CS 61C: Great Ideas in Computer Architecture (Machine Structures)
Caches Part 1

Instructors:
John Wawrzynek & Vladimir Stojanovic
http://inst.eecs.berkeley.edu/~cs61c/
New-School Machine Structures (It’s a bit more complicated!)

- **Parallel Requests**
  Assigned to computer
  e.g., Search “Katz”

- **Parallel Threads**
  Assigned to core
  e.g., Lookup, Ads

- **Parallel Instructions**
  >1 instruction @ one time
  e.g., 5 pipelined instructions

- **Parallel Data**
  >1 data item @ one time
  e.g., Add of 4 pairs of words

- **Hardware descriptions**
  All gates @ one time

- **Programming Languages**

---

**Software**

**Harness Parallelism & Achieve High Performance**

**Hardware**

Warehouse Scale Computer

How do we know?

Computer

Core

Memory

Input/Output

Instruction Unit(s)

Functional Unit(s)

Logic Gates

A_0+B_0 A_1+B_1 A_2+B_2 A_3+B_3

Cache Memory

Smart Phone

2
Components of a Computer

- Processor
  - Control
  - Datapath
    - Registers
    - Arithmetic & Logic Unit (ALU)
- Memory
  - Program
  - Bytes
  - Data
- Processor-Memory Interface
- I/O-Memory Interfaces

Input

Output
Problem: Large memories slow?

Library Analogy

• Finding a book in a large library takes time
  – Takes time to search a large card catalog – (mapping title/author to index number)
  – Round-trip time to walk to the stacks and retrieve the desired book.
• Larger libraries makes both delays worse
• Electronic memories have the same issue, plus the technologies that we use to store an individual bit get slower as we increase density (SRAM versus DRAM versus Magnetic Disk)

However what we want is a large yet fast memory!
1980 microprocessor executes ~one instruction in same time as DRAM access
2015 microprocessor executes ~1000 instructions in same time as DRAM access

Slow DRAM access could have disastrous impact on CPU performance!
What to do: Library Analogy

• Want to write a report using library books
  – E.g., works of J.D. Salinger
• Go to Doe library, look up relevant books, fetch from stacks, and place on desk in library
• If need more, check them out and keep on desk
  – But don’t return earlier books since might need them
• You hope this collection of ~10 books on desk enough to write report, despite 10 being only 0.00001% of books in UC Berkeley libraries
Big Idea: Memory Hierarchy

Processor

Size of memory at each level

As we move to outer levels the latency goes up and price per bit goes down. Why?

Increasing distance from processor, decreasing speed

Inner
Levels in memory hierarchy

Outer
Real Memory Reference Patterns

Big Idea: Locality

- **Temporal Locality** (locality in time)
  - Go back to same book on desktop multiple times
  - If a memory location is referenced, then it will tend to be referenced again soon

- **Spatial Locality** (locality in space)
  - When go to book shelf, pick up multiple books on J.D. Salinger since library stores related books together
  - If a memory location is referenced, the locations with nearby addresses will tend to be referenced soon
### Memory Reference Patterns

#### Temporal Locality

#### Spatial Locality

---

Principle of Locality

- *Principle of Locality*: Programs access small portion of address space at any instant of time (spatial locality) and repeatedly access that portion (temporal locality)
- What program structures lead to *temporal* and *spatial locality* in instruction accesses?
- In data accesses?
Memory Reference Patterns

- Instruction fetches
- Stack accesses
- Data accesses

Address

Time

n loop iterations

subroutine call

argument access

vector access

scalar accesses

subroutine return
Cache Philosophy

• Programmer-invisible hardware mechanism to give illusion of speed of fastest memory with size of largest memory
  – Works fine even if programmer has no idea what a cache is
  – However, performance-oriented programmers today sometimes “reverse engineer” cache design to design data structures to match cache
  – We’ll do that in Project 3
Memory Access without Cache

• Load word instruction: \texttt{lw $t0,0($t1)}
• \( $t1 \) contains 1022_{ten}, Memory[1022] = 99

1. Processor issues address 1022_{ten} to Memory
2. Memory reads word at address 1022_{ten} (99)
3. Memory sends 99 to Processor
4. Processor loads 99 into register $t0
Adding Cache to Computer

Processor - Memory Interface

I/O - Memory Interfaces
Memory Access with Cache

• Load word instruction: \texttt{lw $t0, 0($t1)}
• $t1$ contains $1022_{\text{ten}}$, Memory[1022] = 99
• With cache: Processor issues address $1022_{\text{ten}}$ to Cache
  
