Slide 1

Administrivia

- First homework to be on the Web soon, due a week from Tuesday. (I will send mail.)

Slide 2

Introduction

- “Computers are everywhere” — you know about desktops and servers, which are more and more central to our lives, but also consider “embedded processors”, largely invisible but even more prevalent.
- It seems to be a truism that however fast computers can process information, they can’t keep up with humans’ ability to imagine things for them to do. So performance matters.
- Factors that affect performance include both the ones you learn about in programming courses (order of magnitude of algorithms, e.g.) and lower-level ones (how well the compiler can map HLL onto hardware in various respects, how fast the hardware can execute instructions).

Slide 3

“Below Your Program” — Review?

- Most programming these days is done in a high-level language (HLL), often using a lot of library code. Processors, however, can’t execute it directly. How do we get from HLL to something the processor can do?
- First step is to *compile* — conceptually, to *assembly language* (symbolic representation of instructions the processor can execute).
- Next step is to translate assembly language into *machine language* (actual instructions for processors, in 1s and 0s), a.k.a. *object code*. Might be combined with compiling.

Slide 4

“Below Your Program”, Continued

- Final step is to combine object code for your program with library object code. Can be done as part of compiling process to create an *executable file* or at runtime, or some combination of the two.
- Actual execution of program typically involves operating system (something manages physical resources / provides abstraction for applications). Contents/format of executable files depends on operating system as well as hardware.
- Worth noting that some languages/implementations don’t exactly follow this scheme — some languages (e.g., shell scripts) are translated/interpreted at runtime, and others (e.g., Scala and Java) are compiled to machine language for a virtual processor (the JVM), which may then be translated into “native code” at runtime.

Slide 5

Hardware Components — An Abstract View

- Input devices — way to get info into computer from outside. Examples include keyboard and . . . ?
- Output devices — way to get information back to outside world. Examples include display and . . . ?
- Processor — “brain” that does actual calculations, etc. Can divide into
  - *Datapath* — stores values, performs operations (e.g., addition).
  - *Control* — “puppet master” for datapath.
- Memory — stores values, “scratch pad” for calculations. Now typically includes “main memory” and “cache memory” (possibly multiple levels).

Slide 6

Hardware Components, Continued

- Other noteworthy components (really I/O devices):
  - Storage devices (e.g., disk). (Often this is also called “memory”, especially by the nontechnical but increasingly by everyone. But in this course, no.)
  - Network interfaces.

Slide 7

“Layers of Abstraction” Idea

- Idea of “layers of abstraction” used over and over in CS.
- In software, you know how this works.

  Example — “shopping cart” abstraction, implemented using “resizable array” abstraction, implemented using “linked list” abstraction . . .

  Goal — “manage complexity” by dividing big complicated problem into manageable parts.
- Same idea can be used in hardware design, for the same reason . . .

Slide 8

“Layers of Abstraction” Idea in Hardware

- *Instruction set architecture* (ISA or architecture) — a definition/specification of how the hardware behaves, detailed enough for programming at assembly-language level.

  E.g, “x86 architecture”, “MIPS architecture”, “IBM 360 architecture”.
- *Implementations of an architecture* — actual hardware that behaves as defined. Can have many implementations of an architecture, allowing the same program executable to run on (somewhat) different hardware systems.

  E.g., Intel chips, IBM 360 family of processors.

“Layers of Abstraction” Idea in Hardware, Continued

- For programs that will run on a computer with an operating system, also define *application binary interface* (ABI) that describes application’s interface with both hardware and operating system.

Slide 9

A Little About Integrated Circuits

- Conceptual view of hardware:
  - *Transistor* — on/off switch controlled by electrical current.
  - Combine/connect a lot of transistors to get *circuit* that does interesting things (e.g., addition).
  - Put a bunch of circuits together to get a *chip* / *integrated circuit* (IC). If lots of transistors, *VLSI chip*.

Slide 10

Slide 11

A Little About Integrated Circuits, Continued

- Manufacturing process starts with a thin flat piece of silicon, adds metal and other stuff to make wires, insulators, transistors, etc.
- Of course, this is all automated! Low-level chip designers use CAD-type tools, which save designs in a standard format, which the chip designers simulate/test with other software, and then send off to be *fabricated*.
- Typically make many chips on a *wafer*, discard those with defects, bond each good one to something larger with *pins* to allow connections to other parts of computer.

Slide 12

Defining Performance

- What does it mean to say that computer A “has better performance than” computer B?
- Really — “it depends”. Some answers:
  - Computer A has better response time / smaller execution time.
  - Computer A has higher throughput.
- We’ll use execution time, and say

  $$\frac{\text{Performance}_A}{\text{Performance}_B} = n$$

  exactly when

  $$\frac{\text{Execution time}_B}{\text{Execution time}_A} = n$$

Slide 13

Measuring Performance

- If we use execution time as criterion, how to measure?
- Wall-clock time seems fairest, since it includes
  - Time for CPU to execute instructions.
  - Any waiting for memory access.
  - Any waiting for I/O.
  - Any waiting for operating system.
- Is that easy to measure reliably / repeatably?

Slide 14

Measuring Performance, Continued

- No — to get repeatable measure of wall clock time, need an otherwise unused system.
- So instead we could use “CPU performance” — amount of time CPU needs to run program. Easier to measure, more consistent, and at least says something about the processor.
- Even that, though, is not as simple as it might seem.

Measuring Performance, Continued

Slide 15

- CPU execution time for program X is given by

$$\text{CPU cycles} \times \text{clock cycle time}$$

and then CPU cycles in turn is the product of count of instructions and cycles per instruction.

- And then it *might* seem like we can say something meaningful about what happens if we change one of these numbers — but only if all other things remain the same, which might or might be true!

Evaluating / Comparing Performance

Slide 16

- Trickier than it sounds to come up with one number that means something.
- Approaches include
  - Use the actual workload, on the actual hardware platform(s), and compare times.
  - Put together a representative simulated workload — “benchmark”; run and compare times.
  - Compare code size.
  - Compare number of instructions per second (“MIPS” or “MFLOPS”).
- Alas, all of these are flawed in some way.

  (In particular, paraphrasing someone whose name I don’t remember, “peak MIPS is just the number you can’t go any faster than.”)

Slide 17

Minute Essay

- Suppose you are trying to decide which of two computers, call them `Foo` and `Bar`, will give you the best performance. You run two test programs on `Foo` and observe execution times of 10 seconds for one and 20 seconds for the other. If the first program takes 5 seconds on `Bar`, how long does the second program take? (Hint: This might be something of a trick question.)
- Other questions?

Slide 18

Minute Essay Answer

- It might seem like that second program would take 10 seconds on `Bar`, but in truth you probably can't be sure without doing the experiment, since the two machines, or the two test programs, could differ in ways that would make this obvious answer wrong.