8
Data Management Policies

18-548/15-548 Memory System Architecture
Philip Koopman
September 21, 1998

Required Reading: Cragon 2.2.4-2.2.6, 3.5.2
Supplemental Reading: VanderWiel paper, July 1997 Computer, pp. 23-30
Przybylski paper, 1990 ISCA, pp. 160-169

Assignments

♦ By Wednesday September 30 read about memory operation & sizing:
  • Understanding SRAM operation (IBM App. Note)
  • What’s all this Flash stuff? (National Semiconductor)

♦ Homework 4 due September 23

♦ Lab 2 due September 25

♦ Test 1 on September 28
  • In-class review September 23 -- look at example tests before class
**Where Are We Now?**

- **Where we’ve been:**
  - Data organization
  - Associativity

- **Where we’re going today:**
  - Policies -- how to manage the data
  - Policies apply to all levels of memory hierarchy

- **Where we’re going next:**
  - Memory operation & cache chip area
  - Multi-level caches to improve performance

**Preview**

- **Data fetching policies**
  - When do you fetch and how much?
  - Blocking vs. non-blocking caches

- **Data replacement strategies**
  - How do you select a victim for replacement?
    - LRU
    - Random

- **Data storing policies**
  - When do you store, and how much?
    - Write Allocation
    - Write-through & Write-back
  - Write buffering
# Fetch Policies

- **Order of moving words from main memory to cache**
  - Which word gets fetched first?

- **When can CPU resume processing after cache miss?**
  - Non-blocking cache

- **Conditions that trigger a fetch from main memory to cache**
  - Fetch on miss (demand fetch for read; block fill for write)
  - Software prefetching (compiler/programmer give hints to HW)
  - Hardware prefetching (hardware speculatively fetches)
    - Special case is instruction prefetching: sequential, branch targets
Cache Fetching: Load Policies

- **Block load**: always start from beginning of block
- **Load Forward**: load only remainder of sector
  - Must load at least all data covered by a single Valid bit => 1 block or more
- **Fetch Bypass**: start from needed word, then fill rest of block
  - Called “critical word first” in H&P
  - Also known as “wrap around”

![Diagram showing cache fetch policies](image)

Cache Fetching: Resumption of Processing

- **Simple fetch**: wait until entire cache block is present
- **Forward (early restart)**: restart as soon as word is available
  - Needed to benefit from wrap-around load policy
  - Risk of cache miss to word already requested for load into cache (complicates control logic) a bit

![Diagram showing cache resumption of processing](image)
Non-Blocking Caches

- Non-Blocking means CPU doesn’t stall on cache miss or write completion
  - “Blocking” caches stall CPU until access is completed
- Non-Blocking speeds operation
  - Out-of-order execution unit can issue multiple reads
  - Once write is issued, can proceed without waiting for write to complete
- Adds complexity
  - Check if reads refer to data not yet written
  - Ensure proper ordering of reads that map into same cache block (might return out of order in some memory subsystems)
- Control ties in with data dependency control (e.g., “scoreboard”) -- beyond scope of this course

Non-Blocking Cache Benefits

(After Hennessy & Patterson Figure 5.22)
Hardware Prefetching

- **Instruction prefetching** [Smith 1982]; “useful for 32- to 64-byte blocks”
  - Always prefetch: prefetch word after current word
  - Tagged prefetch: prefetch next block if current block was a cache miss
  - Prefetch on misses: have blocks > 1 word; prefetch entire block

- **Instruction queues trigger prefetching**
  - Queue refill for in-line instructions
  - Branch target queue speculatively fetches branch targets

- **Data prefetching**
  - IBM S/360/91 speculatively fetches data before instruction released to execution unit
  - Vector address generators (discussed later)
  - Data prefetching is difficult
    - Need to know effective address, which may be computed
    - Need way to inhibit for memory-mapped I/O (e.g., C “volatile” keyword)

Software Prefetching

- **Software-initiated, non-blocking** load of cache block in anticipation of need
  - Doesn’t halt execution
  - BUT, does consume **bandwidth**
    - Might cause stall if another cache miss occurs when this load is being processed
    - Want to put in otherwise unused instruction issue slots
  - [Callahan 91]: ~33% of data prefetches turn out to have be useful

