2)
lw x3, (x4)

1. Guess that x4 != x2 and execute load before store

2. If later find x4==x2, squash load and all following instructions, but mark load instruction as *store-wait*

   • Subsequent executions of the same load instruction will wait for all previous stores to complete

   • Periodically clear *store-wait* bits
Memory Dependence Prediction
(Alpha 21264)

sw x1, (x2)
lw x3, (x4)

1. Guess that x4 != x2 and execute load before store

2. If later find x4==x2, squash load and all following instructions, but mark load instruction as *store-wait*

  • Subsequent executions of the same load instruction will wait for all previous stores to complete

  • Periodically clear *store-wait* bits

Notice the general problem of predictors that learn something but can’t unlearn it
Store Sets
(Alpha 21464)

- Multiple Readers
- Multiple Writers
  - multiple code paths
  - multiple components of a single location

Program Order

<table>
<thead>
<tr>
<th>PC</th>
<th>0</th>
<th>Store</th>
<th>4</th>
<th>Store</th>
<th>8</th>
<th>Store</th>
<th>12</th>
<th>Store</th>
<th>28</th>
<th>Load</th>
<th>32</th>
<th>Load</th>
<th>36</th>
<th>Load</th>
<th>40</th>
<th>Load</th>
</tr>
</thead>
</table>

Multiple Readers

Multiple Writers

PC 0
PC 12
PC 8
Memory Dependence Prediction using Store Sets

- A load must wait for any stores in its store set that have not yet executed

- The processor approximates each load’s store set by initially allowing naïve speculation and recording memory-order violations
# The Store Set Map Table

- Store/Load Pair causing Memory Order Violation

<table>
<thead>
<tr>
<th>Program Order</th>
<th>Store Set Map Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>Store</td>
<td>Index</td>
</tr>
<tr>
<td>.</td>
<td></td>
</tr>
<tr>
<td>.</td>
<td></td>
</tr>
<tr>
<td>.</td>
<td></td>
</tr>
<tr>
<td>.</td>
<td></td>
</tr>
<tr>
<td>.</td>
<td></td>
</tr>
<tr>
<td>Store</td>
<td>Index</td>
</tr>
<tr>
<td>Load</td>
<td>Index</td>
</tr>
<tr>
<td>.</td>
<td></td>
</tr>
<tr>
<td>.</td>
<td></td>
</tr>
<tr>
<td>.</td>
<td></td>
</tr>
<tr>
<td>.</td>
<td></td>
</tr>
<tr>
<td>.</td>
<td></td>
</tr>
<tr>
<td>Load</td>
<td>Index</td>
</tr>
<tr>
<td>.</td>
<td></td>
</tr>
<tr>
<td>.</td>
<td></td>
</tr>
<tr>
<td>.</td>
<td></td>
</tr>
<tr>
<td>.</td>
<td></td>
</tr>
<tr>
<td>.</td>
<td></td>
</tr>
<tr>
<td>Load</td>
<td>Index</td>
</tr>
<tr>
<td>.</td>
<td></td>
</tr>
<tr>
<td>.</td>
<td></td>
</tr>
<tr>
<td>.</td>
<td></td>
</tr>
<tr>
<td>.</td>
<td></td>
</tr>
<tr>
<td>.</td>
<td></td>
</tr>
</tbody>
</table>

- Store Set A

Writer

Reader
Store Set Sharing for Multiple Readers

Program Order

Store Set Map Table

- Store/Load Pair causing Memory Order Violation
- Store/Load Pair causing Memory Order Violation
Prefetching

- Execution of a load ‘depends’ on the data it needs being in the cache...
Prefetching

• Execution of a load ‘depends’ on the data it needs being in the cache...

• Speculate on future instruction and data accesses and fetch them into cache(s)
  – Instruction accesses easier to predict than data accesses
Prefetching

- Execution of a load ‘depends’ on the data it needs being in the cache...

- Speculate on future instruction and data accesses and fetch them into cache(s)
  - Instruction accesses easier to predict than data accesses

- Varieties of prefetching
  - Hardware prefetching
  - Software prefetching
  - Mixed schemes
Prefetching

• Execution of a load ‘depends’ on the data it needs being in the cache...

