Review

• Although caches are software-invisible, a "cache-aware" performance programmer can improve performance by large factors by changing order of memory accesses
• Three C’s of cache misses
  – Compulsory
  – Capacity
  – Conflict

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 instructions, compulsory misses are insignificant

• **Capacity**:
  – Cache cannot contain all blocks accessed by the program
  • Misses that would not occur with infinite cache

• **Conflict** (collision):
  – Multiple memory locations mapped to the same cache location
  • Misses that would not occur with ideal fully associative cache

Flashcard Quiz: With a fixed cache capacity, what effect does a larger block size have on the 3Cs?

- **Decrees compulsory, increases conflicts**
- **Increases conflicts**
- **Decreases compulsory, decreases capacity misses**

Flashcard Quiz: With a fixed cache block size, what effect does a larger cache capacity have on the 3Cs?

- **Increases compulsory, decreases conflicts**
- **Decreases compulsory, increases conflicts, decreases capacity misses**
- **Decreases compulsory, decreases conflicts**
- **Decreases conflicts, decreases capacity misses**

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 instructions, compulsory misses are insignificant
  • Solution: increase block size (increases miss penalty; very large blocks could increase miss rate)

• **Capacity**:
  – Cache cannot contain all blocks accessed by the program
  • Solution: increase cache size (may increase access time)
  • Or structure software so reuse data in cache before fetching new data

• **Conflict** (collision):
  – Multiple memory locations mapped to the same cache location
  • Solution 1: Increase cache size (may increase hit time)
  • Solution 2: (later in semester) increase associativity (may increase hit time)
New-School Machine Structures
(It’s a bit more complicated!)

• Parallel Requests
  Assigned to computer
  e.g., Search “Kats”

• 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

Alternative Kinds of Parallelism: The Programming Viewpoint

• Job-level parallelism/process-level parallelism
  – Running independent programs on multiple processors simultaneously
  – Example?

• Parallel-processing program
  – Single program that runs on multiple processors simultaneously
  – Example?

Alternative Kinds of Parallelism: Single-Instruction/Single-Data Stream

• Single Instruction, Single Data stream (SISD)
  – Sequential computer that exploits no parallelism in either the instruction or data streams. Examples of SISD architecture are traditional uniprocessor machines

Alternative Kinds of Parallelism: Single-Instruction/Multiple-Data Stream

• Single-Instruction, Multiple-Data streams (SIMD or “sim-dee”)
  – Computer that exploits multiple data streams against a single instruction stream to operations that may be naturally parallelized, e.g., Intel SIMD instruction extensions or NVIDIA Graphics Processing Unit (GPU)

Alternative Kinds of Parallelism: Multiple-Instruction/Single-Data Stream

• Multiple-Instruction, Single-Data stream (MISD)
  – Computer that exploits multiple instruction streams against a single data stream for data operations that can be naturally parallelized. For example, certain kinds of array processors.
  – No longer commonly encountered, mainly of historical interest only

Alternative Kinds of Parallelism: Multiple-Instruction/Multiple-Data Streams

• Multiple-Instruction, Multiple-Data streams (MIMD or “mim-dee”)
  – Multiple autonomous processors simultaneously executing different instructions on different data.
  – MIMD architectures include multicore and Warehouse-Scale Computers
  – (Discuss after midterm)
In 2012, SIMD and MIMD most common parallelism in architectures – usually both in same system!

- Most common parallel processing programming style: Single Program Multiple Data (SPMD)
  - Single program that runs on all processors of a MIMD
  - Cross-processor execution coordination through conditional expressions (thread parallelism after midterm)
- SIMD (aka hw-level data parallelism): specialized function units, for handling lock-step calculations involving arrays
  - Scientific computing, signal processing, multimedia (audio/video processing)

