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

Review

- 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
- Larger caches reduce Miss rate via Temporal and Spatial Locality, but can increase Hit time
- Larger blocks to reduces Miss rate via Spatial Locality, but increase Miss penalty
- AMAT (Average Memory Access Time) helps balance Hit time, Miss rate, Miss penalty

Agenda

- Intro to Caches, revised
- Cache Hits and Misses, Consistency
- Administrivia
- Cache Performance and Size
- Cache blocking (if time permits)

Cache Philosophy

- Programmer invisible hardware mechanism to give illusion of speed of fastest memory with size of largest memory
  - That is, it works fine even if programmer have not 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

Starting at Beginning

- Load word instruction: lw $t0, 0($t1)
- $t1 contains 1022, Memory[1022] = 99
- No cache
  1. Processor issues address 1022 to Memory
  2. Memory reads word at address 1022 (99)
  3. Memory sends 99 to Processor
  4. Processor loads 99 into register $t1
**Starting at Beginning**

- Load word instruction: `lw $t0, 0 ($t1)`
- `$t1` contains `1022_{10}`, Memory[1022] = 99
- With cache (similar to a hash)
  1. Processor issues address `1022_{10}` to Cache
  2. Cache checks to see if have copy of data that matches address `1022_{10}`
     - If finds a match (Hit): cache reads 99, sends to processor
     - No match (Miss): cache sends address 1022 to Memory
   1. Memory reads 99 at address 1022
   2. Memory sends 99 to Cache
   3. Cache replaces word with new 99
   4. Cache sends 99 to processor
- Processor loads 99 into register `$t1`

**Cache Requirements**

- 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 Tag to decide if Processor Address is a Cache Hit or Cache Miss
  - Compares all 4 tags

<table>
<thead>
<tr>
<th>Processor</th>
<th>Cache</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>32-bit Address</td>
<td>32-bit Data</td>
<td>32-bit Address</td>
</tr>
<tr>
<td>252</td>
<td>12</td>
<td>1022</td>
</tr>
<tr>
<td>131</td>
<td>7</td>
<td>2041</td>
</tr>
</tbody>
</table>

**Cache Requirements**

- Suppose processor requests location 511, which contains 11?
- Doesn’t match any cache block, so drop one block
  - Tries to find Least Recently Used block: location 131
- Replace with memory block corresponding to 511

<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>511</td>
<td>11</td>
</tr>
<tr>
<td>2041</td>
<td>20</td>
</tr>
</tbody>
</table>

**Block Must Be Aligned in Memory**

- Word blocks are aligned, so binary address of all words in cache always ends in 00_{two}
- How 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_{two}
  ⇒ Tag, comparator can be narrower
  ⇒ Can get hit for either word in block
Cache Optimizations

- Reduce tag overhead by having larger blocks
  - E.g., 2 words, 4 words, 8 words
- Separate caches for instructions and data
  - Double bandwidth, don’t interfere with each other
- Bigger caches (but access time could get bigger than one clock cycle if too big)
- Divide cache into multiple sets, only search inside one set => saves comparators, energy
  - If as many sets as blocks, then only 1 comparator (aka Direct Mapped Cache)
  - But may increase Miss Rate

Handling Cache Misses

- Write-Through Policy
  - Stall execution, fetch the block from the next level in the memory hierarchy, install it in cache, send requested word to processor, and then let execution resume
- Write misses
  - Write allocate: Stall execution, fetch the block from next level in memory hierarchy, install it in cache, write the word from processor to cache, also update memory, then let execution resume
  - No-write allocate: skip the cache write and just write the word to memory (but must invalidate cache block since it will now hold stale data)

Cache-Memory Consistency?

- 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
  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”)

Administrivia

- Lab #6 posted
- Hundreds of students using GitHub successfully
  - From now on, lose 2 points if GSIs need to repair your account
- Project #2, Part 2 Due Sunday @ 11:59:59
- No Homework this week!
- Midterm in 2 weeks:
  - TA Review: Su, Mar 4, starting 2 PM, 2050 VLSB
  - Exam: Tu, Mar 6, 6:40-9:40 PM, 2050 VLSB (room change)
  - Covers everything through lecture Tue Feb 28
  - TA Review: Su, Mar 6, 2-5 PM, 2050 VLSB
  - Closed book, can bring one sheet notes, both sides
  - Copy of Green card will be supplied
  - No phones, calculators, ...; just bring pencils & eraser

Getting to Know Pros: Family

- Parents, grandparents moved to California
- (Still) married to high school sweetheart
- Parents retired, so sold homestead
- Spend 1 week/summer hosting Patterson Family Reunion
  - I am tour guide

27 people: 2 parents, 3 siblings, 2 sons, 7 nieces and nephews, 8 in-laws, 3 grandchildren, 1 grandnephew, 1 grandniece, 6 dogs ...
Average Memory Access Time (AMAT)

- Average Memory Access Time (AMAT) is the average to access memory considering both hits and misses.

\[ \text{AMAT} = \text{Time for a hit} + \text{Miss rate} \times \text{Miss penalty} \]

- How calculate if separate instruction and data caches?

