Yet More To Consider in Understanding Performance

• We have implicitly assumed that an operation takes one clock cycle.
  ♦ This is rarely true!

• So why can we assume that instructions take one cycle?
  ♦ Because most operations can be started every cycle
    • Each operation is divided into several steps
    • A different step is performed for each operation during a clock cycle
    • This approach is called pipelining
  • Not all instructions can be pipelined!

♦ Impact on algorithms and programming models
  • Full performance requires multiple, concurrent (and independent) operations
Example: Floating Point Addition
The performance measured by the STREAM benchmark is different (and lower) than the “memory bus bandwidth”. Why?

- Memory bus bandwidth is simply the width of the memory bus (in bytes) times the clock rate of the bus
  - Instantaneous rate at which data can be transferred
- Lets look at the STREAM code and see what it does
Understanding STREAM Performance

• Consider the simple case of memory copy:
  ♦ do i=1,n
      a(i) = b(i)
  enddo

♦ Suppose system memory bandwidth is 5.5GB/s. How fast will this loop execute?
BG/L Node

Critical data:
L2 Miss is about 60 cycles

Figure 4: BlueGene/L Node Diagram. The bandwidths listed are targets.
Stream Performance Estimate

- Easy estimate: \( 11 \text{ GB/s} = 2 \times 5.5 \text{ GB/Sec to L3, 5.5 GB/Sec to main memory} \)
  - Minimum link speed is 5.5 GB/s each way, Stream adds both
- Measured performance is 1.2 GB/s!
  - Why?
- Time to move each cache line
  - 5.5 GB/s \( \sim \) 8 bytes/cycle (memory bus bandwidth)
  - \( \sim \)60 cycles L2 miss (latency)
  - 64 byte cache line = 8 cycles (bandwidth) + 60 cycles (latency) = 68 cycles or \( \sim \) 1 byte/cycle (read)
  - Stream bytes read + bytes written / time, so stream estimate is \( 2 \times 1 \text{ byte/cycle}, \text{ or } 1.4 \text{ GB/sec} \)
- This is typical (if anything, better than many systems because L2 miss cost is low)
Impact of Instruction Latency

• Programming languages usually present a model in which one line (or operation) completes before the next starts
  ♦ This is not what happens in pipelined architectures (= real world)

• Algorithms often have the same feature
  ♦ After all, often written as pseudo code with these features
Another Example: Reductions

• Consider this code
  
doi=1,n
  sum = sum + a(i) * b(i)
endo

• How fast can this run (assume data already in cache)?
  ♦ Easy model: L1 Rate (needs 2 8-byte doubles/floating point add/multiply). If 32 GB/sec, then 4GFlops is possible with a 2GHz clock
  ♦ But it is not that simple...
Operation Order

- Fortran specifies the operation order. The loop defines the result as:

\[
\text{sum} = (((a_1 \times b_1) + (a_2 \times b_2)) + (a_3 \times b_3)) + (a_4 \times b_4)) + \ldots
\]
Question

• How many clock cycles does it take to compute
  \[ \text{sum} = a + (b + (c + d)) \]

• Assume
  - Each add takes 3 cycles to complete (latency)
  - Assume all data is in register (no clock cycles needed to load or store values)
  - Assume the calculation must be performed in the order written (parenthesis respected)
Impact of Implied Dependencies

- Assume each add takes 3 cycles to complete (latency)
- Since each add depends on the result of the prior add, only 1 add may be performed every 3 cycles
- Top speed reduced by a factor of 3
- But if the evaluation tree is balanced, there are separate adds (do not involve the results of an immediately prior add), those adds may overlap

A1*B1 + A2*B2 + A3*B3 + A4*B4
Impact of Implied Dependencies

• Assume each add takes 3 cycles to complete (latency)
• Since each add depends on the result of the prior add, only 1 add may be performed every 3 cycles
• Top speed reduced by a factor of 3
• But if the evaluation tree is balanced, there are separate adds (do not involve the results of an immediately prior add), those adds may overlap
Associativity

- Operations that are associative maybe evaluated in any grouping:
  - \((a*b)*c = a*(b*c)\)
- Floating point is *not* associative
  - ♦ Round-off error, e.g., can lead to large errors.
- Options available:
  - ♦ Rewrite code to provide an order of evaluation with fewer immediate dependencies
  - ♦ Tell compiler to assume all operations are associative *(all, in the entire file)*
    - Some higher-levels of optimization imply this
    - Without it, latency of pipeline operations severely limits performance
    - But with it, enabling optimization can *change the computed results!*
A Faster Reduce

• Even better is to divide the tree. Consider this code instead
  ♦ Performance of original version: 530 Mflops
  ♦ Performance of new version: 1700 Mflops