- **Example: Power PC 601**
  - Data Cache Block Touch -- loads block into cache

- **DEC Alpha:**
  - “use prefetching only when transport times ~ 100 clocks”
Patterned Prefetching

- **Obvious prefetching is to exploit sequentiality**
  - In-order prefetching and large block sizes “look” similar
  - Branch prediction prefetches are “logical” in-order instead of physical in-order
- **But, can also do patterned prefetches**
  - Fetch every \(i\)th element when accessing a matrix.
  - Use software hints to generate prefetch instructions via compiler
- **Alternate implementation: large register file**
  - For out-of-order execution, simply load value into a register well before it is needed
  - BUT, might generate page faults, whereas machine support can ignore prefetch if not readily accessible

REPLACEMENT POLICIES
## Replacement Policy

- **Replacement needed for capacity and conflict misses**
  - Read miss
  - Write miss with write-allocate
  - Goal: minimize number of conflict misses

- **Direct-mapped cache** -- only one possible block to replace

- **Set associative & associative caches** -- select a victim
  - Least Recently Used (LRU)
    - Typically best
  - Random (typically pseudorandom)
    - Easier to build, ~12% performance penalty compared to LRU
  - First In First Out (FIFO)
    - Probably no better than random

### LRU Replacement

- **Least Recently Used**
  - Requires status bits to track LRU element per set
  - 2-way set associative: keep flag with most recently accessed sector; replace the other one
  - *m*-way set associative:
    - *m* counters of size $\log_2 m$
      - Can infer state of one counter from all other counter values; might not be worth trouble
    - Initialization:
      - Initialize all counters to different values and mark contents “invalid” on system reset
    - Allocate new sector:
      - allocate sector with counter value of 0
      - proceed to access sector below
    - Access any sector:
      - decrement all counters with values higher than accessed sector
      - set accessed sector counter to all 1
**LRU Example...**

<table>
<thead>
<tr>
<th>LRU VALUE</th>
<th>DATA</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>LRU VALUE</th>
<th>DATA</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>LRU VALUE</th>
<th>DATA</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- 4-way set associative; but only a single 3 bit LRU value
  - True LRU would require 3 @ 2-bit counters and more complicated logic
  - B0, B1, B2 track LRU within set partitions

**Intel Pentium Pseudo-LRU**

- All four lines in the set valid?
  - Yes
    - B0 = 0?
      - Yes: I0 or I1 least recently used
      - No: I2 or I3 least recently used
  - No
    - Replace non-valid line

  - B1 = 0?
    - Yes: Replace I0
    - No: Replace I1

  - B2 = 0?
    - Yes: Replace I2
    - No: Replace I3

(Intel Pentium manual)
Random Replacement

- Simulations indicate almost as good as LRU
  - Less hardware to implement
  - i860 used random replacement

- Obvious way to implement is with Linear Feedback Shift Register (LFSR)

LRU Can Be Brittle...

- LRU is usually the best with “normal” data
  - Works well when temporal locality is smaller than cache size

- BUT, LRU is brittle in degenerate cases
  - Example case: array size A with cache size C, iteratively read array
    - For cache size C \( \geq A \), LRU results in 0% conflict misses, 0% capacity misses
  - Fully associative is brittle
    - For \( A = C+1 \), gets 100% miss rate (each element removed just as it is about to be needed)
  - Set associative is not quite as bad
    - For \( A = C+k \) the first k sets of cache get 100% miss rate
    - Degrades to 100% overall miss rate with \( k = C \) / #sets
  - Direct mapped is best
    - Degrades smoothly to 100% overall miss rate with \( A = 2 \times C \)
... While Random Can Be Robust

- Sometimes brittleness is bad
  - Especially when customers get “unreasonable” surprises
  - For example, increasing the data set of a program to need just one more TLB entry when the TLB is fully associative with LRU replacement...