AMAT for I$ and D$

- E.g., CPI\text{ideal} of 2.
  - 100 cycle miss penalty (to main memory), 36% load/stores, a 2% L1 Instruction Cache (I$) miss rate, a 4% L1 Data Cache (D$) miss rate, \( \text{CPI}_{\text{stalls}} = 2 + .02 \times 100 + .04 \times 100 \)
  - \( = 2 + 2 + 1.44 \)
  - \( = 5.44 \)
- How reduce Miss Penalty?

Local vs. Global Miss Rates

- Local miss rate – the fraction of references to one level of a cache that miss.
  - Local Miss rate L2$ = L2$ Misses / L1$ Misses
- Global miss rate – the fraction of references that miss in all levels of a multilevel cache
  - L2$ local miss rate >> than the global miss rate
  - Global Miss rate = L2$ Misses / Total Accesses
  - AMAT = Time for a hit + Miss rate \times Miss penalty
  - AMAT = Time for a L1$ hit + (local) Miss rate \times \text{L1$ Misses} \times \text{L2$ Miss penalty}

Improving Cache Performance (1 of 3)

1. Reduce the time to hit in the cache
   - Smaller cache
2. Reduce the miss rate
   - Bigger cache
   - Larger blocks (16 to 64 bytes typical)
   - (Later in semester: More flexible placement by increasing associativity)

Reducing Cache Miss Penalty

- Use multiple cache levels!
- With Moore’s Law, have more room on die for bigger L1 caches and for second level cache – normally a unified L2 cache (i.e., it holds both instructions and data), and in some cases even a unified L3 cache
- E.g., CPI\text{ideal} of 2.
  - 100 cycle miss penalty (to main memory), 25 cycle miss penalty (to L2$), 36% load/stores, a 2% (4%) L1$ (I$) miss rate, add a 0.5% L2$ miss rate
  - \( \text{CPI}_{\text{stalls}} = 2 + .02 \times 25 + .04 \times 4 \times 25 + .005 \times 100 + .36 \times .005 \times 100 \)
  - \( = 3.54 \) (vs. 5.44 with no L2$)

Agenda

- Intro to Caches, revised
- Cache Hits and Misses, Consistency
- Administrivia
- Cache Performance and Size
- Cache blocking (if time permits)
Improving Cache Performance (2 of 3)

3. Reduce the miss penalty
   – Smaller blocks
   – Use multiple cache levels
     • L2 cache not tied to processor clock rate
   – Higher DRAM memory bandwidth (faster DRAMs)
   – Use a write buffer to hold dirty blocks being replaced so don’t have to wait for the write to complete before reading

The Cache Design Space (3 of 3)

• Several interacting dimensions
  – Cache size
  – Block size
  – Write-through vs. write-back
  – Write allocation
  – (Later Associativity)
  – (Later Replacement policy)
• Optimal choice is a compromise
  – Depends on access characteristics
    • Workload
      • Use (L-cache, D-cache)
    – Depends on technology / cost
  • Simplicity often wins

Multilevel Cache Design Considerations

• Different design considerations for L1$ and L2$
  – L1 focuses on minimizing hit time for shorter clock cycle: Smaller $ with smaller block sizes
  – L2(s) focus on reducing miss rate to reduce penalty of long main memory access times: Larger $ with larger block sizes
• Miss penalty of L1$ is significantly reduced by presence of L2$, so can be smaller/faster but with higher miss rate
• For the L2$, hit time is less important than miss rate
  – L2$ hit time determines L1$’s miss penalty

Fields within an Address

- For a direct mapped cache with $2^n$ blocks, $n$ bits are used for the index
- For a block size of $2^m$ words ($2^{m+2}$ bytes), $m$ bits are used to address the word within the block and 2 bits are used to address the byte within the word: block offset
- Size of the tag field is Address size – index size – block offset size
  – 32-bit byte address => 32 – $n$ – ($m+2$)