• Speculate on future instruction and data accesses and fetch them into cache(s)
  – Instruction accesses easier to predict than data accesses

• Varieties of prefetching
  – Hardware prefetching
  – Software prefetching
  – Mixed schemes

• How does prefetching affect cache misses?
Prefetching

• Execution of a load ‘depends’ on the data it needs being in the cache...

• Speculate on future instruction and data accesses and fetch them into cache(s)
  – Instruction accesses easier to predict than data accesses

• Varieties of prefetching
  – Hardware prefetching
  – Software prefetching
  – Mixed schemes

• How does prefetching affect cache misses?
  Compulsory
Prefetching

- Execution of a load ‘depends’ on the data it needs being in the cache...

- Speculate on future instruction and data accesses and fetch them into cache(s)
  - Instruction accesses easier to predict than data accesses

- Varieties of prefetching
  - Hardware prefetching
  - Software prefetching
  - Mixed schemes

- How does prefetching affect cache misses?

  Compulsory
  Reduce
Prefetching

- Execution of a load ‘depends’ on the data it needs being in the cache...

- Speculate on future instruction and data accesses and fetch them into cache(s)
  - Instruction accesses easier to predict than data accesses

- Varieties of prefetching
  - Hardware prefetching
  - Software prefetching
  - Mixed schemes

- How does prefetching affect cache misses?

  Compulsory Conflict
  Reduce
Prefetching

• Execution of a load ‘depends’ on the data it needs being in the cache...

• Speculate on future instruction and data accesses and fetch them into cache(s)
  – Instruction accesses easier to predict than data accesses

• Varieties of prefetching
  – Hardware prefetching
  – Software prefetching
  – Mixed schemes

• *How does prefetching affect cache misses?*

  Compulsory: Reduce
  Conflict: ~Increase
Prefetching

- Execution of a load ‘depends’ on the data it needs being in the cache...

- Speculate on future instruction and data accesses and fetch them into cache(s)
  - Instruction accesses easier to predict than data accesses

- Varieties of prefetching
  - Hardware prefetching
  - Software prefetching
  - Mixed schemes

- How does prefetching affect cache misses?

<table>
<thead>
<tr>
<th></th>
<th>Compulsory</th>
<th>Conflict</th>
<th>Capacity</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Reduce</td>
<td>~Increase</td>
<td></td>
</tr>
</tbody>
</table>
Prefetching

- Execution of a load ‘depends’ on the data it needs being in the cache...

- Speculate on future instruction and data accesses and fetch them into cache(s)
  - Instruction accesses easier to predict than data accesses

- Varieties of prefetching
  - Hardware prefetching
  - Software prefetching
  - Mixed schemes

- How does prefetching affect cache misses?

<table>
<thead>
<tr>
<th>Compulsory</th>
<th>Conflict</th>
<th>Capacity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reduce</td>
<td>~Increase</td>
<td>~Increase</td>
</tr>
</tbody>
</table>
Issues in Prefetching

- Usefulness – should produce hits
- Timeliness – not late and not too early
- Cache and bandwidth pollution

![Diagram of CPU, RF, L1 Instruction, L1 Data, Unified L2 Cache, Prefetched data]
Hardware Instruction Prefetching

Instruction prefetch in Alpha AXP 21064

- Fetch two blocks on a miss; the requested block (i) and the next consecutive block (i+1)
- Requested block placed in cache, and next block in instruction stream buffer
- If miss in cache but hit in stream buffer, move stream buffer block into cache and prefetch next block (i+2)
Hardware Data Prefetching

• Prefetch-on-miss:
  – Prefetch $b + 1$ upon miss on $b$

• One Block Lookahead (OBL) scheme
  – Initiate prefetch for block $b + 1$ when block $b$ is accessed
  – Why is this different from doubling block size?
  – Can extend to N-block lookahead (called stream prefetching)

• Strided prefetch
  – If observe sequence of accesses to block $b$, $b+N$, $b+2N$, then prefetch $b+3N$ etc.

Example: IBM Power 5 [2003] supports eight independent streams of strided prefetch per processor, prefetching 12 lines ahead of current access
Good Luck on Quiz 1!

Next lecture: Multithreading