CIS 429/529
Memory Management

Prof. Michel A. Kinsky
Improving Cache Performance

• Average memory access time = 
  \[ \text{Hit time} + \text{Miss rate} \times \text{Miss penalty} \]

• To improve performance:
  ‣ reduce the hit time
  ‣ reduce the miss rate (e.g., larger cache)
  ‣ reduce the miss penalty (e.g., L2 cache)

• What is the simplest design strategy?
  ‣ Biggest cache that doesn’t increase hit time past 1-2 cycles (approx 8-32KB in modern technology)
Causes for Cache Misses

- **Compulsory:**
  - first-reference to a block a.k.a. cold start misses
  - misses that would occur even with infinite cache

- **Conflict:**
  - misses from collisions due to block-placement strategy
  - misses due to the cache being too small to hold all data the program needs
Effect of Cache on Performance

• Larger cache size
  ‣ reduces conflict misses
  ‣ hit time will increase

• Higher associativity
  ‣ reduces conflict misses
  ‣ may increase hit time

• Larger block size
  ‣ reduces compulsory misses
  ‣ exploit burst transfers in memory and on buses
  ‣ increases miss penalty and conflict misses
Replacement Policy

• Which block from a set should be evicted?
  ▶ Random
  ▶ Least Recently Used (LRU)
    • LRU cache state must be updated on every access
    • true implementation only feasible for small sets (2-way)
    • pseudo-LRU binary tree often used for 4-8 way
  ▶ First In, First Out (FIFO) a.k.a. Round-Robin
    • used in highly associative caches
  ▶ Not Least Recently Used (NLRU)
    • FIFO with exception for most recently used block or blocks
Multilevel Caches

- A memory cannot be large and fast
- Increasing sizes of cache at each level

Local miss rate = misses in cache / accesses to cache
Global miss rate = misses in cache / CPU memory accesses
Misses per instruction = misses in cache / number of instructions
A Typical Memory Hierarchy

- **L1 Instruction Cache**
- **L1 Data Cache**
- **Unified L2 Cache**
- **Multi-ported register file (part of CPU)**
- **Large unified secondary cache (on-chip SRAM)**
- **Multiple interleaved memory banks (DRAM)**

**Explanations**:
- **Split instruction & data primary caches (on-chip SRAM)**
- **Multi-ported register file (part of CPU)**
- **Large unified secondary cache (on-chip SRAM)**
- **Multiple interleaved memory banks (DRAM)**
Victim Caches

- Victim cache is a small associative back up cache, added to a direct L1 Data Cache.

![Diagram showing the relationship between CPU, RF, L1 Data Cache, Unified L2 Cache, Victim Cache, and flow of data.]

- Evicted data from L1
- Hit data (miss in L1)
- Evicted data from VC

where?
Victim Caches

- Victim cache is a small associative back up cache, added to a direct mapped cache, which holds recently evicted lines.
  1. First look up in direct mapped cache
  2. If miss, look in victim cache
  3. If hit in victim cache, swap hit line with line now evicted from L1
  4. If miss in victim cache, L1 victim -> VC, VC victim-?>

- Fast hit time of direct mapped but with reduced conflict misses
Itanium-2 On-Chip Caches

- Level 1, 16KB, 4-way s.a., 64B line, quad-port (2 load +2 store), single cycle latency
- Level 2, 256KB, 4-way s.a, 128B line, quad-port (4 load or 4 store), five cycle latency
- Level 3, 3MB, 12-way s.a., 128B line, single 32B port, twelve cycle latency
Direct-Mapped Cache

- **Tag**
- **Index**
- **Block Offset**

- \( V \)
- \( t \)
- \( k \)
- \( b \)

- 2^k lines

**HIT**

**Data Word or Byte**
Write Performance

HIT

Data Word or Byte

\[ V = \text{Tag} \]

\[ t \]

\[ k \]

\[ b \]

\[ 2^k \text{ lines} \]
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

• Solutions:
  ‣ Design data RAM that can perform read and write in one cycle, restore old value after tag miss
  ‣ Fully-associative (CAM Tag) caches: Word line only enabled if hit
  ‣ Pipelined writes: Hold write data for store in single buffer ahead of cache, write cache data during next store’s tag check
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>ST₁</td>
<td>ST₂</td>
</tr>
<tr>
<td></td>
<td>ST₁</td>
<td>LD₃</td>
<td>ST₂</td>
</tr>
<tr>
<td></td>
<td>ST₂</td>
<td>ST₄</td>
<td>ST₄</td>
</tr>
<tr>
<td></td>
<td>ST₄</td>
<td>LD₅</td>
<td></td>
</tr>
</tbody>
</table>

LD₀: Load 0
ST₁: Store 1
ST₂: Store 2
LD₃: Load 3
ST₄: Store 4
LD₅: Load 5
Pipelining Cache Writes

Address and Store Data From CPU

Tags \[\Rightarrow\] Delayed Write Addr. \[\Rightarrow\] Delayed Write Data

Load/Store

Data

Hit?

Load Data to CPU

Data from a store hit written into data portion of cache during tag access of subsequent store
Write Policy Choices

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

• 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

Evicted dirty lines for writeback cache
OR
All writes in writethru 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

• 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
Prefetching

• 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

• What types of misses does prefetching affect?
Issues in Prefetching

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

- **Strided prefetch**
  - If observe sequence of accesses to block \( b, b+N, b+2N, \) then prefetch \( b+3N \) etc.