How do we evaluate computer architectures?

- Think of 5 characteristics that differentiate computers?
  - Can some processors compute things that others can’t?
How do we evaluate computer architectures?

- Think of 5 characteristics that differentiate computers?
Two notions of performance

- Which has higher performance?

- From a passenger’s viewpoint: **latency** (time to do the task)
  - hours per flight, execution time, response time

- From an airline’s viewpoint: **throughput** (tasks per unit time)
  - passengers per hour, bandwidth

- Latency and throughput are often in opposition

<table>
<thead>
<tr>
<th>Aircraft</th>
<th>DC to Paris</th>
<th>Passengers</th>
</tr>
</thead>
<tbody>
<tr>
<td>747</td>
<td>6 hours</td>
<td>500</td>
</tr>
<tr>
<td>Concorde</td>
<td>3 hours</td>
<td>125</td>
</tr>
</tbody>
</table>
Some Definitions

- Relative performance: “x is $N$ times faster than y”

\[
\frac{\text{Performance}(x)}{\text{Performance}(y)} = N
\]

- If we are primarily concerned with latency,

\[
\text{Performance}(x) = \frac{1}{\text{Latency}(x)}
\]

- If we are primarily concerned with throughput,

\[
\text{Performance}(x) = \text{throughput}(x)
\]
The obvious metric: how long does it take to run a test program? This depends upon three factors:

1. The number of *dynamic* instructions $N$ in the program
   - Executing more instructions tends to take longer.

2. The kind of instructions in the program
   - Some instructions take more CPU cycles than others
   - Let $c$ be the *average* number of cycles per instruction (CPI)

3. The time $t$ per CPU clock cycle (clock-cycle time)

\[
\text{CPU time} = \frac{\text{Instructions executed}}{} \times \frac{\text{CPI}}{} \times \frac{\text{Clock cycle time}}{}
\]

\[
\frac{\text{Seconds}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instructions}} \times \frac{\text{Seconds}}{\text{Clock cycle}}
\]
The three components of CPU performance

- Instructions executed:
  - the **dynamic instruction count** (#instructions actually executed)
  - not the (static) number of lines of code

- Average Cycles per instruction:
  - function of the machine and program
    - CPI(floating-point operations) > CPI(integer operations)
    - Improved processor may execute same instructions in fewer cycles
  - Single-cycle machine: each instruction takes 1 cycle (CPI = 1)
    - CPI can be > 1 due to memory stalls and slow instructions
    - CPI can be < 1 on **superscalar** machines

- Clock cycle time: 1 cycle = minimum time it takes the CPU to do any work
  - clock cycle time = 1/ clock frequency
  - 500MHz processor has a cycle time of 2ns (nanoseconds)
  - 2GHz (2000MHz) CPU has a cycle time of just 0.5ns
  - higher frequency is usually better
Execution time, again

CPU time = Instructions executed \times CPI \times Clock cycle time

- Make things faster by making any component smaller!

<table>
<thead>
<tr>
<th></th>
<th>Program</th>
<th>Compiler</th>
<th>ISA</th>
<th>Organization</th>
<th>Technology</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction Executed</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPI</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Clock Cycle Time</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Often easy to reduce one component by increasing another

A: Yes       B: No
Example 1: ISA-compatible processors

- Let’s compare the performances of two x86-based processors.
  - An 800MHz AMD Duron, with a CPI of 1.2 for an MP3 compressor.
  - A 1GHz Pentium III with a CPI of 1.5 for the same program.
- Compatible processors implement identical instruction sets and will use the same executable files, with the same number of instructions.
- But they implement the ISA differently, which leads to different CPIs.

\[
\text{CPU time}_{\text{AMD},p} = \text{Instructions}_p \times \text{CPI}_{\text{AMD},p} \times \text{Cycle time}_{\text{AMD}}
\]

\[
= \\
= \\
= \\
\]

\[
\text{CPU time}_{\text{P3},p} = \text{Instructions}_p \times \text{CPI}_{\text{P3},p} \times \text{Cycle time}_{\text{P3}}
\]

\[
= \\
= \\
= \\
\]
Example 2: Comparing across ISAs

- Intel’s Itanium (IA-64) ISA is designed to facilitate executing multiple instructions per cycle. If an Itanium processor achieves an average CPI of .3 (3 instructions per cycle), how much faster is it than a Pentium4 (which uses the x86 ISA) with an average CPI of 1?

   a) Itanium is three times faster
   b) Itanium is one third as fast
   c) Not enough information
Performance Optimization

- Until you are an expert, first write a working version of the program
- Then, and only then, begin tuning, first collecting data, and iterate
  - Otherwise, you will likely optimize what doesn’t matter

“We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil.” -- Sir Tony Hoare
Building a benchmark

- You need something to gauge your progress.
  - Should be representative of how the program will be used
Instrumenting your program

- We can do this by hand. Consider: \texttt{test.c} --> \texttt{test2.c}
  - Let’s us know where the program is spending its time.
  - But implementing it is tedious; consider instrumenting 130k lines of code.
Using tools to do instrumentation

- Two GNU tools integrated into the GCC C compiler

- **Gprof: The GNU profiler**
  - Compile with the `-pg` flag
    - This flag causes gcc to keep track of which pieces of source code correspond to which chunks of object code and links in a profiling signal handler.
  - Run as normal; program requests the operating system to periodically send it signals; the signal handler records what instruction was executing when the signal was received in a file called `gmon.out`
  - Display results using gprof command
    - Shows how much time is being spent in each function.
    - Shows the calling context (the path of function calls) to the hot spot.
Example gprof output

Each sample counts as 0.01 seconds.