- Random, on the other hand, gives smooth behavior in more cases
  - More than 0% miss rate even in best case (“false” conflict misses)
  - But, not 100% miss rate even when data set larger than cache size

- Random is also suboptimal in the “everyday” case by a few percent

Random Replacement Smooths Response

EFFECTS OF LRU or RANDOM REPLACEMENT ON ARRAY-SCANNING CODE

- If cache weren’t empty when program started, could have greater than 0% miss rate even for cache size > array size
Degenerate Case For LRU Replacement

- Happens when touching a number of elements > cache size before returning to first element
- Three regions of behavior (example -- program that iterates scanning an array):
  - 100% cache misses
    - cache size < array size - (array size/associativity)
  - Linearly decreasing cache misses
    - cache size within (array size/associativity) of array size
  - 100% cache hits
    - cache size > array size

Concept in Real Life:

- Name a real-life situation where LRU replacement of an item is preferred

- Name a real-life situation where random replacement is practiced because of the overhead cost of tracking LRU information isn’t worth the effort
WRITE POLICIES

Write Policies

- **Write data destination:** is value written to memory or just cache?
  - **Write through** -- always written
  - **Write back** *(a.k.a. copy-back)* -- written only when cache block evicted
  - **Write once**
    - First write as write through; subsequent as write back
    - Good hack for multiprocessors

- **Write miss:** allocate block if it’s a miss?
  - **Write-allocate** -- pick a victim and evict it on write miss
  - **No-write-allocate** -- don’t disturb cache on write miss
### Write Through

- **When writing, send value to next level down in memory hierarchy**
  - Typically a write buffer is used as a staging area

- **Advantages**
  - Simpler to implement, especially on multi-processor
  - Makes sense if data is seldom re-written

- **Disadvantages**
  - Potentially increased memory traffic (if words are rewritten multiple times)
  - Potential coherence problem if write buffer is used (must check write buffer as well as caches)

### Write Back

- **When writing to cache, don’t write to memory**
  - Set Dirty bit indicating modified value present
  - Only the last write to a memory location is recorded, when data is “evicted” from cache

- **Advantages**
  - Reduces bus traffic for high-touch variables

- **Disadvantages**
  - Requires space for dirty bits in cache
  - Must be careful to track coherency of evicted value until it reaches memory
  - Increases latency for evicting dirty blocks (may be a net loss if data is seldom rewritten)
    - Cache miss must include time to remove block before writing new data
    - Read miss latency may not be increased -- overlap eviction with fetching new data
### Write-Allocate

- **Treats write miss similarly to read miss** -- **allocates cache sector containing written value**
  - Can be used with either write through or write back; usually used with write back

**Advantages**
- Works well for programs that do a lot of write/read (as opposed to read/write)
  - Stacks/activation records
  - Garbage-collected heaps
- When used with write back can attenuate multiprocessor bus traffic

**Disadvantages**
- Must fetch non-written data to complete block (thus, works best if there is one word per block)
- If large blocks are used, can increase bus traffic to fill unwritten block fragments
- Can pollute cache with “dead” values that won’t be re-read before eviction

### No-Write-Allocate

- **On write miss, value is not cached**
  - Typically used with write-through policy
  - Non allocation implies that all write misses use write-through

**Advantages**
- **Simpler** design
  - In programs with long latency between write and subsequent read, doesn’t pollute cache with long-term-storage items

**Disadvantages**
- Can really hurt performance if write/read behavior is occurring
  - (Software hack: dummy read before writing to simulate write allocation)
WRITE BUFFERING
Write Buffer

- With write-through cache: reduces stalls on consecutive writes
  - Smooths bursts of bus accesses for back-to-back writes
    - Useful for saving multiple registers for procedure call, interrupt, etc.
    - Non-blocking implementation must check contents against data dependencies
  - 80486 has 4-level write buffer; [CRAW90] shows average occupancy of 3

- With write back cache: holds block during multi-cycle write to memory
  - Allows cache to be used while waiting for write when block size > transfer size to next level of memory hierarchy
  - IBM RS/6000 writes 128-byte block in 8 clock cycles

- “Main Memory” could instead be L2 cache

![Diagram of Write Buffer](image)

