Locality and Caches

- Data Locality
  - Temporal: if data item needed now, it is likely to be needed again in near future
  - Spatial: if data item needed now, nearby data likely to be needed in near future

- Exploiting Locality: Caches
  - Keep recently used data in fast memory close to the processor
  - Also bring nearby data there

Storage Hierarchy and Locality
Memory Latency is Long

- 60-100ns not uncommon
- Quick back-of-the-envelope calculation:
  - 2GHz CPU
  - \( \rightarrow 0.5\text{ns/cycle} \)
  - 100ns memory \( \rightarrow 200\text{ cycle memory latency!} \)
- Solution: Caches

Cache

- Placement: where in the cache can a block go?

Cache Basics

- Fast (but small) memory close to processor
- When data referenced
  - If in cache, use cache instead of memory
  - If not in cache, bring into cache (actually, bring entire block of data, toO)
- Important decisions
  - Placement: where in the cache can a block go?
  - Identification: how do we find a block in cache?
  - Replacement: what to kick out to make room in cache?
  - Write policy: What do we do about stores?
  - Solution: Caches
  - Identification: how do we find a block in cache?
  - If in cache, use cache instead of memory
  - Maybe have to kick something else out to do it!
**Cache Basics**

- Cache consists of block-sized lines
  - Line size typically power of two
  - Typically 16 to 128 bytes in size
- Example
  - Suppose block size is 128 bytes
    - Lowest seven bits determine offset within block
  - Read data at address \( A = 0x7fffa3f4 \)
  - Address begins to block with base address \( 0x7fffa380 \)

**Cache Placement**

- Placement
  - Which memory blocks are allowed into which cache lines
- Placement Policies
  - Direct mapped (block can go to only one line)
  - Fully Associative (block can go to any line)
  - Set-associative (block can go to one of \( N \) lines)
    - E.g., if \( N=4 \), the cache is 4-way set associative
    - Other two policies are extremes of this
      (E.g., if \( N=1 \) we get a direct-mapped cache)

**Cache Identification**

- When address referenced, need to
  - Find whether its data is in the cache
  - If it is, find where in the cache
  - This is called a cache lookup
- Each cache line must have
  - A valid bit (1 if line has data, 0 if line empty)
    - We also say the cache line is valid or invalid
  - A tag to identify which block is in the line
    (if line is valid)
**Cache Replacement**

- Need a free line to insert new block
  - Which block should we kick out?
- Several strategies
  - Random (randomly selected line)
  - FIFO (line that has been in cache the longest)
  - LRU (least recently used line)
  - LRU Approximations (Pseudo LRU)

**Implementing LRU**

- Have LRU counter for each line in a set
- When line accessed
  - Get old value X of its counter
  - Set its counter to max value
  - For every other line in the set
    - If counter larger than X, decrement it
- When replacement needed
  - Select line whose counter is 0
**Write Policy**

- Do we allocate cache lines on a write?
  - Write-allocate
    - A write miss brings block into cache
  - No-write-allocate
    - A write miss leaves cache as it was
- Do we update memory on writes?
  - Write-through
    - Memory immediately updated on each write
  - Write-back
    - Memory updated when line replaced

**Write Through/Write Back**

- Write-through
- Write-back

**Write-Back Caches**

- Need a *Dirty* bit for each line (stored in the Tag!
  - A dirty line has more recent data than memory
- Line starts as *clean* (not dirty)
- Line becomes dirty on first write to it
  - Memory not updated yet, cache has the only up-to-date copy of data for a dirty line
- Replacing a dirty line
  - Must write data back to memory (write-back)
Tag Storage

- Any information related to cache other than data is stored in the tag storage.
- Not only tag bits, information for replacement, dirty bits (if we need), valid bit (in the future, cache coherence state information)

Review questions

- Memory addresses A, A+1, A+2, A+3, A+4
  - Spatial locality or temporal locality?
    - Spatial locality
- Memory addresses A, B, C, A, B, C, A, B, C
  - Spatial locality or temporal locality?
    - Temporal locality

Review questions-II

- Here is a series of address references given as word address: 1, 4, 8, 5, 20, 17, 19, 56, 9, 11, 4, 43, 5, 6, 9, 17. Assuming a direct-mapped cache with 16 one-word blocks that is initially empty, label each reference in the list as a hit or miss and show the final contents of the cache.
Review questions-III

- A computer has an 8KB write-through cache. Each cache block is 64 bits, the cache is 4-way set associative and uses the true LRU replacement policy. Assume a 24-bit address space and byte-addressable memory. How big (in bits) is the tag store?