<table>
<thead>
<tr>
<th>% cumulative</th>
<th>time</th>
<th>seconds</th>
<th>self</th>
<th>calls</th>
<th>s/call</th>
<th>s/call</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>81.89</td>
<td>4.16</td>
<td>4.16</td>
<td>37913758</td>
<td>0.00</td>
<td>0.00</td>
<td>cache_access</td>
<td></td>
</tr>
<tr>
<td>16.14</td>
<td>4.98</td>
<td>0.82</td>
<td>1</td>
<td>0.82</td>
<td>5.08</td>
<td>sim_main</td>
<td></td>
</tr>
<tr>
<td>1.38</td>
<td>5.05</td>
<td>0.07</td>
<td>6254582</td>
<td>0.00</td>
<td>0.00</td>
<td>update_way_list</td>
<td></td>
</tr>
<tr>
<td>0.59</td>
<td>5.08</td>
<td>0.03</td>
<td>1428644</td>
<td>0.00</td>
<td>0.00</td>
<td>dl1_access_fn</td>
<td></td>
</tr>
<tr>
<td>0.00</td>
<td>5.08</td>
<td>0.00</td>
<td>711226</td>
<td>0.00</td>
<td>0.00</td>
<td>dl2_access_fn</td>
<td></td>
</tr>
<tr>
<td>0.00</td>
<td>5.08</td>
<td>0.00</td>
<td>256830</td>
<td>0.00</td>
<td>0.00</td>
<td>yylex</td>
<td></td>
</tr>
</tbody>
</table>

Over 80% of time spent in one function

Provides calling context (main calls sim_main calls cache_access) of hot spot

<table>
<thead>
<tr>
<th>index</th>
<th>% time</th>
<th>self</th>
<th>children called</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1]</td>
<td>100.0</td>
<td>0.82</td>
<td>4.26 1/1</td>
<td>main [2]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.82</td>
<td>4.26 1</td>
<td>sim_main [1]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4.18</td>
<td>0.07 36418454/36484188</td>
<td>cache_access &lt;cycle 1&gt; [4]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.00</td>
<td>0.01 10/10</td>
<td>sys_syscall [9]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.00</td>
<td>0.00 2935/2967</td>
<td>mem_translate [16]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.00</td>
<td>0.00 2794/2824</td>
<td>mem_newpage [18]</td>
</tr>
</tbody>
</table>
Using tools for instrumentation (cont.)

- Gprof didn’t give us information on where in the function we were spending time. *(cache_access is a big function; still needle in haystack)*

- Gcov: the GNU coverage tool
  - Compile/link with the `-fprofile-arcs -ftest-coverage` options
    - Adds code during compilation to add counters to every control flow edge (much like our by hand instrumentation) to compute how frequently each block of code gets executed.
  - Run as normal
  - For each `xyz.c` file an `xyz.gdna` and `xyz.gcno` file are generated
  - Post-process with `gcov xyz.c`
    - Computes execution frequency of each line of code
    - Marks with `#####` any lines not executed
      - Useful for making sure that you tested your whole program
Example gcov output

14282656:  540:  if (cp->hsize) {
    541:      int hindex = CACHE_HASH(cp, tag);
  542:
    543:      for (blk=cp->sets[set].hash[hindex];
     -:  544:          blk;
     -:  545:          blk=blk->hash_next)
    -:  546:          {
        547:              if (blk->tag == tag && (blk->status & CACHE_BLK_VALID))
        548:                  goto cache_hit;
     -:  549:          }
     -:  550:      } else {
     -:  551:          /* linear search the way list */
753030193:  552:          for (blk=cp->sets[set].way_head;
     -:  553:              blk;
     -:  554:              blk=blk->way_next)      {
751950759:  555:              if (blk->tag == tag && (blk->status & CACHE_BLK_VALID))
738747537:  556:                  goto cache_hit;
     -:  557:          }
     -:  558:      }

Loop executed over 50 interations on average (751950759/14282656)

Code never executed
Conclusion

- The second step to making a fast program is finding out why it is slow
  - The first step is making a working program
  - Your intuition where it is slow is probably wrong
    - So don’t guess, collect data!

- Many tools already exist for automatically instrumenting your code
  - Identify the “hot spots” in your code where time is being spent
  - Two example tools:
    - Gprof: periodically interrupts program
    - Gcov: inserts counters into code
  - We’ll see Vtune in section, which explains why the code is slow

- If you’ve never tuned your program, there is probably “low hanging fruit”
  - Most of the time is spent in one or two functions
  - Try using better data structures (225) or algorithms (473) to speed these up
Key Idea: **Iterative Refinement**

1. Build simplest possible implementation
2. Does it meet criteria? If so, stop.
   
   Else, what can be improved?
3. Generate ideas on how to improve it
4. Select best ideas, based on benefit/cost
5. Modify implementation based on best ideas

It is very tempting to go straight to an “optimized” solution. Pitfalls:
1. You never get anything working
2. Incomplete problem knowledge leads to selection of wrong optimizations

With iterative refinement, you can stop at any time!
   
Result is optimal for time invested.