Write Assembly Cache

- Expansion of write buffer idea (write buffer with extra circuitry)
  - Holds writes to a physical memory word, waiting for another write to that same word
  - Captures spatial locality of writes
    - Stores to structs
    - Stores to arrays
    - Register pushes for subroutine calls
  - Captures temporal locality of writes (e.g., statically allocated scratch variable)
  - Primarily effective when write is uncached
    - Write no allocate
    - Write through

- Examples
  - VAX 8800 -- single-line WAC
  - NCR -- multi-line WAC’s in workstations
Write Assembly Cache Effectiveness

- WAC block size of 4/8 bytes; Transfer size of 4/8 bytes; 4-way set associativity
  (Flynn Figure 5.40)

Write Priority to Reduce Miss Penalty

- Simple approach is to stall if write buffer non-empty on miss
  - Guarantees read miss will access correct data
  - Increases miss penalty (1.5x for 4-word buffer on MIPS M/1000)

- Better approach is give reads priority over writes
  - On write through, write buffer waits until free bus cycles are available, giving reads priority
  - On write back, reading to fill cache block takes priority over eviction
  - Requires control logic to ensure any read miss correctly reflects contents of write buffer
# Customized Policies

- **Customizable operating mode depending on expected workload**
  - Can be general mode bit
  - Can be specific for a particular instruction
- **MC68040 example**
  - Noncacheable mode: forces data out of cache
    - Shared variables in absence of multiprocessor coherency
  - Cacheable, write-through/write-no-allocate
    - If compiler “knows” variable won’t be accessed for a long time
    - Especially useful for scientific code where arrays > cache size
  - Cacheable, write back/allocate
    - If compiler “knows” variable will be accessed again soon
  - Special access -- freezes cache
    - Read/write misses do not allocate
    - Useful for deterministic execution times
Fetch Policies

- **Prefetch interacts with block & sector size**
  - Can use prefetch to fill entire sector instead of just one block
    - Reduces memory traffic on writes -- only a block is written, not entire sector
  - On unsectored caches prefetch can give sector-prefetch effects
    - But, still pay area penalty for one tag per block

Replacement Policy

- **LRU replacement can become brittle with highly associative caches**
  - Saw this in homework #3 with TLB sizing on some machines
  - Can be an issue with any computer that manipulates large data arrays -- may want to use random replacement instead

- **LRU replacement uses chip area and time**
  - Intel uses pseudo-LRU to save space & speed up operation
**Write Policies**

- **Write through may be effective for large block sizes**
  - Avoids having to write back large block if only one word has changed

- **Write-no-allocate may be effective for large block sizes**
  - Avoids having to read in other words to fill block

- **BUT, can avoid both these problems with sectored cache**
  - Write back conserves bandwidth, especially important on multi-processors
  - Write allocate conserves bandwidth for areas having write/read behavior, generally improves effectiveness of write back cache

- **Write assembly buffer can help if write-through policy is used**
  - Simulates a single-set write back cache
  - Want WAB size to have a “block size” appropriate for spatial write locality in workload.

---

**REVIEW**
Review

- **Fetch policies determine how and when to fetch data**
  - Prefetching to improve hit rate; but at cost of bandwidth
  - Non-blocking caches help decouple memory and processing strategies
    - Required for effective out-of-order execution of memory accesses
- **Replacement policies select which block to allocate/evict**
  - LRU -- complicated but (usually) best
  - Random -- easier, less brittle in degenerate cases
- **Write policies determine when data is written to memory**
  - Write through is simpler, but often higher bandwidth than write back
  - Write-allocate helps with write-before-read locations
  - Write buffering can decouple CPU from memory access

Key Concepts

- **Latency & Concurrency**
  - Prefetch can reduce latency with speculative operations
  - Non-blocking caches reduces latency for concurrent memory accesses
- **Bandwidth**
  - Write through vs. write back is a bandwidth tradeoff that depends on program characteristics
- **Replication**
  - Multiple blocks per sector can decouple desire for prefetch from cost of tags and cost of writing unmodified data
- **Balance**
  - Miss rate vs. traffic ratio is a classic balance issue
    - Write through vs. write back
    - Block size, sector size, and prefetch strategy