  1. Cache checks to see if has copy of data at address $1022_{\text{ten}}$
    2a. If finds a match (Hit): cache reads 99, sends to processor
    2b. No match (Miss): cache sends address 1022 to Memory
      I. Memory reads 99 at address $1022_{\text{ten}}$
      II. Memory sends 99 to Cache
      III. Cache replaces word with new 99
      IV. Cache sends 99 to processor
  2. Processor loads 99 into register $\texttt{t0}$
Administrivia

• HW2 due 10/16 @ 23:59:59
• Project 3-1 due date now 10/21 – already released, *start early*
• Project 3-2 due date now 10/28 (release 10/18)

• Midterm 1:
  – grades posted today
  – Stats, next slide
Cache “Tags”

• Need way to tell if have copy of location in memory so that can decide on hit or miss
• On cache miss, put memory address of block in “tag address” of cache block
  1022 placed in tag next to data from memory (99)

<table>
<thead>
<tr>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>252</td>
<td>12</td>
</tr>
<tr>
<td>1022</td>
<td>99</td>
</tr>
<tr>
<td>131</td>
<td>7</td>
</tr>
<tr>
<td>2041</td>
<td>20</td>
</tr>
</tbody>
</table>

From earlier instructions
Anatomy of a 16 Byte Cache, 4 Byte Block

• Operations:
  1. Cache Hit
  2. Cache Miss
  3. Refill cache from memory

• Cache needs Address Tags to decide if Processor Address is a Cache Hit or Cache Miss
  – Compares all 4 tags
Suppose processor now requests location 511, which contains 11?

Doesn’t match any cache block, so must “evict” one resident block to make room
– Which block to evict?

Replace “victim” with new memory block at address 511
Block Must be Aligned in Memory

• Word blocks are aligned, so binary address of all words in cache always ends in $00_{\text{two}}$
• How to take advantage of this to save hardware and energy?
• Don’t need to compare last 2 bits of 32-bit byte address (comparator can be narrower)
=> Don’t need to store last 2 bits of 32-bit byte address in Cache Tag (Tag can be narrower)
Anatomy of a 32B Cache, 8B Block

- Blocks must be aligned in pairs, otherwise could get same word twice in cache
  ⇒ Tags only have even-numbered words
  ⇒ Last 3 bits of address always 000\textsubscript{two}
  ⇒ Tags, comparators can be narrower
- Can get hit for either word in block
Hardware Cost of Cache

• Need to compare every tag to the Processor address
• Comparators are expensive
• Optimization: use 2 “sets” of data with a total of only 2 comparators
• 1 Address bit selects which set
• Compare only tags from selected set
• Generalize to more sets
Processor Address Fields used by Cache Controller

- **Block Offset**: Byte address within block
- **Set Index**: Selects which set
- **Tag**: Remaining portion of processor address

<table>
<thead>
<tr>
<th>Processor Address (32-bits total)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tag</td>
</tr>
</tbody>
</table>

- Size of Index = $\log_2$ (number of sets)
- Size of Tag = Address size – Size of Index – $\log_2$ (number of bytes/block)
What is limit to number of sets?

• For a given total number of blocks, we can save more comparators if have more than 2 sets

• Limit: As Many Sets as Cache Blocks => only one block per set – only needs one comparator!

• Called “Direct-Mapped” Design

| Tag | Index | Block offset |
Direct Mapped Cache Ex: Mapping a 6-bit Memory Address

- In example, block size is 4 bytes/1 word
- Memory and cache blocks always the same size, unit of transfer between memory and cache
- # Memory blocks >> # Cache blocks
  - 16 Memory blocks = 16 words = 64 bytes => 6 bits to address all bytes
  - 4 Cache blocks, 4 bytes (1 word) per block
  - 4 Memory blocks map to each cache block
- Memory block to cache block, aka **index**: middle two bits
- Which memory block is in a given cache block, aka **tag**: top two bits
One More Detail: Valid Bit

• When start a new program, cache does not have valid information for this program
• Need an indicator whether this tag entry is valid for this program
• Add a “valid bit” to the cache tag entry
  0 => cache miss, even if by chance, address = tag
  1 => cache hit, if processor address = tag
Caching: A Simple First Example

Q: Where in the cache is the mem block?
Use next 2 low-order memory address bits – the index – to determine which cache block (i.e., modulo the number of blocks in the cache)

Q: Is the memory block in cache?
Compare the cache tag to the high-order 2 memory address bits to tell if the memory block is in the cache (provided valid bit is set)

One word blocks
Two low order bits (xx) define the byte in the block (32b words)
Direct-Mapped Cache Example

- One word blocks, cache size = 1K words (or 4KB)

Valid bit ensures something useful in cache for this index

Compare Tag with upper part of Address to see if a Hit

What kind of locality are we taking advantage of?