• (Caveat: The new version has some other advantages)

```plaintext
sum1 = 0.0
sum2 = 0.0
sum3 = 0.0
sum4 = 0.0
do i=1,vecSize,4
    sum1 = sum1 + vecA(i)*vecB(i)
    sum2 = sum2 + vecA(i+1) * vecB(i+1)
    sum3 = sum3 + vecA(i+2) * vecB(i+2)
    sum4 = sum4 + vecA(i+3) * vecB(i+3)
endo
dsum = (sum1 + sum2) + (sum3 + sum4)
```

Discussion: What is wrong with this approach of manually expressing the ordering?
Dot Product in More Detail

• Consider the following architecture:
  ♦ L1 latency is 3 cycles
  ♦ Floating multiplies take 5 cycles
  ♦ Floating adds take 3 cycles
  ♦ Integer operations take 1 cycle
  ♦ Comparisons and branches take 2 cycles
Example Instruction Schedule

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Load/Store</th>
<th>Multiply</th>
<th>Add</th>
<th>Instr</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>La1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Lb1</td>
<td></td>
<td>A++</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>&gt;a1</td>
<td>La2</td>
<td>B++</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Lb2</td>
<td>&gt;b1</td>
<td>A++</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>La3</td>
<td>&gt;a2</td>
<td>A1*b1</td>
<td>B++</td>
</tr>
<tr>
<td>6</td>
<td>&gt;b2</td>
<td>Lb3</td>
<td>A++</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>La4</td>
<td>&gt;a3</td>
<td>A2*b2</td>
<td>B++</td>
</tr>
<tr>
<td>8</td>
<td>Lb4</td>
<td>&gt;b3</td>
<td>A++</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>&gt;a4</td>
<td></td>
<td>A3*b3</td>
<td>B++</td>
</tr>
<tr>
<td>10</td>
<td>&gt;b4</td>
<td></td>
<td>+a1b1</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td>A4*b4</td>
<td>=</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td>+a2b2</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td>=</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td>+a3b3</td>
<td>=</td>
<td>N-=4</td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td>CMP</td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
<td>+a4b4</td>
<td>Branch</td>
</tr>
<tr>
<td>17</td>
<td></td>
<td></td>
<td>=</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td></td>
<td></td>
<td>=</td>
<td></td>
</tr>
</tbody>
</table>
Notables

- 18 cycles for 8 operations = 44%
- Rate that loads can be issued controls performance: must load two values before beginning the related multiply
# Example Instruction Schedule: Idle Functional Units

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Load/Store</th>
<th>Multiply</th>
<th>Add</th>
<th>Instr</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>La1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Lb1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>&gt;a1</td>
<td>La2</td>
<td></td>
<td>A++</td>
</tr>
<tr>
<td>4</td>
<td>Lb2</td>
<td>&gt;b1</td>
<td></td>
<td>B++</td>
</tr>
<tr>
<td>5</td>
<td>La3</td>
<td>&gt;a2</td>
<td>A1*b1</td>
<td>B++</td>
</tr>
<tr>
<td>6</td>
<td>&gt;b2</td>
<td>Lb3</td>
<td></td>
<td>A++</td>
</tr>
<tr>
<td>7</td>
<td>La4</td>
<td>&gt;a3</td>
<td>A2*b2</td>
<td>B++</td>
</tr>
<tr>
<td>8</td>
<td>Lb4</td>
<td>&gt;b3</td>
<td></td>
<td>A++</td>
</tr>
<tr>
<td>9</td>
<td>&gt;a4</td>
<td></td>
<td>A3*b3</td>
<td>B++</td>
</tr>
<tr>
<td>10</td>
<td>&gt;b4</td>
<td></td>
<td>+a1b1</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td>A4*b4</td>
<td>=</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td>=</td>
<td>+a2b2</td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td>+a3b3</td>
<td>=</td>
<td>N-=4</td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td>CMP</td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
<td>=</td>
<td>+a4b4</td>
</tr>
<tr>
<td>17</td>
<td></td>
<td></td>
<td></td>
<td>Branch</td>
</tr>
<tr>
<td>18</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Observations

• The dependencies between the load, multiply, and add operations mean that there is much wasted time.
• We can unroll the loop to provide more operations
• We can skew the operations across loops to hide the latency of the operations
Observations

• Simple, abstract performance model gives insight into loop performance
  ♦ Still not *predictive* – details depend on architecture
  ♦ Does expose impact of multicycle instructions, dependencies between operations

• Also illustrates that all parts of a processor are rarely busy
  ♦ Exploited in some architectures through “simultaneous multithreading”