Anatomy of a 32B Cache, 8B Block

• Suppose Direct Mapped
  – 4 blocks
  – 8 bytes per block
  – 32 bit address
• Bits of Block Offset field of address?
• Bits of Index field of address?
• Bits of Tag field of address?
Cache Sizes

- Number of bits in a direct-mapped cache includes both the storage for data and for the tags + valid bit + dirty bit (if needed)
- Total number of bits in a cache is then $2^n \times (\text{block size} + \text{tag field size} + \text{valid field size} + \text{dirty field size if needed})$
- Why don't need to store Block Offset in Cache? Why not Index in Cache?

Assuming a direct-mapped, write-through cache with 16 KB of data and 4-word blocks, how divide a 32-bit byte address to access a cache?

- Tag <14 bits> | Index <14 bits> | Block Offset <4 bits>
- Valid <1> | Tag <14 bits> | Index <14 bits> | Block Offset <4 bits>
- Tag <18 bits> | Index <10 bits> | Block Offset <4 bits>
- Valid <1> | Tag <18 bits> | Index <10 bits> | Block Offset <4 bits>

CPI/Miss Rates/DRAM Access

### SpecInt2006

<table>
<thead>
<tr>
<th>Name</th>
<th>CPI</th>
<th>Data Only</th>
<th>Data Only</th>
<th>Instructions and Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>perl</td>
<td>0.79</td>
<td>3.5</td>
<td>1.1</td>
<td>1.5</td>
</tr>
<tr>
<td>bzip2</td>
<td>0.85</td>
<td>11.0</td>
<td>5.8</td>
<td>2.5</td>
</tr>
<tr>
<td>gcc</td>
<td>1.72</td>
<td>24.0</td>
<td>12.4</td>
<td>14.8</td>
</tr>
<tr>
<td>mcf</td>
<td>10.0</td>
<td>106.8</td>
<td>88.0</td>
<td>88.5</td>
</tr>
<tr>
<td>gzip</td>
<td>2.09</td>
<td>4.5</td>
<td>1.4</td>
<td>1.7</td>
</tr>
<tr>
<td>parser</td>
<td>0.80</td>
<td>4.4</td>
<td>2.9</td>
<td>0.6</td>
</tr>
<tr>
<td>spam</td>
<td>0.06</td>
<td>1.3</td>
<td>0.6</td>
<td>0.6</td>
</tr>
<tr>
<td>Broadcast</td>
<td>1.61</td>
<td>30.0</td>
<td>22.1</td>
<td>47.7</td>
</tr>
<tr>
<td>h264ref</td>
<td>0.60</td>
<td>6.4</td>
<td>1.6</td>
<td>0.2</td>
</tr>
<tr>
<td>omnet++</td>
<td>2.34</td>
<td>30.9</td>
<td>27.7</td>
<td>29.8</td>
</tr>
<tr>
<td>openmp</td>
<td>1.79</td>
<td>16.5</td>
<td>9.2</td>
<td>8.2</td>
</tr>
<tr>
<td>solitaire</td>
<td>2.70</td>
<td>38.0</td>
<td>35.8</td>
<td>11.4</td>
</tr>
<tr>
<td>tetgen</td>
<td>1.25</td>
<td>13.6</td>
<td>7.5</td>
<td>5.4</td>
</tr>
</tbody>
</table>

Performance Programming: Adjust software accesses to improve miss rate

- Now that understand how caches work, can revise program to improve cache utilization
  - Cache size
  - Block size
  - Multiple levels
Performance of Loops and Arrays

- Array performance often limited by memory speed
- OK if access memory different order as long as get correct result
- Goal: Increase performance by minimizing traffic from cache to memory
  - That is, reduce Miss rate by getting better reuse of data already in cache
- One approach called Cache Blocking:
  "shrink" problem by performing multiple iterations within smaller cache blocks
- Use Matrix Multiply as example: Next Lab and Project 3

Matrix Multiplication

Assumption: the matrices are stored as 2-D N x N arrays

```
for (i=0;i<N;i++)
    for (j=0;j<N;j++)
        for (k=0;k<N;k++)
            c[i][j] += a[i][k] * b[k][j];
```

Advantage: code simplicity
Disadvantage: Marches through memory and caches

Note on Matrix in Memory

- A matrix is a 2-D array of elements, but memory addresses are "1-D"
- Conventions for matrix layout
  - by column, or "column major" (Fortran default); A(i,j) at A+i+j
  - by row, or "row major" (C default) A(i,j) at A+i*n+j