Read data from cache instead of memory if a Hit

Valid bit
Hit
Data

Tag
Index
Data

Byte offset

31 30 ... 13 12 11 ... 2 1 0

Index
Valid
Tag
Data

0
1
2
...
1021
1022
1023

20

Comparator

32
Multiword-Block Direct-Mapped Cache

- Four words/block, cache size = 1K words

What kind of locality are we taking advantage of?
Cache Names for Each Organization

- "Fully Associative": Block can go anywhere
  - First design in lecture
  - Note: No Index field, but 1 comparator/block
- "Direct Mapped": Block goes one place
  - Note: Only 1 comparator
  - Number of sets = number of blocks
- "N-way Set Associative": N places for a block
  - Number of sets = number of blocks / N
  - N comparators
  - *Fully Associative: N = number of blocks*
  - *Direct Mapped: N = 1*
Range of Set-Associative Caches

• For a fixed-size cache, and a given block size, each increase by a factor of 2 in associativity doubles the number of blocks per set (i.e., the number of “ways”) and halves the number of sets –
  • decreases the size of the index by 1 bit and increases the size of the tag by 1 bit

More Associativity (more ways)

| Tag | Index | Block offset |

What if we can also change the block size?
Clickers/Peer Instruction

- For a cache with constant total capacity, if we increase the number of ways by a factor of 2, which statement is false:
- A: The number of sets could be doubled
- B: The tag width could decrease
- C: The block size could stay the same
- D: The block size could be halved
- E: Tag width must increase
Total Cash Capacity =

Associativity * # of sets * block_size

Bytes = blocks/set * sets * Bytes/block

\[ C = N \times S \times B \]

<table>
<thead>
<tr>
<th>Tag</th>
<th>Index</th>
<th>Byte Offset</th>
</tr>
</thead>
</table>

address_size = tag_size + index_size + offset_size

= tag_size + \log_2(S) + \log_2(B)

Clicker Question: C remains constant, S and/or B can change such that

\[ C = 2N \times (SB)' \Rightarrow (SB)' = SB/2 \]

Tag_size = address_size − (\log_2(S) + \log_2(B)) = address_size − \log_2(SB)

= address_size − (\log_2(SB) − 1)
Typical Memory Hierarchy

- **Principle of locality + memory hierarchy** presents programmer with ≈ as much memory as is available in the *cheapest* technology at the ≈ speed offered by the *fastest* technology.
Handling Stores with Write-Through

• Store instructions write to memory, changing values
• Need to make sure cache and memory have same values on writes: 2 policies

1) **Write-Through Policy**: write cache and write *through* the cache to memory
   – Every write eventually gets to memory
   – Too slow, so include Write Buffer to allow processor to continue once data in Buffer
   – Buffer updates memory in parallel to processor
Write-Through Cache

• Write both values in cache and in memory
• Write buffer stops CPU from stalling if memory cannot keep up
• Write buffer may have multiple entries to absorb bursts of writes
• What if store misses in cache?
Handling Stores with Write-Back

2) Write-Back Policy: write only to cache and then write cache block back to memory when evict block from cache

- Writes collected in cache, only single write to memory per block
- Include bit to see if wrote to block or not, and then only write back if bit is set
  - Called “Dirty” bit (writing makes it “dirty”)
Write-Back Cache

- Store/cache hit, write data in cache *only* & set dirty bit
  - Memory has stale value
- Store/cache miss, read data from memory, then update and set dirty bit
  - “Write-allocate” policy
- Load/cache hit, use value from cache
- On any miss, write back evicted block, only if dirty. Update cache with new block and clear dirty bit.
Write-Through vs. Write-Back

• Write-Through:
  – Simpler control logic
  – More predictable timing simplifies processor control logic
  – Easier to make reliable, since memory always has copy of data (big idea: Redundancy!)

• Write-Back
  – More complex control logic
  – More variable timing (0, 1, 2 memory accesses per cache access)
  – Usually reduces write traffic
  – Harder to make reliable, sometimes cache has only copy of data
And In Conclusion, ...

- Principle of Locality for Libraries / Computer Memory
- Hierarchy of Memories (speed/size/cost per bit) to Exploit Locality
- Cache – copy of data lower level in memory hierarchy
- Direct Mapped to find block in cache using Tag field and Valid bit for Hit
- Cache design choice:
  - Write-Through vs. Write-Back