**Big Idea: Amdahl’s (Heartbreaking) Law**

- Speedup due to enhancement $E$ is
  \[ \text{Speedup w/} E = \frac{\text{Exec time w/o} E}{\text{Exec time w/} E} \]
- Suppose that enhancement $E$ accelerates a fraction $F$ ($F < 1$) of the task by a factor $S$ ($S > 1$) and the remainder of the task is unaffected

\[ \text{Execution Time w/} E = \text{Execution Time w/o} E \times \left( \frac{1}{1-F} + \frac{F}{S} \right) \]

\[ \text{Speedup w/} E = \frac{1}{\frac{1}{1-F} + \frac{F}{S}} \]

**Example:** the execution time of half of the program can be accelerated by a factor of 2. What is the program speed-up overall?

\[ \frac{1}{0.5 + 0.25} = \frac{1}{0.75} = 1.33 \]

**Big Idea: Amdahl’s Law**

- Speedup

\[ \text{Speedup} = \frac{1}{(1 - F) + \frac{F}{S}} \]

**Example:** the execution time of half of the program can be accelerated by a factor of 2. What is the program speed-up overall?

\[ \frac{1}{0.5 + 0.25} = \frac{1}{0.75} = 1.33 \]

**Administrivia**

- Lab #6 posted
- Midterm Tuesday Oct 9, 8PM:
  - Two rooms: 1 Pimentel and 2050 LSB
  - Check your room assignment!
  - Covers everything through lecture today
  - Closed book, can bring one sheet notes, both sides
  - Copy of Green card will be supplied
  - No phones, calculators, ... just bring pencils & eraser
"Computer Science is the most popular major at Stanford." 45% of MIT undergrads enrolled in EECs. Also now probably most popular major at UCB.

**Example #1: Amdahl’s Law**

\[ \text{Speedup} \frac{1}{E} = 1 / \left[ (1 - F) + F/S \right] \]

- Consider an enhancement which runs 20 times faster but which is only usable 25% of the time
  
  \[ \text{Speedup} w/ E = 1 / (1.75 + 0.25/100) = 1.31 \]

- What if its usable only 15% of the time?
  
  \[ \text{Speedup} w/ E = 1 / (0.85 + 0.15/20) = 1.17 \]

- Amdahl’s Law tells us that to achieve linear speedup with 100 processors, none of the original computation can be scalar!

- To get a speedup of 90 from 100 processors, the percentage of the original program that could be scalar would have to be 0.1% or less
  
  \[ \text{Speedup} w/ E = 1 / (0.001 + .999/100) = 90.99 \]

**Example #2: Amdahl’s Law**

\[ \frac{1}{E} = 1 / \left[ (1 - F) + F/S \right] \]

- Consider summing 10 scalar variables and two 10 by 10 matrices (matrix sum) on 10 processors
  
  \[ \text{Speedup} w/ E = 1 / (0.091 + 0.909/10) = 1.0181 + 5.5 \]

- What if there are 100 processors?
  
  \[ \text{Speedup} w/ E = 1 / (0.991 + .009/100) = 1/0.1099 + 10.0 \]

- What if the matrices are 100 by 100 (or 10,010 adds in total)
  
  \[ \text{Speedup} w/ E = 1 / (0.001 + .999/100) = 1/0.1099 + 9.9 \]

- What if there are 100 processors?
  
  \[ \text{Speedup} w/ E = 1 / (0.001 + .999/100) = 1/0.01099 + 91 \]

**Strong and Weak Scaling**

- To get good speedup on a multiprocessor while keeping the problem size fixed is harder than getting good speedup by increasing the size of the problem.
  
  - **Strong scaling**: when speedup can be achieved on a parallel processor by increasing the size of the problem proportionally to the increase in the number of processors
  
  - **Weak scaling**: when speedup is achieved on a parallel processor by increasing the size of the problem

- Load balancing is another important factor: every processor doing same amount of work
  
  - Just one unit with twice the load of others cuts speedup almost in half
Suppose a program spends 80% of its time in a square root routine. How much must you speedup square root to make the program run 5 times faster?

\[
\text{Speedup with } E = \frac{1}{(1-F) \times F/S}
\]