Improving reuse via Blocking:
1st "Naïve" Matrix Multiply

```
for i = 1 to n
    (read row i of A into cache)
    for j = 1 to n
        (read column j of B into cache)
        for k = 1 to n
            c[i][j] = c[i][j] + a[i][k] * b[k][j]
            (write c[i][j] back to main memory)
```

Simple Matrix Multiply - www.youtube.com/watch?v=q0TcDHxw
100 x 100 Matrix, Cache 1000 blocks, 1 word/block
Linear Algebra to the Rescue!
- Instead of multiplying two, say, 6 x 6 matrices

\[ A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}, \quad B = \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix} \]

Where \( A_{ij} = \begin{bmatrix} a_{ij} & a_{ij} & a_{ij} \\ a_{ij} & a_{ij} & a_{ij} \end{bmatrix} \)

\[ AB = \begin{bmatrix} a_{11}b_{11} + a_{12}b_{21} + a_{13}b_{31} & \cdots & a_{11}b_{1n} + a_{12}b_{2n} + a_{13}b_{3n} \\ \vdots & \ddots & \vdots \\ a_{ij}b_{1n} + a_{ij}b_{2n} + a_{ij}b_{3n} & \cdots & a_{ij}b_{jn} + a_{ij}b_{jn} + a_{ij}b_{kn} \end{bmatrix} \]

- Thus, can get same result as multiplication of a set of submatrices

Another View of “Blocked” Matrix Multiplication

\[ C_{ij} = A_{ij}B_{ij} \]

\[ C_{ij} = \sum_{k=1}^{n} A_{ik}B_{kj} \]

Main Point: each multiplication operates on small “block” matrices, whose size may be chosen so that they fit in the cache.

Sources of Cache Misses (3 C’s)
- Compulsory (cold start, first reference):
  - 1st access to a block, “cold” fact of life, not a lot you can do about it.
  - If running billions of instruction, compulsory misses are insignificant
- Capacity: Cache cannot contain all blocks accessed by the program
- Conflict (collisions): Multiple memory locations mapped to the same cache location

Read
- To access cache, Memory Address divided into 3 fields: Tag, Index, Block Offset
- Cache size is Data + Management (tags, valid, dirty bits)
- Write misses trickier to implement than reads
  - Write back vs. Write through
  - Write allocate vs. No write allocate
- Cache Performance Equations:
  - \( CPI = IC \times CP_{pate} \times CC \)
  - \( IC = \frac{CPU\ time}{IC\ plus\ Memory\ stall\ cycles} \times CC \)
  - \( AMAT = \frac{Time\ for\ a\ hit}{Miss\ rate \times Miss\ penalty} \)
- If understand caches, can adapt software to improve cache performance and thus program performance

Maximum Block Size
- The blocking optimization works only if the blocks fit in cache.
- That is, 3 blocks of size \( r \times r \) must fit in memory (for A, B, and C)
- \( M = \text{size of cache (in elements/words)} \)
- \( M = \text{size of cache (in elements/words)} \)
- We must have: \( 3r^2 = M, \text{ or } r = \sqrt{M}/3 \)
- Ratio of cache misses blocked vs. unblocked up to \( = \sqrt{M} \)

Blocked Matrix Multiply
Consider \( A, B, C \) to be \( N \times N \) matrices of \( b \times b \) subblocks where \( b=\sqrt{N} \) is called the block size

\[ \text{for } i = 1 \text{ to } N \]
\[ \text{for } j = 1 \text{ to } N \]

(read block \( C(i,j) \) into cache)

(read block \( A(i,k) \) into cache)

\( C(i,j) = C(i,j) + A(i,k) \times B(k,j) \) (do a matrix multiply on blocks)

(write block \( C(i,j) \) back to main memory)

Review
- To access cache, Memory Address divided into 3 fields: Tag, Index, Block Offset
- Cache size is Data + Management (tags, valid, dirty bits)
- Write misses trickier to implement than reads
  - Write back vs. Write through
  - Write allocate vs. No write allocate
- Cache Performance Equations:
  - \( CPI = IC \times CP_{pate} \times CC \)
  - \( IC = CPU\ time/(IC\ plus\ Memory\ stall\ cycles) \times CC \)
  - \( AMAT = Time\ for\ a\ hit \times Miss\ rate \times Miss\ penalty \)
- If understand caches, can adapt software to improve cache performance and thus program performance