Interleaving

- Multiple Concurrent

![Diagram of multiple concurrent banks]

Word Address

Bank 0
Bank 1
Bank 2
Bank 3

Chip Enable

Overlapped

Bank 0

Bank 1

Bank 2

Bank 3

Works as like multiple ports

Cache Performance

- Miss rate
  - Fraction of memory accesses that miss in cache
  - Hit rate = 1 – miss rate
- Average memory access time
  AMAT = hit time + miss rate * miss penalty
- Memory stall cycles

\[
\text{CPU time} = \text{Cycle time} \times (\text{Cycles}_{\text{exec}} + \text{Cycles}_{\text{stall}})
\]

\[
\text{Cycle time} = \text{Cache misses} \times (\text{Miss latency}_{\text{hit}} + \text{Miss latency}_{\text{stall}})
\]
Improving Cache Performance

• AMAT = hit time + miss rate * miss penalty
  – Reduce miss penalty
  – Reduce miss rate
  – Reduce hit time

• Cycles_{MemoryStall} = CacheMisses x (MissLatency_{Total} − MissLatency_{Overlapped})
  – Increase overlapped miss latency
  – Increase memory level parallelism

Kinds of Cache Misses

• The “3 Cs”
  – Compulsory: have to have these
    • Miss the first time each block is accessed
  – Capacity: due to limited cache capacity
    • Would not have them if cache size was infinite
  – Conflict: due to limited associativity
    • Would not have them if cache was fully associative

CPU-DRAM
**SRAM vs. DRAM**

- DRAM = Dynamic RAM

- SRAM: 6T per bit
  - built with normal high-speed CMOS technology

- DRAM: 1T per bit
  - built with special DRAM process optimized for density

**Hardware Structures**

- DRAM = Dynamic RAM
  - built with normal high-speed CMOS technology

**DRAM Chip Organization**

- Memory Cell Array
- Data Bus
- Column Decoder
- Row Decoder
• Differences with SRAM
  • reads are destructive: contents are erased after reading
    – row buffer
  • read lots of bits all at once, and then parcel them out based on different column addresses
    – similar to reading a full cache line, but only accessing one word at a time
  • “Fast-Page Mode” FPM DRAM organizes the DRAM row to contain bits for a complete page
    – row address held constant, and then fast read from different locations from the same page

• DRAM Read Operation

• Destructive Read

- After read of 0 or 1, cell contains something close to 1/2
Prefetcher

1st level cache 1-3 cycles  
Software prefetcher

2nd level cache 10 – 20 cycles  
Hardware prefetcher

memory 200 – 600 cycles

Prefetching

• Predict future misses and get data into cache  
  – If access does happen, we have a hit now  
    (or a partial miss, if data is on the way)  
  – If access does not happen, cache pollution  
    (replaced other data with junk we don’t need)

• To avoid pollution, prefetch buffers  
  – Pollution a big problem for small caches  
  – Have a small separate buffer for prefetches  
    – How big?

• Use 2nd level cache as a prefetch buffer.

Software Prefetching

• Two flavors: register prefetch and cache prefetch

• Each flavor can be faulting or non-faulting  
  – If address bad, does it create exceptions?

• Faulting register prefetch is binding  
  – It is a normal load, address must be OK, uses register

• Not faulting cache prefetch is non-binding  
  – If address bad, becomes a NOP  
    – Does not affect register state  
    – Has more overhead (load still there),  
      ISA change (prefetch instruction),  
      complicates cache (prefetches and loads different)
**Hardware Prefetcher**

- Stream
- Stride
- Markov
- Content based prefetcher

---

**HW Stream Prefetcher**

- Observer cache miss stream address
- Detect stream or stride behavior
  - L2 cache miss creates stream
  - L1 or L2 miss trains stream

```
L2 cache miss (create stream)
L1 or L2 miss (Train stream)
```