- 10
- 20
- 100
- None of the Above

**SIMD Architectures**

- Data parallelism: executing one operation on multiple data streams
  - Example to provide context:
    - Multiplying a coefficient vector by a data vector (e.g., in filtering)
    - \[ y[i] := c[i] \times x[i], 0 \leq i < n \]
  - Sources of performance improvement:
    - One instruction is fetched & decoded for entire operation
    - Multiplications are known to be independent
    - Pipelining/concurrency in memory access as well

**“Advanced Digital Media Boost”**

- To improve performance, Intel’s SIMD instructions
  - Fetch one instruction, do the work of multiple instructions
  - MMX (MultiMedia eXtension, Pentium II processor family)
  - SSE (Streaming SIMD Extension, Pentium III and beyond)

**Example: SIMD Array Processing**

\[
\text{for each } f \text{ in array } \\
\quad f = \sqrt{f} \\
\text{for each } f \text{ in array } \\
\quad \{ \\
\quad \text{load } f \text{ to the floating-point register} \\
\quad \text{calculate the square root} \\
\quad \text{write the result from the register to memory} \}
\]

\[
\text{for each 4 members in array } \\
\quad \{ \\
\quad \text{load 4 members to the SSE register} \\
\quad \text{calculate 4 square roots in one operation} \\
\quad \text{store the 4 results from the register to memory} \}
\]

**Data-Level Parallelism and SIMD**

- SIMD wants adjacent values in memory that can be operated in parallel
- Usually specified in programs as loops
  \[
  \text{for(i=1000; i>0; i--) } \\
  x[i] = x[i] + s;
  \]
- How can reveal more data-level parallelism than available in a single iteration of a loop?
  - **Unroll loop** and adjust iteration rate

**Looping in MIPS**

Assumptions:
- \$t1 is initially the address of the element in the array with the highest address
- \$t0 contains the scalar value \( s \)
- \$t(32) is the address of the last element to operate on

CODE:

\[
\text{Loop:1. } \text{l.d } \$f2,0(\$t1) ; \$f2= array element} \\
2. \text{add.d } \$f10,\$f2,\$f0 ; \text{add to } \$f2 \\
3. \text{s.d } \$f10,0(\$t1) ; \text{store result} \\
4. \text{addui } \$t1,\$t1,\#-8 ; \text{decrement pointer 8 byte} \\
5. \text{bne } \$t1,\$t2,\text{Loop; repeat loop if } \$t1 \neq \$t2
\]
Loop Unrolled

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Address</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>l.d</td>
<td>$f2,0($t1)</td>
<td>load double-precision floating-point value to $f2</td>
</tr>
<tr>
<td>add.d</td>
<td>$f10,$f2,$f0</td>
<td>add two double-precision floating-point values and store the result in $f10</td>
</tr>
<tr>
<td>s.d</td>
<td>$f10,0($t1)</td>
<td>store double-precision floating-point value to memory at address $f10</td>
</tr>
<tr>
<td>l.d</td>
<td>$f4,-8($t1)</td>
<td>load double-precision floating-point value to $f4</td>
</tr>
<tr>
<td>add.d</td>
<td>$f12,$f4,$f0</td>
<td>add two double-precision floating-point values and store the result in $f12</td>
</tr>
<tr>
<td>s.d</td>
<td>$f12,-8($t1)</td>
<td>store double-precision floating-point value to memory at address $f12</td>
</tr>
<tr>
<td>l.d</td>
<td>$f6,-16($t1)</td>
<td>load double-precision floating-point value to $f6</td>
</tr>
<tr>
<td>add.d</td>
<td>$f14,$f6,$f0</td>
<td>add two double-precision floating-point values and store the result in $f14</td>
</tr>
<tr>
<td>s.d</td>
<td>$f14,-16($t1)</td>
<td>store double-precision floating-point value to memory at address $f14</td>
</tr>
<tr>
<td>l.d</td>
<td>$f8,-24($t1)</td>
<td>load double-precision floating-point value to $f8</td>
</tr>
<tr>
<td>add.d</td>
<td>$f16,$f8,$f0</td>
<td>add two double-precision floating-point values and store the result in $f16</td>
</tr>
<tr>
<td>s.d</td>
<td>$f16,-24($t1)</td>
<td>store double-precision floating-point value to memory at address $f16</td>
</tr>
<tr>
<td>addui</td>
<td>$t1,$t1,#-32</td>
<td>add immediate value to $t1</td>
</tr>
<tr>
<td>bne</td>
<td>$t1,$t2,Loop</td>
<td>branch if not equal to</td>
</tr>
</tbody>
</table>

NOTE:
1. Only 1 Loop overhead every 4 iterations
2. This unrolling works if loop_limit(mod 4) = 0
3. (Different Registers eliminate stalls in pipeline; we'll see later in course)

Loop Unrolled Scheduled

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Address</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>l.d</td>
<td>$f2,0($t1)</td>
<td>load double-precision floating-point value to $f2</td>
</tr>
<tr>
<td>l.d</td>
<td>$f4,-8($t1)</td>
<td>load double-precision floating-point value to $f4</td>
</tr>
<tr>
<td>l.d</td>
<td>$f6,-16($t1)</td>
<td>load double-precision floating-point value to $f6</td>
</tr>
<tr>
<td>l.d</td>
<td>$f8,-24($t1)</td>
<td>load double-precision floating-point value to $f8</td>
</tr>
<tr>
<td>add.d</td>
<td>$f10,$f2,$f0</td>
<td>add two double-precision floating-point values and store the result in $f10</td>
</tr>
<tr>
<td>add.d</td>
<td>$f12,$f4,$f0</td>
<td>add two double-precision floating-point values and store the result in $f12</td>
</tr>
<tr>
<td>add.d</td>
<td>$f14,$f6,$f0</td>
<td>add two double-precision floating-point values and store the result in $f14</td>
</tr>
<tr>
<td>add.d</td>
<td>$f16,$f8,$f0</td>
<td>add two double-precision floating-point values and store the result in $f16</td>
</tr>
<tr>
<td>s.d</td>
<td>$f10,0($t1)</td>
<td>store double-precision floating-point value to memory at address $f10</td>
</tr>
<tr>
<td>s.d</td>
<td>$f12,-8($t1)</td>
<td>store double-precision floating-point value to memory at address $f12</td>
</tr>
<tr>
<td>s.d</td>
<td>$f14,-16($t1)</td>
<td>store double-precision floating-point value to memory at address $f14</td>
</tr>
<tr>
<td>s.d</td>
<td>$f16,-24($t1)</td>
<td>store double-precision floating-point value to memory at address $f16</td>
</tr>
<tr>
<td>addui</td>
<td>$t1,$t1,#-32</td>
<td>add immediate value to $t1</td>
</tr>
<tr>
<td>bne</td>
<td>$t1,$t2,Loop</td>
<td>branch if not equal to</td>
</tr>
</tbody>
</table>

4 Loads side-by-side: Could replace with 4-wide SIMD Load
4 Adds side-by-side: Could replace with 4-wide SIMD Add
Stores side-by-side: Could replace with 4-wide SIMD Store

Loop Unrolling in C

• Instead of compiler doing loop unrolling, could do it yourself in C
  for(i=1000; i>0; i=i-1)
    x[i] = x[i] + s;
• Could be rewritten
  for(i=1000; i>0; i=i-4) {
    x[i] = x[i] + s;
    x[i-1] = x[i-1] + s;
    x[i-2] = x[i-2] + s;
    x[i-3] = x[i-3] + s;
  }

Generalizing Loop Unrolling

• A loop of n iterations
• k copies of the body of the loop
• Assuming (n mod k) ≠ 0
  Then we will run the loop with 1 copy of the body (n mod k) times and with k copies of the body floor(n/k) times
• (Will revisit loop unrolling again when get to pipelining later in semester)

Review

• Flynn Taxonomy of Parallel Architectures
  – SIMD: Single Instruction Multiple Data
  – MIMD: Multiple Instruction Multiple Data
  – SISD: Single Instruction Single Data (sequential machines)
  – MISD: Multiple Instruction Single Data (unused)
• Amdahl’s Law
  – Strong versus weak scaling
• SIMD Extensions
  – Exploit data-level parallelism in loops